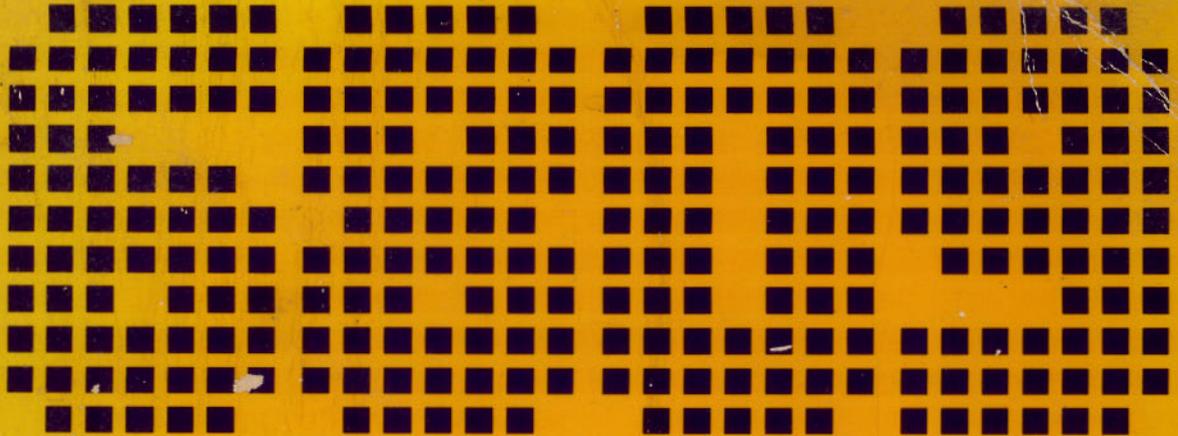


OSBORNE/McGRAW-HILL

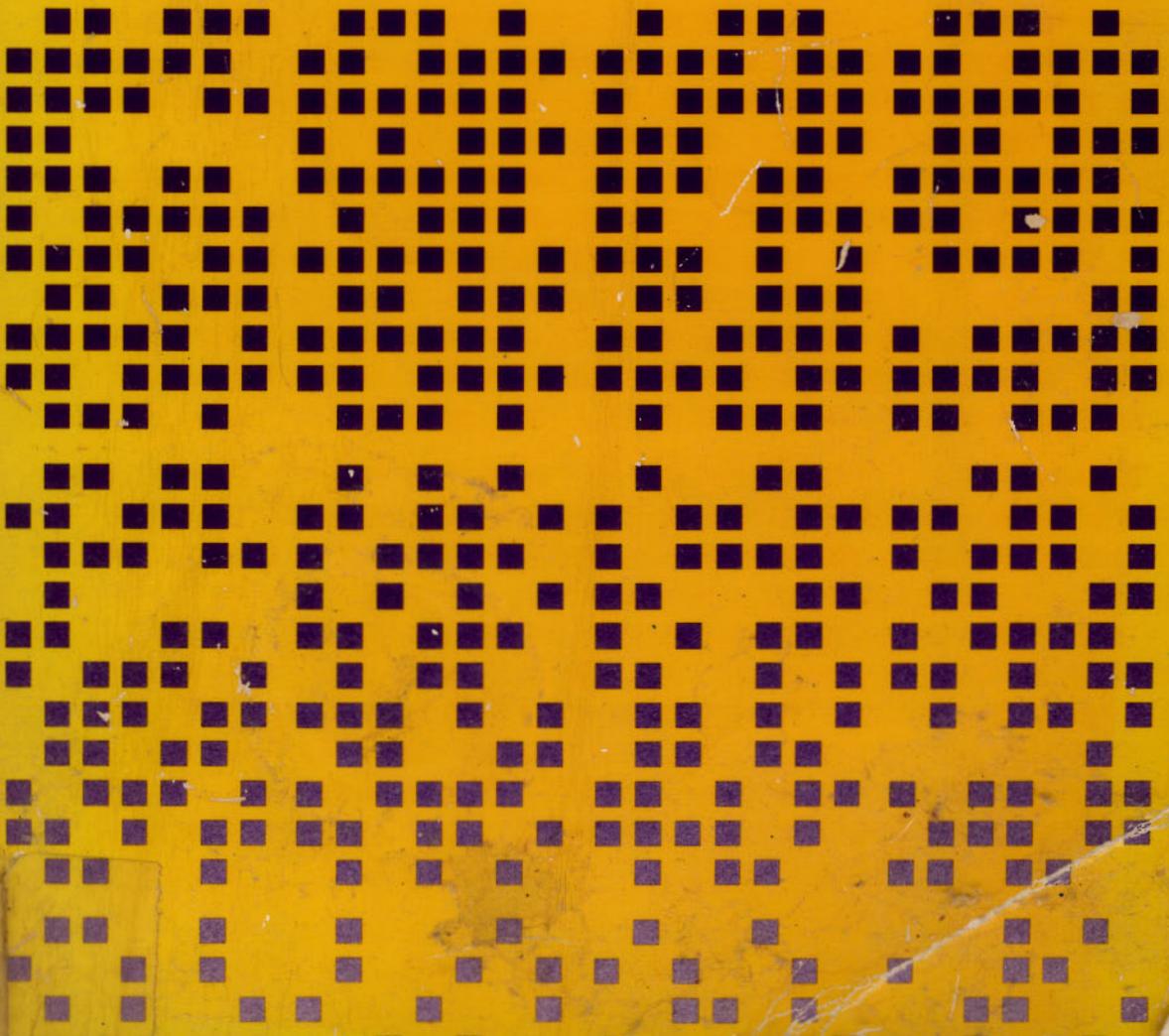


6809 ASSEMBLY LANGUAGE PROGRAMMING

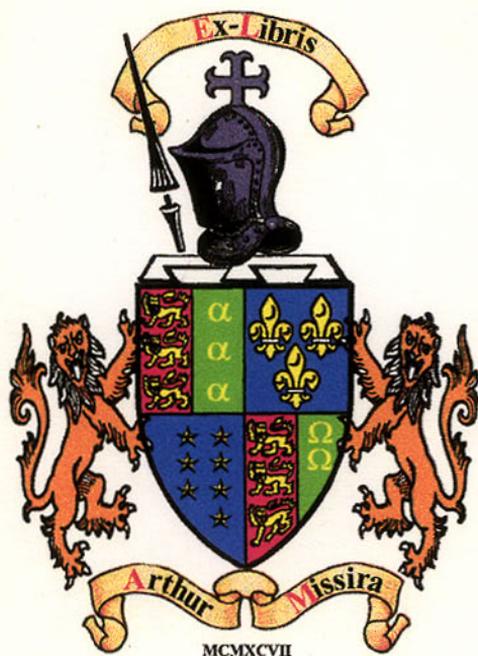
LEVENTHAL



ASSEMBLY LANGUAGE PROGRAMMING LANCE LEVENTHAL

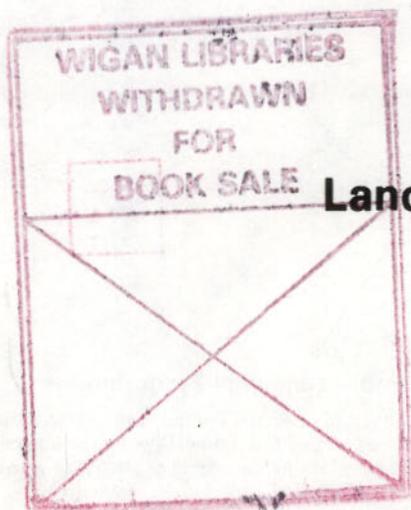


005.
265
MOT
7



O/PG

6809 Assembly Language Programming



Lance A. Leventhal

**OSBORNE/McGraw-Hill
Berkeley, California**

Published by
OSBORNE/McGraw-Hill
630 Bancroft Way
Berkeley, California 94710
U. S. A

For information on translations and book distributors outside of the U. S. A. ,
please write OSBORNE/McGraw-Hill at the above address.

05236



6809 Assembly Language Programming

Copyright © 1981 McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publishers.

1234567890 DODO 8987654321

ISBN 0-931988-35-7

Cover design by Tim Sullivan.

Acknowledgments

Mr. Irvin Stafford of Burroughs Corporation constructed the 6809-based computer, tried all the examples, and suggested numerous corrections and improvements. Mr. Curt Ingraham, Ms. Susanna Jacobson, Ms. Denise Penrose, and Ms. Janice Enger of Osborne/McGraw-Hill contributed greatly to this project; Susanna Jacobson and Curt Ingraham insisted on a high level of clarity and accuracy. Mr. Lothar Stern and Mr. Marshall Rothen of Motorola's Technical Information Center (Phoenix) were very generous in providing materials. Others who helped include Ms. Marielle Carter, Mr. Romeo Favreau, and Mr. Gary Hankins of Sorrento Valley Associates; Mr. Michael Lehman and Mr. Winthrop Saville of MT MicroSystems; and my wife Donna, who has been both patient and understanding.

Special thanks go to Mr. Terry Ritter of Motorola (Austin, Texas), the original architect of the 6809 microprocessor, who was kind enough to review the manuscript. Also Dr. Jack Lipovski of the University of Texas at Austin provided me with a preliminary version of his 6809-based book *Microcomputer Interfacing: Principles and Practices* (Lexington Books, Lexington, Mass., 1980); it is an excellent book and I have borrowed heavily from the ideas in it.

Mr. Allan Robbins, P.E., of SDS Technical Services, Ltd., Winnepeg, Canada, contributed material for Chapter 3, Chapter 22, and the appendices.

I would like to take this opportunity to thank those who have reviewed previous books in this series. In particular, I should mention Mark Bernstein, Jim Butterfield, Art Childs, James Demas, and Philip Hooper. Of course, my initial reaction to their negative comments was defensive. However, after some complaining and some prodding from my editors, I have responded to their criticism in this book. I have revised several chapters considerably and I have stressed clear, concise explanations and interesting examples. Reviewing is a thankless job, so I want these people to know that I have learned from their efforts.

This book is dedicated to the friends I made along South Highway 101 in Solana Beach, California: Don and Hazel Cahoon, Lou and Marge DiCarlo, and Bob and June Vallery.

— Lance A. Leventhal

This book's assembler listings were generated on a 6809-based EXORciser system loaned to the publisher by Motorola Microsystems, Mesa, Arizona.

The publisher also wishes to thank Mr. Bernard Löhrr for assembling and testing the programs in this book.

About the Author

Lance A. Leventhal is a partner in Emulative Systems Company, Inc., a San Diego-based consulting firm specializing in microprocessors and microprogramming. He is a national lecturer on microprocessors for the IEEE, the author of ten books and over sixty articles on microprocessors, and a regular contributor to such publications as *Simulation and Microcomputing*. He also serves as technical editor for the Society for Computer Simulation and as contributing editor for *Digital Design* magazine.

Dr. Leventhal has authored four previous books in this series and has just begun work on a new series, *Some Common Assembly Language Programs*. He received a B.A. degree from Washington University in St. Louis, and M.S. and Ph.D. degrees from the University of California at San Diego. He is a member of SCS, ACM, IEEE, and the IEEE Computer Society.

Contents

Section I. Fundamental Concepts

1. **Introduction Assembly Language Programming**
 - A Computer Program 1-2
 - High-Level Languages 1-8
2. **Assemblers**
 - Features of Assemblers 2-1
 - Types of Assemblers 2-15
 - Errors 2-16
 - Loaders 2-17
3. **6809 Machine Structure and Assembly Language**
 - 6809 Registers and Flags 3-3
 - 6809 Addressing Modes 3-6
 - Modes Which Do Not Specify Memory Locations 3-8
 - Memory Addressing Modes 3-9
 - Indexed Memory Addressing Modes 3-16
 - Program Relative Addressing for Branches 3-36
 - 6809 Instruction Set 3-38
 - 6800/6809 Compatibility 3-38
 - 6801/6809 Compatibility 3-44
 - 6502/6809 Compatibility 3-45
 - Motorola 6809 Assembler Conventions 3-45

Section II. Introductory Problems

- 4. Beginning Programs**
Program Examples 4-1
Problems 4-11
- 5. Simple Program Loops**
Program Examples 5-4
Problems 5-15
- 6. Character-Coded Data**
Handling Data in ASCII 6-1
Program Examples 6-3
Problems 6-13
- 7. Code Conversion**
Program Examples 7-2
Problems 7-10
- 8. Arithmetic Problems**
Program Examples 8-2
Problems 8-16
- 9. Tables and Lists**
Program Examples 9-1
Problems 9-14

Section III. Advanced Topics

- 10. Subroutines**
Program Examples 10-3
Position-Independent Code 10-15
Nested Subroutines 10-16
Problems 10-16
- 11. Parameter Passing Techniques**
The PSH and PUL Instructions 11-1
General Parameter Passing Techniques 11-3
Types of Parameters 11-14
- 12. Input/Output Considerations**
I/O Device Categories 12-2
Time Intervals 12-9
Logical and Physical Devices 12-13
Standard Interfaces 12-14
6809 Input/Output Chips 12-14
- 13. Using the 6820 Peripheral Interface Adapter (PIA)**
Initializing a PIA 13-6
Using the PIA to Transfer Data 13-10
Program Examples 13-1
More Complex I/O Devices 13-29
Problems 13-52

- 14. Using the 6850 Asynchronous Communications Interface Adapter (ACIA)**
 - Program Examples 14-5

- 15. Interrupts**
 - Characteristics of Interrupt Systems 15-1
 - 6809 Interrupt System 15-3
 - 6820 PIA Interrupts 15-8
 - 6850 ACIA Interrupts 15-8
 - 6809 Polling Interrupt Systems 15-9
 - 6809 Vectored Interrupt Systems 15-10
 - Communications Between Main Program and Service Routines 15-10
 - Enabling and Disabling Interrupts 15-11
 - Changing Values in the Stack 15-13
 - Interrupt Overhead 15-15
 - Program Examples 15-15
 - More General Service Routines 15-30
 - Problems 15-31

Section IV. Software Development

- 16. Problem Definition**
 - Inputs 16-1
 - Outputs 16-2
 - Processing Section 16-2
 - Error Handling 16-3
 - Human Factors/Operator Interaction 16-3
 - Examples 16-4
 - Review 16-14

- 17. Program Design**
 - Basic Principles 17-1
 - Flowcharting 17-2
 - Modular Programming 17-11
 - Structured Programming 17-15
 - Top-Down Design 17-26
 - Designing Data Structures 17-31
 - Review of Problem Definition and Program Design 17-32

- 18. Documentation**
 - Self-Documenting Programs 18-1
 - Comments 18-2
 - Flowcharts as Documentation 18-7
 - Structured Programs as Documentation 18-7
 - Memory Maps 18-7
 - Parameter and Definition Lists 18-8
 - Library Routines 18-9
 - Total Documentation 18-12

- 19. Debugging**
 - Simple Debugging Tools 19-2
 - Advanced Debugging Tools 19-8
 - Debugging With Checklists 19-10
 - Looking for Errors 19-11
 - Examples 19-17
- 20. Testing**
 - Selecting Test Data 20-2
 - Examples 20-3
 - Rules for Testing 20-4
 - Conclusions 20-4
- 21. Maintenance and Redesign**
 - Saving Memory 21-2
 - Saving Execution Time 21-4
 - Major Reorganization 21-4

Section V. 6809 Instruction Set

22. The Instruction Set

(For page number reference, see the list of 6809 instructions at the back of the book.)

Appendices

- A. Summary of the 6809 Instruction Set
- B. Summary of 6809 Indexed and Indirect Addressing Modes
- C. 6809 Instruction Codes, Memory Requirements, and Execution Times
- D. 6809 Instruction Object Codes in Numerical Order
- E. 6809 Post Bytes in Numerical Order

Index

Program Examples

- 4-1. 8-Bit Data Transfer 4-1
 - 4-2. 8-Bit Addition 4-2
 - 4-3. Shift Left One Bit 4-2
 - 4-4. Mask Off Most Significant Four Bits 4-3
 - 4-5. Clear a Memory Location 4-4
 - 4-6. Byte Disassembly 4-4
 - 4-7. Find Larger of Two Numbers 4-5
 - 4-8. 16-Bit Addition 4-7
 - 4-9. Table of Squares 4-8
 - 4-10. 16-Bit Ones Complement 4-11
-
- 5-1. Sum of Data 5-4
 - 5-2. 16-Bit Sum of Data 5-6
 - 5-3. Number of Negative Elements 5-9
 - 5-4. Maximum Value 5-10
 - 5-5. Justify a Binary Fraction 5-12
-
- 6-1. Length of a String of Characters 6-3
 - 6-2. Find First N-Blank Character 6-5
 - 6-3. Replace Leading Zeros with Blanks 6-7
 - 6-4. Add Even Parity to ASCII Characters 6-8
 - 6-5. Pattern Match 6-11
-
- 7-1. Hexadecimal to ASCII 7-2
 - 7-2. Decimal to Seven-Segment 7-3
 - 7-3. ASCII to Decimal 7-6
 - 7-4. BCD to Binary 7-8
 - 7-5. Binary Number to ASCII String 7-8

- 8-1. Multiple-Precision Binary Addition 8-2
- 8-2. Decimal Addition 8-4
- 8-3. 8-Bit by 16-Bit Binary Multiplication 8-17
- 8-4. Binary Division 8-8
- 8-5. Self-Checking Numbers Double Add Double Mod 10 8-12

- 9-1. Add Entry to List 9-1
- 9-2. Check an Ordered List 9-3
- 9-3. Remove Element from Queue 9-5
- 9-4. 8-Bit Sort 9-9
- 9-5. Using an Ordered Jump Table 9-12

- 10-1. Converting Hexadecimal to ASCII 10-3
- 10-2. Length of a String of Characters 10-7
- 10-3. Maximum Value 10-9
- 10-4. Pattern Match 10-11
- 10-5. Multiple-Precision Addition 10-13

- 13-1. A Pushbutton 13-12
- 13-2. A Multi-Position (Rotary, Selector, or Thumbwheel) Switch 13-16
- 13-3. A Single LED 13-20
- 13-4. Seven-Segment LED Display 13-22
- 13-5. An Unencoded Keyboard 13-32
- 13-6. An Encoded Keyboard 13-38
- 13-7. Digital-to-Analog Converter 13-40
- 13-8. Analog-to-Digital Converter 13-43
- 13-9. A Teletypewriter (TTY) 13-47

- 14-1. Receive Data from TTY 14-5
- 14-2. Send Data to TTY 14-6

- 15-1. A Startup Interrupt 15-15
- 15-2. A Keyboard Interrupt 15-17
- 15-3. A Printer Interrupt 15-20
- 15-4. A Real-Time Clock Interrupt 15-23
- 15-5. A Teletypewriter Interrupt 15-28

- 18-1. Commenting a Multiple-Precision Addition Routine 18-4
- 18-2. Commenting a Teletypewriter Output Routine 18-5
- 18-3. Sum of Data/Library Routine 18-10
- 18-4. Decimal to Seven-Segment Conversion/Library Routine 18-11
- 18-5. Decimal Sum/Library Routine 18-11

- 19-1. Debugging a Code Conversion Program 19-17
- 19-2. Debugging a Sort Program 19-22

- 20-1. Testing a Sort Program 20-3
- 20-2. Testing an Arithmetic Program 20-4

Fundamental Concepts

This book describes assembly language programming. It assumes that you are familiar with *An Introduction to Microcomputers: Volume 1 — Basic Concepts* (Berkeley: Osborne/McGraw-Hill, 1980). Chapters 6 and 7 of that book are especially relevant. This book does not discuss the general features of computers, microcomputers, addressing methods, or instruction sets; you should refer to *An Introduction to Microcomputers: Volume 1* for that information.

The chapters in this section provide basic information on assembly language in general and the 6809 in particular. Chapter 1 discusses the purpose of assembly language and compares it with higher-level computer languages. Chapter 2 discusses assemblers and, briefly, loaders. Chapter 3 describes the architecture of the 6809 microprocessor, compares it with similar processors, and discusses important features of Motorola's 6809 assemblers.

HOW THIS BOOK HAS BEEN PRINTED

This book contains both boldface and lightface type. The material in lightface type only expands on information presented in the previous boldface type. Thus you can skip subject areas with which you are familiar by skipping the material in lightface type. When you reach an unfamiliar subject, read both the material in boldface type and the material in lightface type.



1

Introduction to Assembly Language Programming

A computer program is ultimately a series of numbers and therefore has very little meaning to a human being. In this chapter we will discuss the levels of human-like language in which a computer program may be expressed. We will further discuss the reasons for and uses of assembly language, which is the subject of this book.

THE MEANING OF INSTRUCTIONS

The instruction set of a microprocessor is the set of binary inputs that produce defined actions during an instruction cycle. An instruction set is to a microprocessor what a function table is to a logic device such as a gate, adder, or shift register. Of course, the actions that the microprocessor performs in response to its instruction inputs are far more complex than the actions that logic devices perform in response to their inputs.

Binary Instructions

An instruction is a binary digit pattern — it must be available at the data inputs to the microprocessor at the proper time in order to be interpreted as an instruction. For example, when the 6809 microprocessor receives the 8-bit binary pattern 01001111 as the input during an instruction fetch operation, the pattern means:

“Clear (put zero in) Accumulator A”

Similarly, the pattern 10000110 means:

The microprocessor (like any other computer) only recognizes binary patterns as instructions or data; it does not recognize words or octal, decimal, or hexadecimal numbers.

A COMPUTER PROGRAM

A program is a series of instructions that causes a computer to perform a particular task.

Actually, a computer program includes more than instructions; it also contains the data and memory addresses that the microprocessor needs to accomplish the tasks defined by the instructions. Clearly, if the microprocessor is to perform an addition, it must have two numbers to add and a place to put the result. The computer program must determine the sources of the data and the destination of the result as well as the operation to be performed.

All microprocessors execute instructions sequentially unless an instruction changes the order of execution or halts the processor. That is, the processor gets its next instruction from the next higher memory address unless the current instruction specifically directs it to do otherwise.

Ultimately, every program is a set of binary numbers. For example, this is a 6809 program that adds the contents of memory locations 0060₁₆ and 0061₁₆ and places the result in memory location 0062₁₆:

```
10110110
00000000
01100000
10111011
00000000
01100001
10110111
00000000
01100010
```

This is a machine language, or object, program. If this program were entered into the memory of a 6809-based microcomputer, the microcomputer would be able to execute it directly.

THE BINARY PROGRAMMING PROBLEM

There are many difficulties associated with creating programs as object, or binary machine language, programs. These are some of the problems:

1. The programs are difficult to understand or debug. (Binary numbers all look the same, particularly after you have looked at them for a few hours.)
2. The programs are slow to enter since you must set a front panel switch for each bit and load memory one byte at a time.
3. The programs do not describe the task which you want the computer to perform in anything resembling a human-readable format.
4. The programs are long and tiresome to write.

5. The programmer often makes careless errors that are very difficult to locate and correct.

For example, the following version of the addition object program contains a single bit error. Try to find it:

```

10110110
00000000
01100000
10111011
00000000
01110001
10110111
00000000
01100010

```

Although the computer handles binary numbers with ease, people do not. People find binary programs long, tiresome, confusing, and meaningless. Eventually, a programmer may start remembering some of the binary codes, but such effort should be spent more productively.

USING OCTAL OR HEXADECIMAL

We can improve the situation somewhat by writing instructions using octal or hexadecimal numbers, rather than binary. We will use hexadecimal numbers in this book because they are shorter, and because they are the standard for the microprocessor industry. Table 1-1 defines the hexadecimal digits and their binary equivalents. The 6809 program to add two numbers now becomes:

```

B6
00
60
BB
00
61
B7
00
62

```

At the very least, the hexadecimal version is shorter to write and not quite so tiring to examine.

Table 1-1. Hexadecimal Conversion Table

Hexadecimal Digit	Binary Equivalent	Decimal Equivalent
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
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Errors are somewhat easier to find in a sequence of hexadecimal digits. The erroneous version of the addition program, in hexadecimal form, becomes:

```
B6
00
60
BB
00
71
B7
00
62
```

The mistake is far more obvious.

What do we do with this hexadecimal program? The microprocessor understands only binary instruction codes. If your front panel has a hexadecimal keyboard instead of bit switches, you can key the hexadecimal program directly into memory — the keyboard logic translates the hexadecimal digits into binary numbers. But what if your front panel has only bit switches? You can convert the hexadecimal digits to binary by yourself, but this is a repetitive, tiresome task. People who attempt it make all sorts of petty mistakes, such as looking at the wrong line, dropping a bit, or transposing a bit or a digit. Besides, once we have converted our hexadecimal program we must still place the bits in memory through the switches on the front panel.

Hexadecimal Loader

These repetitive, grueling tasks are, however, perfect jobs for a computer. The computer never gets tired or bored and never makes silly mistakes. The idea then is to write a program that accepts hexadecimal numbers, converts them into binary numbers, and places them in memory. This is a standard program provided with many microcomputers; it is called a hexadecimal loader.

The hexadecimal loader is a program like any other. It occupies memory space. In some systems, it resides in memory just long enough to load another program; in others, it occupies a reserved, read-only section of memory. Your microcomputer may not have bit switches on its front panel; it may not even have a front panel. This reflects the machine designer's decision that binary programming is not only impossibly tedious but also wholly unnecessary. The hexadecimal loader in your system may be part of a larger program called a monitor, which also provides a number of tools for program debugging and analysis.

A hexadecimal loader certainly does not solve every programming problem. The hexadecimal version of the program is still difficult to read or understand; for example, it does not distinguish operations from data or addresses, nor does the program listing provide any suggestion as to what the program does. What does B6 or 3F mean? Memorizing a card full of codes is hardly an appetizing proposition. Furthermore, the codes will be entirely different for a different microprocessor and the program will require a large amount of documentation.

INSTRUCTION CODE MNEMONICS

An obvious programming improvement is to assign a name to each instruction code. The instruction code name is called a "mnemonic" or memory jogger. The

instruction mnemonic should describe, in a minimum number of characters, what the instruction does.

Devising Mnemonics

In fact, all microprocessor manufacturers (they cannot remember hexadecimal codes either) provide a set of mnemonics for the microprocessor instruction set. **You do not have to abide by the manufacturer's mnemonics**; there is nothing sacred about them. **However, they are standard for a given microprocessor, and therefore understood by all users.** These are the instruction codes that you will find in manuals, cards, books, articles, and programs. The problem with selecting instruction mnemonics is that not all instructions have "obvious" names. Some instructions do (for example, ADD, AND, OR), others have obvious contractions (such as SUB for subtraction, XOR for exclusive-OR), while still others have neither. The result is such mnemonics as WMP, PCHL, and even SOB. Most manufacturers come up with some reasonable names and some hopeless ones. However, users who devise their own mnemonics rarely do much better.

Along with the instruction mnemonics, the manufacturer will usually assign names to the CPU registers. As with the instruction names, some register names are obvious (such as A for Accumulator) while others may have only historical significance. Again, we will use the manufacturer's suggestions simply to promote standardization.

Standard Mnemonics

There is a proposed standard set of assembly language mnemonics.¹ The amount of use that it will receive is uncertain, but it should at least serve as a basis for comparing instruction sets and for selecting mnemonics for future processors.

An Assembly Language Program

If we use standard 6809 instruction and register mnemonics, as defined by Motorola, our 6809 addition program becomes:

LDA	\$0060
ADDA	\$0061
STA	\$0062

The program is still far from obvious, but at least some parts are comprehensible. ADDA is a considerable improvement over BB. LDA and STA suggest loading and storing the contents of an accumulator. We now see that some lines are operations and others are data or addresses. **Such a program is an assembly language program.**

THE ASSEMBLER PROGRAM

How do we get the assembly language program into the computer? We have to translate it, either into hexadecimal or into binary numbers. **You can translate an assembly language program by hand**, instruction by instruction. This is called hand assembly.

The following table illustrates the hand assembly of the addition program:

Instruction Mnemonic	Addressing Method	Hexadecimal Equivalent
LDA	extended direct	B6
ADDA	extended direct	BB
STA	extended direct	B7

As with hexadecimal-to-binary conversion, hand assembly is a rote task which is uninteresting, repetitive, and subject to numerous minor errors. Picking the wrong line, transposing digits, omitting instructions, and misreading the codes are only a few of the mistakes that you may make. Most microprocessors complicate the task even further by having instructions with different lengths. Some instructions are one byte long while others may be two to five bytes long. Some instructions require data in the second and third bytes; others require memory addresses, register numbers, or who knows what?

Assembly is another rote task that we can assign to the microcomputer. The microcomputer never makes any mistakes when translating codes; it always knows how many bytes and what format each instruction requires. The program that does this job is an "assembler." The assembler program translates a user program, or "source" program written with mnemonics, into a machine language program, or "object" program, which the microcomputer can execute. The assembler's input is a source program and its output is an object program.

An assembler is a program, just as the hexadecimal loader is. However, assemblers are more expensive, occupy more memory, and require more peripherals and execution time than do hexadecimal loaders. While users may (and often do) write their own loaders, few care to write their own assemblers.

Futhermore, **assemblers have their own rules that you must learn.** These include the use of certain markers (such as spaces, commas, semicolons, or colons) in appropriate places, correct spelling, the proper control of information, and perhaps even the correct placement of names and numbers. These rules are usually simple and can be learned quickly.

Additional Features of Assemblers

Early assemblers did little more than translate the mnemonic names of instructions and registers into their binary equivalents. However, most assemblers now provide such additional features as:

- Allowing the user to assign names to memory locations, input and output devices, and even sequences of instructions
- Converting data or addresses from various number systems (for example, decimal or hexadecimal) to binary and converting characters into their ASCII or EBCDIC binary codes
- Performing some arithmetic as part of the assembly process
- Telling the loader program where in memory parts of the program or data should be placed
- Allowing the user to assign areas of memory as temporary data storage and to place fixed data in areas of program memory

- Providing the information required to include standard programs from program libraries, or programs written at some other time, in the current program
- Allowing the user to control the format of the program listing and the input and output devices employed

Choosing an Assembler

All of these features, of course, involve additional cost and memory. Microcomputers generally have much simpler assemblers than do larger computers, but the tendency is always for the size of assemblers to increase. You will often have a choice of assemblers. The important criterion is not how many off-beat features the assembler has, but rather how convenient it is to use in normal practice.

DISADVANTAGES OF ASSEMBLY LANGUAGE

The assembler, like the hexadecimal loader, does not solve all the problems of programming. One problem is the tremendous gap between the microcomputer instruction set and the tasks which the microcomputer is to perform. Computer instructions tend to do things like add the contents of two registers, shift the contents of the Accumulator one bit, or place a new value in the Program Counter. On the other hand, a user generally wants a microcomputer to do something like check if an analog reading has exceeded a threshold, look for and react to a particular command from a teletypewriter, or activate a relay at the proper time. An assembly language programmer must translate such tasks into a sequence of simple computer instructions. The translation can be a difficult, time-consuming job.

Furthermore, if you are programming in assembly language, you must have detailed knowledge of the particular microcomputer that you are using. You must know what registers and instructions the microcomputers has, precisely how the instructions affect the various registers, what addressing methods the computer uses, and a mass of other information. None of this information is relevant to the task which the microcomputer must ultimately perform.

Lack of Portability

In addition, assembly language programs are not portable. Each microcomputer has its own assembly language which reflects its own architecture. An assembly language program written for the 6809 will not run on a 6502, Z80, 8080, or 3870 microprocessor. For example, the addition program written for the 8080 would be:

```
LDA    60H
MOV    B,A
LDA    61H
ADD    B
STA    62H
```

The lack of portability not only means that you will not be able to use your assembly language program on a different microcomputer, but also that you will not be able to use any programs that were not specifically written for the microcomputer you are using. This is a particular drawback for microcomputers, since these devices are new and few assembly language programs exist for them. The result, too frequently, is that you are

on your own. If you need a program to perform a particular task, you are not likely to find it in the small program libraries that most manufacturers provide. Nor are you likely to find it in an archive, journal article, or someone's old program file. You will probably have to write it yourself.

HIGH-LEVEL LANGUAGES

The solution to many of the difficulties associated with assembly language programs is to use, instead, "high-level" or "procedure-oriented" languages. Such languages allow you to describe tasks in forms that are problem-oriented rather than computer-oriented. Each statement in a high-level language performs a recognizable function; it will generally correspond to many assembly language instructions. A program called a compiler translates the high-level language source program into object code or machine language instructions.

FORTRAN — A HIGH-LEVEL LANGUAGE

Many different high-level languages exist for different types of tasks. If, for example, you can express what you want the computer to do in algebraic notation, you can write your program in FORTRAN (Formula Translation Language), the oldest and most widely used of the high-level languages. Now, if you want to add two numbers, you just tell the computer:

```
SUM = NUMB1 + NUMB2
```

That is a lot simpler (and a lot shorter) than either the equivalent machine language program or the equivalent assembly language program. Other high-level languages include COBOL (for business applications), PASCAL (a language designed for structured programming), PL/I (a combination of FORTRAN and COBOL), APL and BASIC (popular for time-sharing systems), and C (a systems-programming language developed at Bell Telephone Laboratories).

ADVANTAGES OF HIGH-LEVEL LANGUAGES

Clearly, high-level languages make programs easier and faster to write. A common estimate is that a programmer can write a program about ten times as fast in a high-level language as in assembly language.²⁻⁴ That is just writing the program; it does not include problem definition, program design, debugging, testing, or documentation, all of which become simpler and faster. The high-level language program is, for instance, partly self-documenting. Even if you do not know FORTRAN, you probably could tell what the statement illustrated above does.

Machine Independence

High-level languages solve many other problems associated with assembly language programming. The high-level language has its own syntax (usually defined by a national or international standard). The language does not mention the instruction

set, registers, or other features of a particular computer. The compiler takes care of all such details. Programmers can concentrate on their own tasks; they do not need a detailed understanding of the underlying CPU architecture — for that matter, they do not need to know anything about the computer they are programming.

Portability

Programs written in a high-level language are portable — at least, in theory. They will run on any computer that has a standard compiler for that language.

At the same time, all previous programs written in a high-level language for prior computers are available to you when programming a new computer. This can mean thousands of programs in the case of a common language like FORTRAN or BASIC.

DISADVANTAGES OF HIGH-LEVEL LANGUAGES

If all the good things we have said about high-level languages are true — if you can write programs faster and make them portable besides — why bother with assembly languages? Who wants to worry about registers, instruction codes, mnemonics, and all that garbage! As usual, there are disadvantages that balance the advantages.

Syntax

One obvious problem is that, as with assembly language, **you have to learn the “rules” or “syntax” of any high-level language** you want to use. A high-level language has a fairly complicated set of rules. You will find that it takes a lot of time just to get a program that is syntactically correct (and even then it probably will not do what you want). A high-level computer language is like a foreign language. If you have talent, you will get used to the rules and be able to turn out programs that the compiler will accept. Still, learning the rules and trying to get the program accepted by the compiler does not contribute directly to doing your job.

Here, for example, are some FORTRAN rules:

- Labels must be numbers placed in the first five card columns
- Statements must start in column 7
- Integer variables must start with the letters I, J, K, L, M, or N

Cost of Compilers

Another obvious problem is that you need a compiler to translate programs written in a high-level language into machine language. Compilers are expensive and use a large amount of memory. While most assemblers occupy 2K to 16K bytes of memory (1K = 1024), compilers occupy 4K to 64K bytes. So the amount of overhead involved in using the compiler is rather large.

Adapting Tasks to a Language

Furthermore, **only some compilers will make the implementation of your task simpler.** FORTRAN, for example, is well-suited to problems that can be expressed as algebraic formulas. If, however, your problem is controlling a printer, editing a string of characters, or monitoring an alarm system, your problem cannot be easily expressed in algebraic notation. In fact, formulating the solution in algebraic notation may be more awkward and more difficult than formulating it in assembly language. The answer is, of course, to use a more suitable high-level language. Languages specifically designed for tasks such as those mentioned above do exist — they are called system implementation languages. However, these languages are less widely used and standardized than FORTRAN.

Inefficiency

High-level languages do not produce very efficient machine language programs. The basic reason for this is that compilation is an automatic process which is riddled with compromises to allow for many ranges of possibilities. The compiler works much like a computerized language translator — sometimes the words are right but the sounds and sentence structures are awkward. A simple compiler cannot know when a variable is no longer being used and can be discarded, when a register should be used rather than a memory location, or when variables have simple relationships. The experienced programmer can take advantage of shortcuts to shorten execution time or reduce memory usage. A few compilers (known as optimizing compilers) can also do this, but such compilers are much larger than regular compilers.

SUMMARY OF ADVANTAGES AND DISADVANTAGES

Advantages of High-Level Languages:

- Easier to learn (and teach to others)
- More convenient descriptions of tasks
- Less time spent writing programs
- Easier documentation
- Standard syntax
- Independence of the structure of a particular computer
- Portability
- Availability of library and other programs

Disadvantages of High-Level Languages:

- Special rules
- Extensive hardware and software support required
- Orientation of common languages to algebraic or business problems
- Inefficient programs
- Difficulty of optimizing code to meet time and memory requirements
- Inability to use special features of a computer conveniently

HIGH-LEVEL LANGUAGES FOR MICROPROCESSORS

Microprocessor users will encounter several special difficulties when using high-level languages. Among these are:

- **Few high-level languages exist for microprocessors.** This is particularly true for processors that are new, relatively unpopular, or intended for simple control applications.
- **Few standard languages are widely available.**
- **Compilers usually require a large amount of memory or even a completely different computer.**
- **Most microprocessor applications are not well-suited to high-level languages.**
- **Many microprocessor languages produce no object program.** That is, they translate the program and run it line by line — this is referred to as interpreting rather than compiling — or they produce an output that requires special systems software (a run-time package) to execute. Either approach may result in programs that execute slowly and use a large amount of memory. BASIC and PASCAL, the most commonly available high-level languages, generally use one of these approaches.
- **Memory costs are often critical in microprocessor applications.**

The relatively small number of high-level languages for microcomputers is a result of the short history of microprocessors and their origin in the semiconductor industry, rather than in the computer industry. Among the high-level languages that are most often available are BASIC⁵, PASCAL^{6, 7}, FORTRAN, C⁸, and the PL/I-type languages such as PL/M⁹.

Many of the high-level languages that exist do not conform to recognized standards, so the microprocessor user cannot expect to gain much program portability, access to program libraries, or use of previous experience or programs. The main advantages remaining are the reduction in programming effort, easier documentation, and the smaller amount of detailed understanding of the computer architecture that is necessary.

Overhead for High-Level Languages

The overhead involved in using a high-level language with microprocessors is considerable. Until very recently, microprocessors have been better suited to control and slow interactive applications than to the character manipulation and language analysis involved in compilation. Therefore, compilers for some microprocessors will not run on a microprocessor-based system. Instead, they require a much larger computer; that is, they are cross-compilers rather than self-compilers. A user must not only bear the expense of the larger computer, but must also transfer the program from the larger computer to the micro.

Some self-compilers are available. These compilers run on the microcomputer for which they produce object code. Unfortunately, they usually require large amounts of memory (16K or more), plus special supporting hardware and software.

Unsuitability of High-Level Languages

High-level languages also are not generally well-suited to microprocessor applications. Most of the common languages were devised either to help solve scientific problems or to handle large-scale business data processing. Few microprocessor applications fall in either of these areas. Most microprocessor applications involve sending data and control information to output devices and receiving data and status information from input devices. Often the control and status information consists of a few binary digits with very precise hardware-related meanings. If you try to write a typical control program in a high-level language, you may feel like someone who is trying to eat soup with chopsticks. For tasks in such areas as test equipment, terminals, navigation systems, signal processing, and business equipment, the high-level languages work much better than they do in instrumentation, communications, peripherals, and automotive applications.

Application Areas for Language Levels

Applications better suited to high-level languages are those which require large memories. If, as in a valve controller, electronic game, appliance controller, or small instrument, the cost of a single memory chip is important, then the inefficient memory use of high-level languages is intolerable. If, on the other hand, as in a terminal or test equipment, the system has many thousands of bytes of memory anyway, this inefficiency is not as important. Clearly the size and volume of the product are important factors as well. A large program will greatly increase the advantages of high-level languages. On the other hand, a high-volume application will mean that fixed software development costs are not as important as memory costs that are part of each system.

WHICH LEVEL SHOULD YOU USE?

Which language level you use depends on your particular application. Let us briefly note some of the factors which may favor particular levels:

Applications for Machine Language:

- Virtually no one programs in machine language because it wastes human time and is difficult to document. An assembler costs very little and greatly reduces programming time.

Applications for Assembly Language:

- Short to moderate-sized programs
- Applications where memory cost is a factor
- Real-time control applications
- Limited data processing
- High-volume applications
- Applications involving more input/output or control than computation

Applications for High-Level Language:

- Long programs

- Low-volume applications
- Applications where the amount of memory required is already very large
- Applications involving more computation than input/output or control
- Compatibility with similar applications using larger computers
- Availability of specific programs in a high-level language which can be used in the application

Other Considerations

Many other factors are also important, such as the availability of a large computer for use in development, experience with particular languages, and compatibility with other applications.

If hardware will ultimately be the largest cost in your application, or if speed is critical, you should favor assembly language. But be prepared to spend much extra time in software development in exchange for lower memory costs and higher execution speeds. If software will be the largest cost in your application, you should favor a high-level language. But be prepared to spend the extra money required for the supporting hardware and software.

Of course, no one except some theorists will object if you use both assembly and high-level languages. You can write the program originally in a high-level language and then patch some sections in assembly language.^{10, 11} However, most users prefer not to do this because it can create havoc in debugging, testing, and documentation.

FUTURE TRENDS IN LANGUAGE LEVELS

We expect the future will favor high-level languages for the following reasons:

- Programs always add extra features and grow larger
- Hardware and memory are becoming less expensive
- Software and programmers are becoming more expensive
- Memory chips are becoming available in larger sizes, at lower "per bit" cost, so actual savings in chips are less likely
- More suitable and more efficient high-level languages are being developed
- More standardization of high-level languages will occur

Assembly language programming of microprocessors will not be a dying art any more than it is for large computers. But longer programs, cheaper memory, and more expensive programmers will make software costs a larger part of most applications. The edge in many applications will therefore go to high-level languages.

WHY THIS BOOK?

If the future favors high-level languages, why have a book on assembly language programming? The reasons are:

1. Most industrial microcomputer users program in assembly language (almost two thirds, according to a recent survey).

2. Many microcomputer users will continue to program in assembly language since they need the detailed control that it provides.
3. No suitable high-level language has yet become widely available or standardized.
4. Many applications require the efficiency of assembly language.
5. An understanding of assembly language can help in evaluating high-level languages.
6. **Almost all microcomputer programmers ultimately find that they need some knowledge of assembly language**, most often to debug programs, write I/O routines, speed up or shorten critical sections of programs written in high-level languages, utilize or modify operating system functions, and understand other people's programs.

The rest of this book will deal exclusively with assemblers and assembly language programming. However, we do want readers to know that assembly language is not the only alternative. You should watch for new developments that may significantly reduce programming costs if such costs are a major factor in your application.

REFERENCES

1. W. P. Fischer, "Microprocessor Assembly Language Draft Standard," *Computer*, December 1979, pp. 96-109.
2. M. H. Halstead, *Elements of Software Science*, American Elsevier, New York, 1977.
3. L. H. Putnam and A. Fitzsimmons, "Estimating Software Costs," *Datamation*, September 1979, pp. 189-98.
4. M. Phister, Jr., *Data Processing Technology and Economics*, Santa Monica Publishing Co., Santa Monica, Calif., 1976. Also available from Digital Press, Educational Services, Digital Equipment Corp., Bedford, Mass.
5. Albrecht, Finkel, and Brown, *BASIC for Home Computers*, Wiley, New York, 1978.
6. G. M. Schneider et al., *An Introduction to Programming and Problem Solving with PASCAL*, Wiley, New York, 1978.
7. K. L. Bowles, *Microcomputer Problem Solving Using PASCAL*, Springer-Verlag, New York, 1977.
8. B. W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J., 1978.
9. D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*, Addison-Wesley, Reading, Mass., 1978.
10. P. Caudill, "Using Assembly Coding to Optimize High-Level Language Programs," *Electronics*, February 1, 1979, pp. 121-24.
11. D. B. Wecker et al., "High Level Design Language Develops Low Level Microprocessor-Independent Software," *Computer Design*, June 1979, pp. 140-49.

2

Assemblers

This chapter discusses the functions performed by assemblers, beginning with features common to most assemblers and proceeding through more elaborate capabilities such as macros and conditional assembly. You may wish to skim this chapter for the present and return to it when you feel more comfortable with the material.

FEATURES OF ASSEMBLERS

As we mentioned previously, today's assemblers do much more than translate assembly language mnemonics into binary codes. But we will describe how an assembler handles the translation of mnemonics before describing additional assembler features. Finally we will explain how assemblers are used.

ASSEMBLY LANGUAGE FIELDS

Assembly language instructions (or "statements") are divided into a number of "fields," as shown in Table 2-1.

The operation code field is the only field which can never be empty; it always contains either an instruction mnemonic or a directive to the assembler, sometimes called a "pseudo-instruction," "pseudo-operation," or "pseudo-op."

The operand or address field may contain an address or data, or it may be blank.

Table 2-1. The Fields of an Assembly Language Instruction

Label Field	Operation Code or Mnemonic Field	Operand or Address Field	Comment Field
START	LDA ADDA STA	VAL1 VAL2 SUM	LOAD FIRST NUMBER INTO A ADD SECOND NUMBER TO A STORE SUM
NEXT	?	?	NEXT INSTRUCTION
.			
.			
VAL1	RMB	1	
VAL2	RMB	1	
SUM	RMB	1	

The comment and label fields are optional. A programmer will assign a label to a statement or add a comment as a personal convenience: namely, to make the program easier to read and use.

Of course, the assembler must have some way of telling where one field ends and another begins. Assemblers that use punched card input often require that each field start in a specific card column. This is a "fixed format." However, fixed formats are inconvenient when the input medium is paper tape; fixed formats are also a nuisance to programmers. The alternative is a "free format" where the fields may appear anywhere on the line.

Delimiters

If the assembler cannot use the position on the line to tell the fields apart, it must use something else. Most assemblers use a special symbol or "delimiter" at the beginning or end of each field. The most common delimiter is the space character. Commas, periods, semicolons, colons, slashes, question marks, and other characters which would not otherwise be used in assembly language programs also may serve as delimiters. Table 2-2 lists standard 6809 assembler delimiters.

You will have to exercise a little care with delimiters. Some assemblers are fussy about extra spaces or the appearance of delimiters in comments or labels. A well-written assembler will handle these minor problems, but many assemblers are not well-written. Our recommendation is simple: avoid potential problems if you can. The following rules will help:

- Do not use extra spaces, particularly after commas that separate operands.
- Do not use delimiter characters in names or labels.
- Include standard delimiters even if your assembler does not require them. Then it will be more likely that your programs are in correct form for another assembler.

Table 2-2. Standard 6809 Assembler Delimiters

'space'	Between label and operation code, between operation code and address, and before an entry in the comment field
,	Between operands in the address field
*	Before an entire line of comment

Labels

The label field is the first field in an assembly language instruction; it may be blank. If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code resulting from that instruction will be loaded. You may subsequently use the label as an address or as data in another instruction's address field. The assembler will replace the label with the assigned value when creating an object program.

Labels are most frequently used in Jump, Call, or Branch instructions. These instructions place a new value in the Program Counter and so alter the normal sequential execution of instructions. JUMP 150₁₆ means "place the value 150₁₆ in the Program Counter." The next instruction to be executed will be the one in memory location 150₁₆. The instruction JUMP START means "place the value assigned to the label START in the Program Counter." The next instruction to be executed will be the one at the address corresponding to the label START. Table 2-3 contains an example.

Why use a label? Here are some reasons:

- A label makes a program location easier to find and remember.
- The label can easily be moved, if required, to change or correct a program. The assembler will automatically change all instructions that use the label when the program is reassembled.
- The assembler or loader can relocate the whole program by adding a constant (a "relocation constant") to each address in which a label was used. Thus we can move the program to allow for the insertion of other programs or simply to rearrange memory.
- The program is easier to use as a library program; that is, it is easier for someone else to take your program and add it to some totally different program.
- You do not have to figure out memory addresses. Figuring out memory addresses is particularly difficult with microprocessors which have instructions that vary in length.

You should assign a label to any instruction that you might want to refer to later.

The next question is how to choose a label. The assembler often places some restrictions on the number of characters (usually 5 or 6), the leading character (often must be a letter), and the trailing characters (often must be letters, numbers, or one of a few special characters). Beyond these restrictions, the choice is up to you.

Our own preference is to use labels that suggest their purpose, i.e., mnemonic labels. Typical examples are ADDW in a routine that adds one word into a sum, SRETX in a routine that searches for the ASCII character ETX, or NKEYS for a location in data memory that contains the number of key entries. Meaningful labels are easier to

Table 2-3. Assigning and Using a Label

Assembly Language Program	
START	LOAD ACCUMULATOR
	.
	.
	· (MAIN PROGRAM)
	.
	.
	JUMP START

When the machine language version of this program is executed, the instruction JUMP START causes the address of the instruction labeled START to be placed in the program counter. That instruction will then be executed.

remember and contribute to program documentation. Some programmers use a standard format for labels, such as starting with L0000. These labels are self-sequencing (you can skip a few numbers to permit insertions), but they do not help document the program.

Some label selection rules will keep you out of trouble. We recommend the following:

- Do not use labels that are the same as operation codes or other mnemonics. Most assemblers will not allow this usage; others will, but it is confusing.
- Do not use labels that are longer than the assembler permits. Assemblers have various truncation rules.
- Avoid special characters (non-alphabetic and non-numeric) and lower-case letters. Some assemblers will not permit them; others allow only certain ones. The simplest practice is to stick to capital letters and numbers.
- Start each label with a letter. Such labels are always acceptable.
- Do not use labels that could be confused with each other. Avoid the letters I, O, and Z and the numbers 0, 1, and 2. Also avoid things like XXXX and XXXXX. There's no sense in tempting fate and Murphy's Law.
- When you are not sure if a label is legal, do not use it. You will not get any real benefit from discovering exactly what the assembler will accept.

These are recommendations, not rules. You do not have to follow them but don't blame us if you waste time on unnecessary problems.

ASSEMBLER OPERATION CODES (MNEMONICS)

The main task of the assembler is the translation of mnemonic operation codes into their binary equivalents. The assembler performs this task using a fixed table much as you would if you were doing the assembly by hand.

The assembler must, however, do more than just translate the operation codes. It must also somehow determine how many operands the instruction requires and what type they are. This may be rather complex — some instructions (like a Halt) have no

operands, others (like an Addition or a Jump instruction) have one, while still others (like a transfer between registers or a multiple-bit shift) require two. Some instructions may even allow alternatives; for example, some computers have instructions (like Shift or Clear) which can either apply to the Accumulator or to a memory location. We will not discuss how the assembler makes these distinctions; we will just note that it must do so.

ASSEMBLER DIRECTIVES

Some assembly language instructions are not directly translated into machine language instructions. These instructions are directives to the assembler; they assign the program to certain areas in memory, define symbols, designate areas of RAM for temporary data storage, place tables or other fixed data in memory, allow references to other programs, and perform minor housekeeping functions.

To use these assembler directives or pseudo-operations a programmer places the directive's mnemonic in the operation code field, and, if the specified directive requires it, an address or data in the address field.

The most common directives are:

DATA
EQUATE (=) or DEFINE
ORIGIN
RESERVE

Linking directives (used to connect separate programs) are:

ENTRY
EXTERNAL

Different assemblers use different names for those operations but their functions are the same. Housekeeping directives include:

END
LIST
NAME
PAGE
SPACE
TITLE
PUNCH

We will discuss these pseudo-operations briefly, although their functions are usually obvious.

The DATA Directive

The **DATA** directive allows the programmer to enter fixed data into program memory. This data may include:

- Lookup tables
- Code conversion tables
- Messages
- Synchronization patterns
- Thresholds
- Names
- Coefficients for equations
- Commands
- Conversion factors
- Weighting factors
- Characteristic times or frequencies
- Subroutine addresses
- Key identifications
- Test patterns
- Character generation patterns
- Identification patterns
- Tax tables
- Standard forms
- Masking patterns
- State transition tables

The **DATA** directive treats the data as a permanent part of the program.

The format of a **DATA** directive is usually quite simple. An instruction like:

```
DZCON DATA 12
```

will place the number 12 in the next available memory location and assign that location the name **DZCON**. Every **DATA** directive usually has a label, unless it is one of a series. The data and label may take any form that the assembler permits.

Most assemblers allow more elaborate **DATA** directives that handle a large amount of data at one time, for example:

```
EMESS DATA 'ERROR'  
SQRS DATA 1,4,9,16,25
```

A single directive may fill many bytes of program memory, limited perhaps by the length of a line or by the restrictions of a particular assembler. Of course, you can always overcome any restrictions by following one **DATA** directive with another:

```
MESSG DATA 'NOW IS THE '  
DATA 'TIME FOR ALL '  
DATA 'GOOD MEN '  
DATA 'TO COME TO THE '  
DATA 'AID OF THEIR '  
DATA 'COUNTRY '
```

Microprocessor assemblers typically have some variations of standard DATA directives. DEFINE BYTE or FORM CONSTANT BYTE handles 8-bit numbers; DEFINE WORD or FORM CONSTANT WORD handles 16-bit numbers or addresses. Other special directives may handle character-coded data.

The EQUATE (or DEFINE) Directive

The EQUATE directive allows the programmer to equate names with addresses or data. This pseudo-operation is almost always given the mnemonic EQU or =. The names may refer to device addresses, numeric data, starting addresses, fixed addresses, etc.

The EQUATE directive assigns the numeric value in its operand field to the label in its label field. Here are two examples:

```
TTY    EQU    5
LAST   EQU    5000
```

Most assemblers will allow you to define one label in terms of another, for example:

```
LAST   EQU    FINAL
ST1    EQU    START+1
```

The label in the operand field must, of course, have been previously defined. Often, the operand field may contain more complex expressions, as we shall see later. Double name assignments (two names for the same data or address) may be useful in patching together programs that use different names for the same variable (or different spellings of what was supposed to be the same name).

Note that an EQU directive does not cause the assembler to place anything in memory. The assembler simply enters an additional name into a table (called a "symbol table") which the assembler maintains. This table, unlike the mnemonic table, must be in RAM since it varies with each program. The assembler always needs some RAM to hold the symbol table; the more RAM it has, the more symbols it can accept. This RAM is in addition to any that the assembler needs as temporary storage.

When do you use a name? The answer is: whenever you have a parameter that you might want to change or that has some meaning besides its ordinary numeric value. We typically assign names to time constants, device addresses, masking patterns, conversion factors, and the like. A name like DELAY, TTY, KBD, KROW, or OPEN not only makes the parameter easier to change, but it also adds to program documentation. We also assign names to memory locations that have special purposes; they may hold data, mark the start of the program, or be available for intermediate storage.

What name do you use? The best rules are much the same as in the case of labels, except that here meaningful names really count. Why not call the teletypewriter TTY instead of X15, a bit time delay BTIME or BTDLY rather than WW, the number of the "GO" key on a keyboard GOKEY rather than HORSE? This advice seems straightforward, but a surprising number of programmers do not follow it.

Where do you place the EQUATE directives? The best place is at the start of the program, under appropriate comment headings such as I/O ADDRESSES, TEMPORARY STORAGE, TIME CONSTANTS, or PROGRAM LOCATIONS. This makes the definitions easy to find if you want to change them. Furthermore, another user will be able to look up all the definitions in one centralized place. Clearly this practice improves documentation and makes the program easier to use.

Definitions used only in a specific subroutine should appear at the start of the subroutine.

The ORIGIN Directive

The **ORIGIN** directive (almost always abbreviated **ORG**) allows the programmer to specify the memory locations where programs, subroutines, or data will reside. Programs and data may be located in different areas of memory depending on the memory configuration. Startup routines, interrupt service routines, and other required programs may be scattered around memory at fixed or convenient addresses.

The assembler maintains a **Location Counter** (comparable to the computer's **Program Counter**) which contains the location in memory of the next instruction or data item being processed. An **ORG** directive causes the Assembler to place a new value in the **Location Counter**, much as a **Jump** instruction causes the CPU to place a new value in the **Program Counter**. The output from the Assembler must not only contain instructions and data, but must also indicate to the loader program where in memory it should place the instructions and data.

Microprocessor programs often contain several **ORIGIN** statements for the following purposes:

- Reset (startup) address
- Interrupt service addresses
- Trap (software interrupt) addresses
- RAM storage
- Memory stack
- Main program
- Subroutines
- Memory addresses used for input/output devices or special functions

Still other **ORIGIN** statements may allow room for later insertions, place tables or data in memory, or assign vacant RAM space for data buffers. Program and data memory in microcomputers may occupy widely scattered addresses to simplify the hardware.

Typical ORIGIN statements are:

```
ORG RESET
ORG 1000
ORG INT3
```

Some assemblers assume an origin of zero if the programmer does not put an **ORG** statement at the start of the program. The convenience is slight; we recommend the inclusion of an **ORG** statement to avoid confusion.

The RESERVE Directive

The **RESERVE** directive allows the programmer to allocate RAM for various purposes such as data tables, temporary storage, indirect addresses, a Stack, etc.

Using the **RESERVE** directive, you assign a name to the memory area and declare the number of locations to be assigned. Here are some examples:

```
NOKEY RESERVE 1
TEMP RESERVE 50
VOLTG RESERVE 80
BUFR RESERVE 100
```

You can use the **RESERVE** directive to reserve memory locations in program memory or in data memory; however, the **RESERVE** directive is more meaningful when applied to data memory.

In reality, all the RESERVE directive does is increase the assembler's Location Counter by the amount declared in the operand field. The assembler does not actually produce any object code.

Note the following features of RESERVE:

1. **The label of the RESERVE directive is assigned the value of the first address reserved.** For example, the pseudo-operation:

```
TEMP RESERVE 20
```

reserves 20 bytes of RAM and assigns the name TEMP to the address of the first byte.

2. **You must specify the number of locations to be reserved. There is no default case.**
3. **No data is placed in the reserved locations.** Any data that, by chance, may be in these locations will be left there.

Some assemblers allow the programmer to specify initial values for the RESERVE area in RAM. We strongly recommend that you do not use this feature; it assumes that the program (along with the initial values) will be loaded from an external device (e.g., paper tape or floppy disk) each time it is run. Microprocessor programs, on the other hand, often reside in non-volatile ROM and start when power comes on. The RAM in such situations does not retain its contents, nor is it reloaded. Therefore, always include instruction sequences to initialize RAM in your program; this will insure that initialization occurs every time the program is executed and not just during load time.

Linking Directives

We often want statements in one program or subroutine to use names that are defined in a different assembly. Such uses are called "external references"; a special linking program is necessary to actually fill in the values and determine if any names are undefined or doubly defined.

The directive EXTERNAL, usually abbreviated EXT, signifies that the name is defined elsewhere.

The directive ENTRY, usually abbreviated ENT, signifies that the name is available for use elsewhere; that is, it is defined in this program.

The precise way in which linking directives are implemented varies greatly from assembler to assembler. We will not refer to such directives again, but they are very useful in actual applications.

Output Control Directives

There are various assembler directives that affect the operation of the assembler and its program listing rather than the output program itself. Common house-keeping directives include:

- END, which marks the end of the assembly language source program.
- LIST, which tells the Assembler to print the source program. Some assemblers

allow such variations as NO LIST or LIST SYMBOL TABLE to avoid long, repetitive listings.

- NAME or TITLE, which prints a name at the top of each page of the listing.
- PAGE or SPACE, which skips to the next page or next line, respectively, and improves the appearance of the listing, making it easier to read.
- PUNCH, which transfers subsequent object code to the paper tape punch. This pseudo-operation may in some cases be the default option and therefore unnecessary.

When to Use Labels

Users often wonder if or when they can assign a label to an assembler directive. These are our recommendations:

- All EQUATE directives must have labels; they are useless otherwise, since the purpose of an EQUATE is to define its label.
- DATA and RESERVE directives usually have labels. The label identifies the first memory location used or assigned.
- Other directives should not have labels. Some assemblers allow such labels, but we recommend against their use because there is no standard way to interpret them.

OPERANDS AND ADDRESSES

Most assemblers allow the programmer a lot of freedom in describing the contents of the Operand or Address field. But remember that the assembler has built-in names for registers and instructions and may have other built-in names. We will now describe some common options for the operand field.

Decimal Numbers

Most assemblers assume all numbers to be decimal unless they are marked otherwise. So:

```
ADD 100
```

means "add the contents of memory location 100_{10} to the contents of the Accumulator."

Other Number Systems

Most assemblers will also accept binary, octal, or hexadecimal entries. But you must identify these number systems in some way: for example, by preceding or following the number with an identifying character or letter. Here are some common identifiers:

- B or % for binary
- O, @, Q, or C for octal (the letter O should be avoided because of the confusion with zero)

- H or \$ for hexadecimal (or standard BCD)
- D for decimal. D may be omitted; it is the default case.

Assemblers generally require hexadecimal numbers to start with a digit (for example, 0A36 instead of A36) in order to distinguish between numbers and names or labels. It is good practice to enter numbers in the base in which their meaning is the clearest: that is, decimal constants in decimal; addresses and BCD numbers in hexadecimal; masking patterns or bit outputs in binary if they are short, and in hexadecimal if they are long.

Names

Names can appear in the operand field; they will be treated as the data that they represent. Remember, however, that **there is a difference between operands and addresses.** In a 6809 assembly language program the sequence:

```
FIVE EQU 5
      ADDA FIVE
```

will add the contents of memory location 5 (not necessarily the number 5) to the contents of the accumulator. A sequence which adds in the number 5 itself would be

```
FIVE EQU 5
      ADDA #FIVE
```

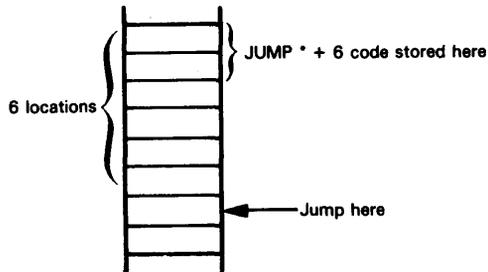
The symbol # tells the assembler that the number represented by the name FIVE is the value of the operand instead of its memory location.

The Location Counter

You can use **the current value of the location counter, which is usually referred to as * or \$.** This is useful mainly in Jump instructions; for example:

```
JUMP *+6
```

causes a Jump to the memory location 6 bytes beyond the byte that contains the first byte of the JUMP instruction.



One reason to use this technique is to reduce the number of symbols in an assembly language program. This may be necessary if the assembler can handle only a limited number of symbols. Reducing the number of symbols may also decrease assembly time. Such benefits are almost negligible, however, unless your program is extremely large or your assembler rather primitive.

Most microprocessors have many two and three-byte instructions. Thus you will have difficulty determining exactly how far apart two assembly language statements are.

Using offsets from the location counter therefore frequently results in errors that you can avoid if you use labels.

Character Codes

Most assemblers allow text to be entered as ASCII strings. Such strings may be surrounded either with single or double quotation marks; strings may also use a beginning or ending symbol such as A or C. **A few assemblers also permit EBCDIC strings.**

We recommend that you use character strings for all text. It improves the clarity and readability of the program.

Arithmetic and Logical Expressions

Assemblers permit combinations of the data forms described above, connected by arithmetic, logical, or special operators. These combinations are called expressions. Almost all assemblers allow simple arithmetic expressions such as `START + 1`. Some assemblers also permit multiplication, division, logical functions, shifts, etc. Note that the assembler evaluates expressions at assembly time. Even though an expression in the operand field may involve division, you may not be able to use division in the logic of your own program — unless you write a subroutine for that specific purpose.

Assemblers vary in what expressions they accept and how they interpret them. Complex expressions make a program difficult to read and understand.

We have made some recommendations during this section but will repeat them and add others here. **In general, the user should strive for clarity and simplicity.** There is no payoff for being an expert in the intricacies of an assembler or in having the most complex expression on the block. **We suggest the following approach:**

- Use the clearest number system or character code for data.
- Masks and BCD numbers in decimal, ASCII characters in octal, or ordinary numerical constants in hexadecimal serve no purpose and therefore should not be used.
- Remember to distinguish data from addresses.
- Don't use offsets from the Location Counter.
- Keep expressions simple and obvious. Don't rely on obscure features of the assembler.

CONDITIONAL ASSEMBLY

Some assemblers allow you to include or exclude parts of the source program, depending on conditions existing at assembly time. This is called conditional assembly; it gives the assembler some of the flexibility of a compiler. Most microcomputer assemblers have limited capabilities for conditional assembly. A typical form is:

```
IF      COND
.
.      (CONDITIONAL PROGRAM)
.
.
ENDIF
```

If the expression COND is true at assembly time, the instructions between IF and ENDIF (two pseudo-operations) are included in the program.

Typical uses of conditional assembly are:

- To include or exclude extra variables
- To place diagnostics or special conditions in test runs
- To allow data of various bit lengths

Unfortunately, conditional assembly tends to clutter programs and make them difficult to read. Use conditional assembly only if it is necessary.

MACROS

You will often find that particular sequences of instructions occur many times in a source program. Repeated instruction sequences may reflect the needs of your program logic, or they may be compensating for deficiencies in your microprocessor's instruction set. You can avoid repeatedly writing out the same instruction sequence by using a "macro."

Macros allow you to assign a name to an instruction sequence. You then use the macro name in your source program instead of the repeated instruction sequence. The assembler will replace the macro name with the appropriate sequence of instructions. The shaded parts of Figure 2-1 illustrate the assembler's treatment of a macro in an example program. Do not bother trying to figure out what the program or the instructions do; just observe that the assembler expands the macro MAC1 into the defined sequence.

A macro resembles a subroutine because it is a shorthand reference to a frequently used instruction sequence. However, macros are not the same as subroutines. The code for a subroutine occurs once in a program, and program execution branches to the subroutine. In contrast, the assembler replaces each occurrence of a macro name with the specified sequence of instructions; therefore program execution does not branch to a macro as it does to a subroutine. A macro name is a user-defined assembler directive; it directs assembly rather than program execution.

Advantages of Macros:

- Shorter source programs
- Better program documentation
- Use of debugged instruction sequences. Once the macro has been debugged, you are sure of an error-free instruction sequence every time you use the macro.
- Easier changes. Change the macro definition and the assembler makes the change for you every time the macro is used.
- Inclusion of commands, keywords, or other computer instructions in the basic instruction set. You can use macros to extend or clarify the instruction set.

Assembler Input		Assembler Output	
Source Program		Object Code	Corresponding Mnemonics
MAC1	MACR (Macro definition)		
	CLRA		
	SUBA ,Y+		
	ASLA		
	ENDM (End of macro definition)		
	(Beginning of main program)	E6 9F 2025	LDB [OFFSET]
		3A	ABX
		E6 84	LDB ,X
	LDB [OFFSET]	4F	MAC1
	LDB ,X	A0 A0	CLRA
	MAC1	48	SUBA ,Y+
	MUL	3D	ASLA
	STD RESLOC	FD 2027	MUL
	ROLB	59	STD RESLOC
	ADDB #1	CB 01	ROLB
	MAC1		ADDB #1
	ADDD RESLOC	4F	MAC1
	STD RESLOC	A0 A0	CLRA
	MAC1	48	SUBA ,Y+
	STA BYTE	F3 2027	ASLA
	BCC ALTER	FD 2027	ADDD RESLOC
		4F	STD RESLOC
		A0 A0	MAC1
		48	CLRA
		B7 2029	SUBA ,Y+
		24 05	ASLA
			STA BYTE
			BCC ALTER

Figure 2-1. Expansion of a Macro by the Assembler

Disadvantages of Macros:

- Since the macro is expanded every time it is used, memory space may be wasted by the repetition of instruction sequences.
- A single macro may create a lot of instructions.
- Lack of standardization makes programs difficult to read and understand.
- Possible effects on registers and flags may not be clearly described.

One problem is that variables used in a macro are only known within it (i.e., they are local rather than global). This can often create a great deal of confusion without any gain in return. You should be aware of this problem when using macros.¹

COMMENTS

All assemblers allow you to place comments in a source program. Comments have no effect on the object code, but they help you to read, understand, and document the program. Good commenting is an essential part of writing computer programs; programs without comments are very difficult to understand.

We will discuss commenting along with documentation in a later chapter, but here are some guidelines:

- Use comments to tell what application task the program is performing, not how the microcomputer executes the instructions.

Comments should say things like "IS TEMPERATURE ABOVE LIMIT?," "LINE FEED TO TTY," or "EXAMINE LOAD SWITCH."

Comments should not say things like "ADD 1 TO ACCUMULATOR," "JUMP TO START," or "LOOK AT CARRY." You should describe how the program is affecting the system; internal effects on the CPU are seldom of any interest.

- Keep comments brief and to the point. Details should be available elsewhere in the documentation.
- Comment all key points.
- Do not comment standard instructions or sequences that change counters or pointers; pay special attention to instructions that may not have an obvious meaning.
- Do not use obscure abbreviations.
- Make the comments neat and readable.
- Comment all definitions, describing their purposes. Also mark all tables and data storage areas.
- Comment sections of the program as well as individual instructions.
- Be consistent in your terminology. You can (should) be repetitive; you need not consult a thesaurus.
- Leave yourself notes at points that you find confusing: for example, "REMEMBER CARRY WAS SET BY LAST INSTRUCTION." If such points get cleared up later in program development, you may drop these comments in the final documentation.

A well-commented program is easy to use. You will recover the time spent in commenting many times over. We will try to show good commenting style in the programming examples, although we often over-comment for instructional purposes.

TYPES OF ASSEMBLERS

Although all assemblers perform the same tasks, their implementations vary greatly. We will not try to describe all the existing types of assemblers; we will merely define the terms and indicate some of the choices.

A cross-assembler is an assembler that runs on a computer other than the one for which it assembles object programs.

The computer on which the cross-assembler runs is typically a large computer with extensive software support and fast peripherals — such as an IBM 360 or 370, a Univac 1108, or a Burroughs 6700. The computer for which the cross-assembler assembles programs is typically a micro like the 6809 or 8080. Most cross-assemblers are written in FORTRAN so that they are portable.

A self-assembler or resident assembler is an assembler that runs on the computer for which it assembles programs. The self-assembler will require some memory and peripherals, and it may run quite slowly compared to a cross-assembler.

A macroassembler is an assembler that allows you to define sequences of instructions as macros.

A microassembler is an assembler used to write the microprograms which define the instruction set of a computer. Microprogramming has nothing specifically to do with programming microcomputers.^{2,3}

A meta-assembler is an assembler that can handle many different instruction sets. The user must define the particular instruction set being used.

A one-pass assembler is an assembler that goes through the assembly language program only once. Such an assembler must have some way of resolving forward references, for example, Jump instructions which use labels that have not yet been defined.

A two-pass assembler is an assembler that goes through the assembly language source program twice. The first time the assembler simply collects and defines all the symbols; the second time it replaces the references with the actual definitions. A two-pass assembler has no problems with forward references but may be quite slow if no backup storage (like a floppy disk) is available; then the assembler must physically read the program twice from a slow input medium (like a teletypewriter paper tape reader). Most microprocessor-based assemblers require two passes.

ERRORS

Assemblers normally provide error messages, often consisting of a single coded letter. Some typical errors are:

- Undefined name (often a misspelling or an omitted definition)
- Illegal character (such as a 2 in a binary number)
- Illegal format (wrong delimiter or incorrect operands)
- Invalid expression (for example, two operators in a row)
- Illegal value (usually too large)
- Missing operand
- Double definition (two different values assigned to one name)
- Illegal label (such as a label on a pseudo-operation that cannot have one)
- Missing label
- Undefined operation code.

In interpreting assembler errors, you must remember that the assembler may get on the wrong track if it finds a stray letter, an extra space, or incorrect punctuation. Many assemblers will then proceed to misinterpret the succeeding instructions and produce meaningless error messages. Always look at the first error very carefully; subsequent ones may depend on it. Caution and consistent adherence to standard formats will eliminate many annoying mistakes.

LOADERS

The loader is the program which actually takes the output (object code) from the assembler and places it in memory. Loaders range from the very simple to the very complex. We will describe a few different types.

A “bootstrap loader” is a program that uses its own first few instructions to load the rest of itself or another loader program into memory. The bootstrap loader may be in ROM, or you may have to enter it into the computer memory using front panel switches. The assembler may place a bootstrap loader at the start of the object program that it produces.

A “relocating loader” can load programs anywhere in memory. It typically loads each program into the memory space immediately following that used by the previous program. The programs, however, must themselves be capable of being moved around in this way; that is, they must be relocatable. An “absolute loader,” in contrast, will always place the programs in the same area of memory.

A “linking loader” loads programs and subroutines that have been assembled separately; it resolves cross-references — that is, instructions in one program that refer to a label in another program. Object programs loaded by a linking loader must be created by an assembler that allows external references. An alternative approach is to separate the linking and loading functions and have the linking performed by a program called a “link editor.”

REFERENCES

1. A complete monograph on macros is M. Campbell-Kelly, *An Introduction to Macros*, American Elsevier, New York, 1973.
2. A. Osborne, *An Introduction to Microcomputers: Volume 1 — Basic Concepts*, Osborne/McGraw-Hill, Berkeley, Calif., 1980.
3. A. K. Agrawala and T. G. Rauscher, *Foundations of Microprogramming*, Academic Press, New York, 1976.
4. D. W. Barron, *Assemblers and Loaders*, American Elsevier, New York, 1972.
5. C. W. Gear, *Computer Organization and Programming*, McGraw-Hill, New York, 1974.



3

6809 Machine Structure and Assembly Language

This chapter outlines the 6809 processor's architecture and describes the syntax rules of the Motorola assembler. Chapter 9 of *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*¹ describes the hardware aspects of the 6809 microprocessor, including its output signals and interfaces. This book considers the 6809 from the point of view of the assembly language programmer, to whom pins and signals are irrelevant and microcomputers and minicomputers are essentially identical. Later chapters of this book describe the 6809's stack and interrupt system in more detail.

Tables 3-1 through 3-3 divide the 6809 instruction set into instructions that are frequently used (Table 3-1), occasionally used (Table 3-2), and seldom used (Table 3-3). If you are an experienced assembly language programmer, you will probably not find this division important; you may even disagree with it. However, **if you are a novice, we recommend that you write your first programs using only the frequently used instructions (Table 3-1).** This restriction will help you overcome the obstacle of learning both the entire 6809 instruction set and the basic methods of assembly language programming at the same time. Once you have mastered the concepts of assembly language programming, you should start using other instructions (Tables 3-2 and 3-3).

3-2 6809 Assembly Language Programming

Table 3-1. Frequently Used Instructions of the 6809

Operation Code	Meaning
ADC	Add with Carry
ADD	Add
AND	Logical AND
ASL or LSL	Arithmetic (Logical) Shift Left
BCC or BHS	Branch if Carry Clear ("Higher or Same")
BCS or BLO	Branch if Carry Set ("Lower")
BEQ	Branch if Zero Set ("Equal")
BMI	Branch if Sign (Negative) Set ("Minus")
BNE	Branch if Zero Clear ("Not Equal")
BPL	Branch if Sign (Negative) Clear ("Plus")
BRA	Branch Always
BSR	Branch to Subroutine
CLR	Clear
CMP	Compare
DEC	Decrement by 1
INC	Increment by 1
JSR	Jump to Subroutine
LD	Load
LSR	Logical Shift Right
PSH	Push Data onto Stack
PUL	Pull Data from Stack
ROL	Rotate Left
ROR	Rotate Right
RTS	Return from Subroutine
ST	Store
SUB	Subtract

Table 3-2. Occasionally Used Instructions of the 6809

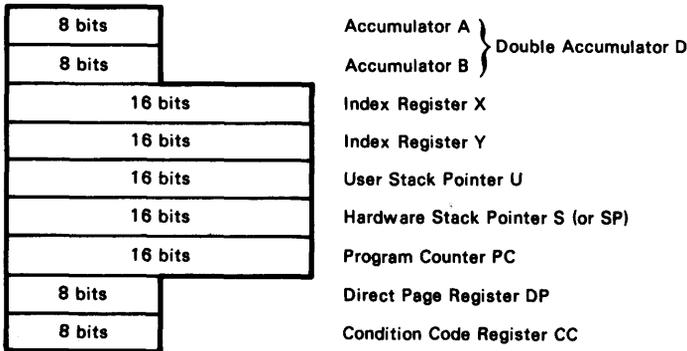
Operation Code	Meaning
ANDCC	Logical AND Mask with Status Register (Clear Flags)
ASR	Arithmetic Shift Right
BGE	Branch if Greater Than or Equal
BGT	Branch if Greater Than
BHI	Branch if Higher
BIT	Bit Test (Logical AND)
BLE	Branch if Less Than or Equal
BLS	Branch if Lower or Same
BLT	Branch if Less Than
COM	Ones Complement
DAA	Decimal Adjust Accumulator A
EOR	Exclusive OR
EXG	Exchange Registers
JMP	Jump
LEA	Load Effective Address
MUL	Multiply
NEG	Twos Complement ("Negate")
NOP	No Operation
OR	Logical (Inclusive) OR
ORCC	Logical OR Mask with Status Register (Set Flags)
RTI	Return from Interrupt
SWI	Software Interrupt
TFR	Transfer Register to Register
TST	Test for Zero or Minus

Table 3-3. Seldom Used Instructions of the 6809

Operation Code	Meaning
ABX	Add Accumulator B to Index Register X
BRN	Branch Never (No Operation)
BVC	Branch if Overflow Clear
BVS	Branch if Overflow Set
CWAI	Clear Condition Code Register Bits and Wait for Interrupt
SBC	Subtract with Carry (Borrow)
SEX	Sign Extend
SYNC	Synchronize with Interrupt Line

6809 REGISTERS AND FLAGS

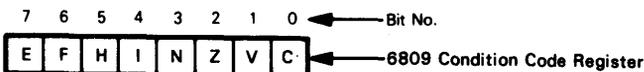
The 6809 microprocessor has two accumulators, a status (or “condition code”) register, two index registers, two stack pointers, a program counter, and a direct page register. The following diagram summarizes the 6809 registers. Note that the index registers, stack pointers, and program counter are 16 bits long, whereas the accumulators, direct page register, and condition code register are eight bits long.



The 6809’s Condition Code register contains five status flags, two interrupt control bits (one for the regular IRQ interrupt and one for the fast FIRQ interrupt), and one bit used to differentiate between the regular and fast interrupts. The five status flags are:

- Carry/Borrow (C)
- Overflow (O or V)
- Zero (Z)
- Sign (S or N for Negative)
- Half-Carry (H)

The flags occupy the following bit positions in the Condition Code register:



E is the entire flag used to differentiate between regular and fast interrupts, F is the fast interrupt mask bit, and I is the regular interrupt mask bit.

6809 REGISTERS

The two accumulators, A and B, are both primary accumulators. The only instructions that treat the accumulators differently are ABX (Add Accumulator B to Index Register X), DAA (Decimal Adjust Accumulator A), and SEX (Sign Extend Accumulator B into Accumulator A).

The two 8-bit Accumulators A and B can be referred to as a single 16-bit Double Accumulator D. Within D, A contains the most significant bits and B the least significant bits. The 6809 has special instructions for loading (LDD), storing (STD), adding (ADDD), comparing (CMPD), and subtracting (SUBD) the Double Accumulator D.

Index Registers X and Y are typical microcomputer index registers, as described in *An Introduction to Microcomputers: Volume 1*.² The X register is preferred over the Y register only because a few operation codes (such as CMP, LD, and ST) execute more slowly when applied to Y than when applied to X.

Stack Pointer U is a cross between the typical microcomputer index register and the typical microcomputer stack pointer as described in *An Introduction to Microcomputers: Volume 1*. Registers may be pushed onto or pulled from the User Stack (indexed by the User Stack Pointer). However, the processor does not employ the User Stack to store subroutine return addresses or the status of interrupted tasks; the processor uses only the Hardware Stack for those purposes.

The 6809 has a Stack implemented in memory and indexed by the Hardware Stack Pointer S as described in Volume 1 of *An Introduction to Microcomputers*. The instruction set allows S, as well as U, to be used as a data counter or index register.

Memory reference instructions make it easy to store the contents of either stack pointer or either index register in read/write memory. By assigning some memory locations on the base (direct) page as storage for these four address registers, you can put them all to multiple use. Another easy storage method is pushing the registers onto a stack.

The program counter is a typical program counter, as described in Volume 1 of *An Introduction to Microcomputers*.

The Direct Page Register DP generalizes the concept of a base page as described in *An Introduction to Microcomputers: Volume 1*. This register provides the eight most significant bits of a 16-bit address in the direct (base page) addressing mode. In most microprocessors (including the 6800), the base page is always page zero. The 6809 maintains compatibility with this concept by clearing the Direct Page register on hardware Reset. The Direct Page register allows the programmer to move the base page anywhere in memory and thus take advantage of short paged addresses without being limited to the first 256 bytes of memory. Different programs can have different base pages, thus both making it unnecessary to apportion page zero and reducing the chance of interference.

6809 FLAGS

The Carry flag holds the carry from the most significant bit produced by arithmetic operations or shifts. Like most microprocessors, the 6809 inverts the actual carry after subtraction so that the Carry flag also acts as a Borrow. The 6809 multiplication instruction, MUL, affects the Carry flag in yet another way: Carry represents bit 7 of the 16-bit result. This makes rounding to an 8-bit result very simple.

The Zero flag is standard. It is set to 1 when any operation produces a zero result. It is set to 0 when any operation produces a non-zero result.

The Sign (Negative) flag is standard. It takes on the value of the most significant bit of any result. Thus, a Sign flag value of 1 identifies a negative result and a Sign flag value of 0 identifies a positive result if the standard twos complement notation is being used. The Sign flag will be set or reset on the assumption that you are using signed binary arithmetic. If you are not using signed binary arithmetic, you can ignore the Sign flag or you can use it to identify the value of the most significant bit of the result.

The Half-Carry flag holds any carry from bit 3 to 4 resulting from the execution of an 8-bit addition instruction (ADC or ADD). The purpose of this flag is to simplify Binary-Coded-Decimal (BCD) operations. This is the standard use of a Half-Carry flag as described in *An Introduction to Microcomputers: Volume 1*, Chapter 4 (the flag is referred to there as an "intermediate carry").

The Overflow flag represents standard arithmetic overflow as described in Volume 1 of *An Introduction to Microcomputers*; that is, the flag is set when an arithmetic result is greater in magnitude than can be represented in the register. A processor implements this function by setting the overflow flag when the carry out of the most significant bit is different from the carry out of the next most significant bit; that is, an overflow is the exclusive-OR of the carries into and out of the sign bit. In the 6809, logical operations clear the Overflow flag, as do loads and stores.

The I and F flags are standard interrupt disable or interrupt mask flags. When I or F is 1, interrupts are disabled from the corresponding source (IRQ or FIRQ). When I or F is 0, the corresponding interrupt is enabled.

The E (or Entire) flag differentiates between regular interrupts and fast interrupts. E is set to 1 when any interrupt occurs that stacks the entire set of registers; E is set to 0 when an FIRQ occurs, stacking only the program counter and Condition Code register. The E flag thus allows proper unstacking of the registers by the RTI (return from interrupt) instruction.

The flags do not change until the processor executes an instruction that modifies them. Logical instructions, for example, do not affect the Carry or Half-Carry, but they do affect the Sign, Zero, and Overflow flags. Any of the flags can be specifically set or cleared by means of an ORCC or ANDCC instruction with the appropriate mask. You must use the bit positions shown earlier to create the mask; executing an ORCC with a 1 in a particular bit position will set a flag, while executing an ANDCC with a 0 in a particular bit position will clear a flag.

6809 literature refers to the Sign flag as a Negative flag and uses the symbol N for it. We will follow this convention to be compatible with the literature and to avoid confusion with the Hardware Stack Pointer (or S register). We will also follow the 6809 literature in referring to the Overflow flag by the symbol V (O leads to continual confusion) and the Half-Carry flag (sometimes called an Auxiliary or Intermediate Carry) by the symbol H. The 6809's flags are set and reset as described for the hypothetical microcomputer in *An Introduction to Microcomputers: Volume 1*.

6809 ADDRESSING MODES

Assembly language instructions tell the processor what operation to perform and what addresses to use in performing the operation — that is, where to find the data to be operated upon. The part of an instruction that tells the processor what operation to perform is the “operation code.” Appendix C lists the 6809 microprocessor’s mnemonic operation codes and their numerical equivalents. The part of an instruction that tells the processor what addresses to use is the “operand” or “address field.” The processor may use this part of an instruction to determine where to obtain the operands or where to store the result.

GENERAL DESCRIPTION OF ADDRESSING MODES

There are many different ways to specify what addresses the processor is to use. These ways are called “addressing modes.” We will describe them generally before discussing how the 6809 processor implements them. The following two modes do not involve memory at all:

1. **Inherent addressing** means that the operation code alone tells the processor what to do. Typical inherent addressing instructions are Halt, No Operation, and instructions that use specific registers.
2. **Register addressing** means that only registers are involved in the operation. Typical of such operations are moving data from one register to another and exchanging registers.

Common addressing modes that involve memory are as follows:

3. **Immediate addressing** means that the operand is located immediately after the operation code in program memory.
4. **Direct addressing** means that the address to be used follows the operation code in program memory.
5. **Indexed addressing** means that the address to be used is the sum of a base address and an index or offset.
6. **Indirect addressing** means that the address to be used is either in a register or in memory. That is, the instruction tells the processor where the address is, not where the data is.
7. **Relative addressing** means that the operand is located a certain distance from the current position in the program.

Chapter 6 of Volume 1 of *An Introduction to Microcomputers* describes all these addressing modes plus their common combinations.

6809 Addressing Modes

The 6809 microprocessor has a powerful and versatile set of addressing modes. The available modes are the following, listed in the order in which we will describe them:

1. Inherent operand (instructions that require no addresses)
2. Registers as operands (instructions that use only register contents as operands)

The other modes specify memory addresses; they are:

3. Immediate
4. Base page direct
5. Extended direct
6. Extended indirect
7. Constant offset from base register
8. Indirect with constant offset from base register
9. Accumulator offset from base register
10. Indirect with accumulator offset from base register
11. Autoincrement or autodecrement
12. Indirect with autoincrement or autodecrement
13. Program relative for branches

EFFECTIVE ADDRESS

In describing how the processor executes these modes and how the programmer uses them, we must often refer to the actual address that the processor ultimately uses to perform the specified operation. We call that address the “effective address”: it is the place from which the processor obtains an operand or in which the processor stores the result. In some modes (for example, immediate) the effective address is simply the location immediately following the operation code. In other modes, determining the effective address may be complicated. The address may be part of the instruction, the contents of a base register, or the contents of a pair of memory locations. Determining the effective address may involve computations, such as adding an offset to a base register. Some of the addressing modes are difficult to understand, since they involve sequences of operations that finally culminate in an effective address. We will explain why these sequences are useful and we will describe typical cases from real applications. You should try to trace each sequence, since the various addressing modes are the keys to writing programs that are both general and powerful. Remember, the processor always determines the effective address correctly, no matter how complex the required operations are.

In the following discussion, we will describe each addressing mode, explain at least one of its common uses, present a diagram of how it is executed, and discuss a specific example. All of these together should give you a picture of the power of the 6809 microprocessor.

MODES WHICH DO NOT SPECIFY MEMORY LOCATIONS

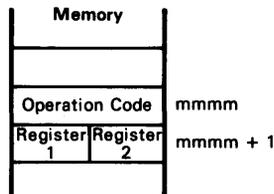
INHERENT ADDRESSING

In this mode, the processor knows from the operation code alone which addresses to use. For example, the instruction ABX (Add Accumulator B to Index Register X) tells the processor where to get both operands for the addition. Similarly, the instructions DAA (Decimal Adjust Accumulator A), MUL (Multiply), and SEX (Sign Extend) also tell the processor which registers to use. NOP (No Operation) and SYNC (Synchronize to External Event) require no operands, whereas RTI (Return from Interrupt), RTS (Return from Subroutine), and SWI (Software Interrupt) all force the processor to use the Hardware Stack Pointer to move data to or from memory. In all these instructions, the operation codes are complete by themselves; no further addressing information is necessary.

REGISTER ADDRESSING

Single-operand instructions can be applied to either Accumulator A or Accumulator B; the accumulator to be used is specified in the operation mnemonic. Typical examples are CLRB (Clear Accumulator B) and INCA (Increment Accumulator A). One bit in the actual operation code selects the accumulator. The following instructions fall in this category: ASL or LSL (Logical Shift Left), ASR (Arithmetic Shift Right), CLR (Clear: Set to Zero), COM (Ones Complement), DEC (Decrement: Subtract 1), INC (Increment: Add 1), LSR (Logical Shift Right), NEG (Negate: Twos Complement), ROL (Rotate Left), ROR (Rotate Right), and TST (Test for Zero or Minus).

The instructions TFR (Transfer Registers) and EXG (Exchange Registers) must have two registers of the same size as operands. For example, EXG X,U causes the processor to exchange the contents of Index Register X and the User Stack Pointer. The byte following the operation code designates (in coded form) which registers EXG or TFR is to use. We can illustrate these instructions as follows:



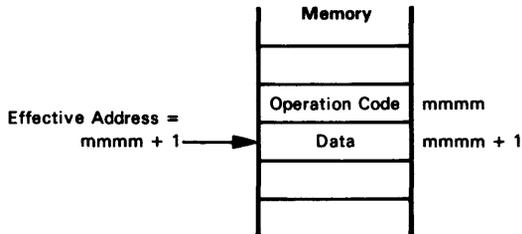
For details on how the registers are coded, see the descriptions of EXG and TFR in Chapter 22.

The instructions PSH (Store Data in Stack) and PUL (Load Data from Stack) also require a second byte that designates which registers are to be stored or loaded. These instructions, however, may load or store any number of user registers. Each bit of the second byte represents a register; if the bit is set, the processor will store the corresponding register in the stack or load it from the stack. For details on how the register addressing byte is organized, see the descriptions of the PSH and PUL instructions in Chapters 11 and 22.

MEMORY ADDRESSING MODES

IMMEDIATE ADDRESSING

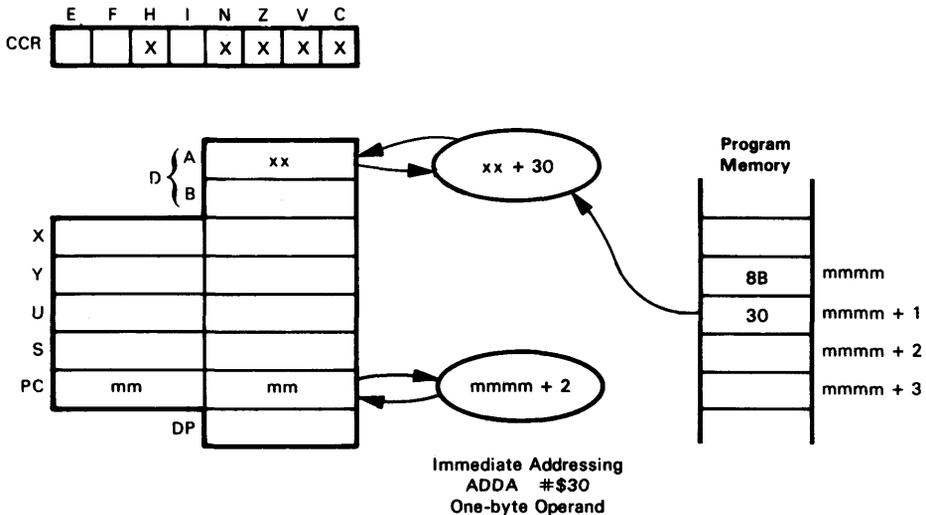
In immediate addressing, the data follows immediately after the operation code. That is, the effective address is simply the contents of the program counter after the processor has fetched the operation code. We can illustrate this mode as follows:



In standard 6809 assembly language, we specify immediate addressing by preceding the operand with the # symbol. Instructions may require either 8-bit or 16-bit immediate operands; 16-bit operands are stored with the most significant bits in the first byte. For example, the 6809 assembler converts the statement

ADDA #\$30

(# means “immediate addressing” and \$ means “hexadecimal”) into an ADD instruction that adds the value 30_{16} to Accumulator A. The following diagram illustrates the execution of the instruction.



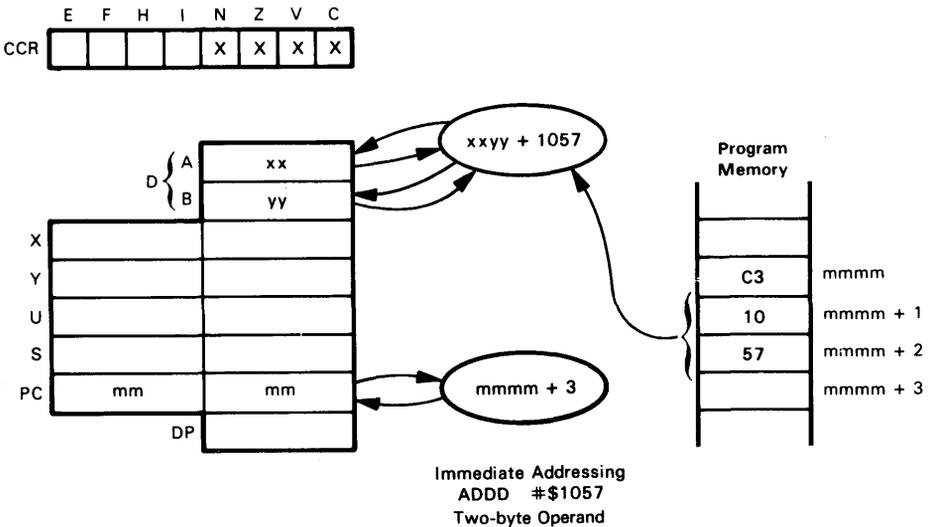
As a specific example, assume that Accumulator A contains $B7_{16}$ initially. After the processor executes $ADDA\ \$30$, the contents of Accumulator A will be $B7_{16} + 30_{16} = E7_{16}$. The processor increments its program counter twice, once after fetching the operation code and once after fetching the immediate data, 30_{16} in this example.

16-Bit Operations

Instructions that handle 16 bits at a time require a double-byte immediate operand. For example, the instruction

```
ADDD #$1057
```

causes the processor to add the 16-bit value 1057_{16} to the Double Accumulator D. Remember, D consists of Accumulators A and B with A holding the high-order byte. The following diagram shows how the processor executes the instruction.



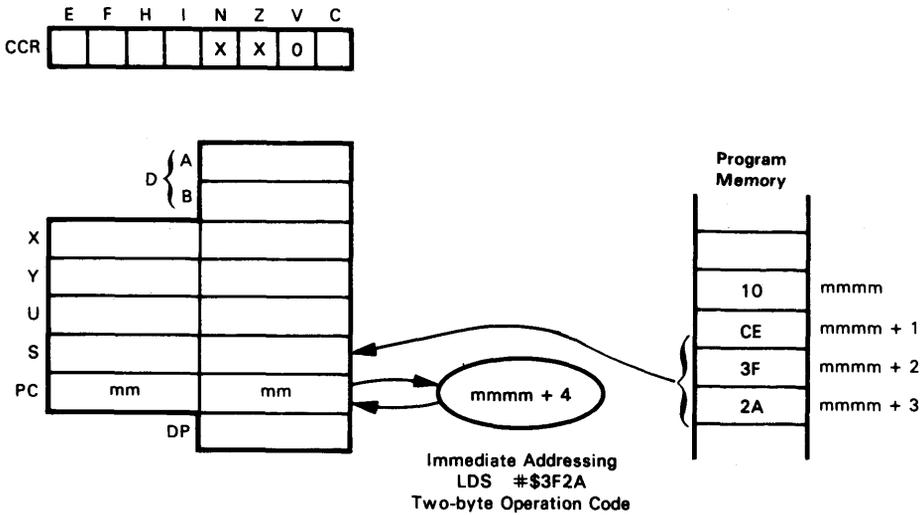
As a specific example, assume that the initial contents of the Double Accumulator are $3A48_{16}$. After the processor executes the instruction `ADDD #$1057`, the contents of the Double Accumulator will be $3A48_{16} + 1057_{16} = 4A9F_{16}$. The processor increments its program counter three times while executing the instruction, once after fetching the operation code and once after fetching each byte of the immediate operand.

Two-Byte Operation Codes

Some instructions require a two-byte operation code. Typical examples are `CMPD`, `CMPY`, `LDS`, and `LDY`. Since these instructions also require a 16-bit immediate operand, the immediate versions are four bytes long. For example, the instruction

```
LDS #$3F2A
```

has a two-byte operation code (10 CE), followed by a 16-bit (two-byte) immediate operand. We can illustrate the execution of this instruction as follows.



Instructions that Lack an Immediate Mode

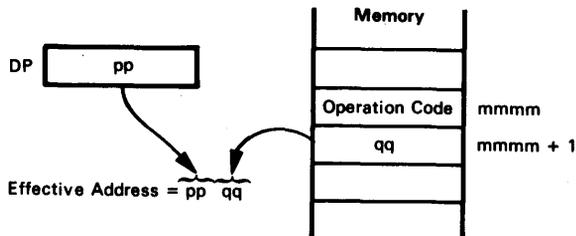
Some instructions do not make sense with immediate addressing.

1. You cannot store the contents of a register in a number, so **Store instructions** cannot use immediate addressing.
2. You cannot transfer control to a number, so **Jump and Jump-to-Subroutine instructions** cannot use immediate addressing.
3. You cannot clear or shift a specific number, so **single-operand instructions** cannot use immediate addressing.

You should refer to Appendix C or to your instruction set summary card if you are not sure whether an instruction allows immediate addressing.

BASE PAGE DIRECT ADDRESSING

In this mode, the effective address is on the base or direct page as defined by the contents of the direct page register. The low-order half of the address (that is, the address on the direct page) follows the operation code in memory. We can illustrate base page direct addressing as follows:



You should note that 6809 manufacturers usually refer to this mode as “direct,” whereas Volume 1 of *An Introduction to Microcomputers* refers to it as “base page.”

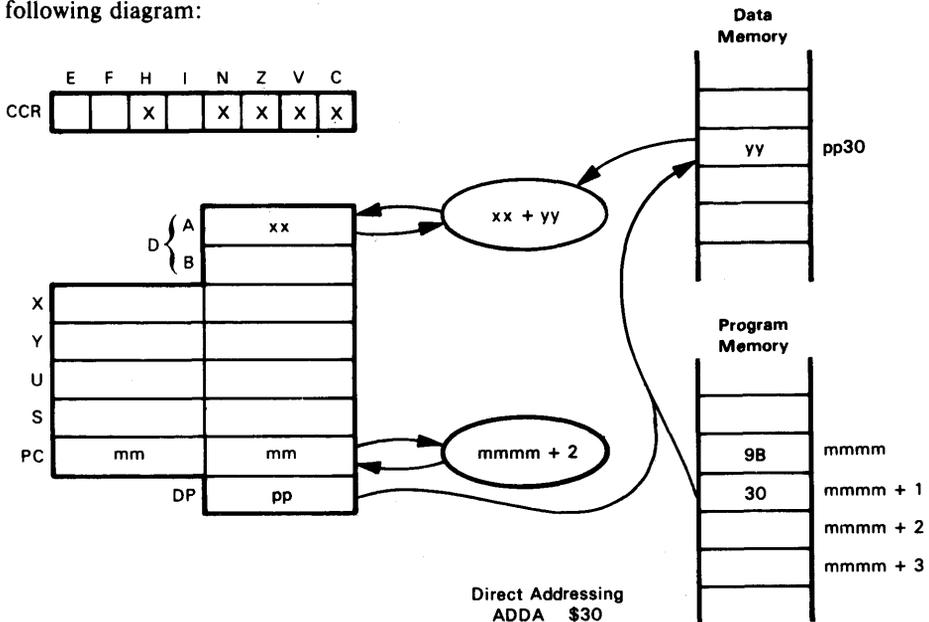
This mode provides a short, quick way to use temporary storage on the direct page. It is short and quick because the page number is in the direct page register on the processor chip, thus saving a byte of program memory and a read cycle. Obviously, there is an overall savings of time and memory only if the programmer rarely changes the contents of the direct page register. Otherwise the instructions that load the direct page register more than offset the savings from using it.

The standard 6809 assembler uses direct addressing whenever the mode is available, no other mode is specified, and the address is on the direct page. The assembler assumes that the direct page is page zero (thus maintaining compatibility with the earlier 6800 microprocessor, which has no direct page register) unless told otherwise; the programmer may specify a different direct page with a SETDP assembler directive. The programmer may also force the assembler to use direct addressing by preceding an address with the symbol "<", but this is rarely necessary.

For example, the assembler converts the statement

```
ADDA #S30
```

into an ADD instruction that adds the contents of memory location $pp30_{16}$ to Accumulator A, where pp is the contents of the Direct Page register as shown in the following diagram:

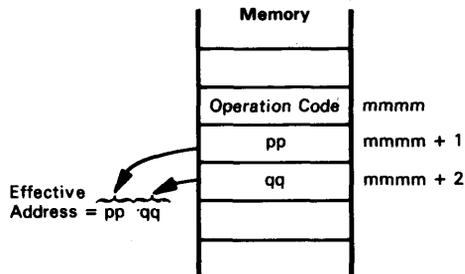


As a specific example, let us assume the initial contents of Accumulator A are 47_{16} , the contents of the Direct Page register are $2B_{16}$, and the contents of memory address $2B30_{16}$ are $6A_{16}$. After the processor executes the instruction, the sum in Accumulator A will be $47_{16} + (pp30_{16}) = 47_{16} + (2B30_{16}) = 47_{16} + 6A_{16} = B1_{16}$. The processor increments its program counter twice, once after fetching the operation code and once after fetching the direct address.

The direct address occupies only one byte even if the instruction (such as ADD, LDS, or STX) handles 16-bit operands. In that case, the processor uses the addresses $ppqq$ and $ppqq + 1$ to fetch or store the high-order and low-order bytes of the data, respectively. Instructions such as LDY and STS require a two-byte operation code, in which case the base page direct form occupies three bytes of program memory.

EXTENDED DIRECT ADDRESSING

In this mode, the effective address occupies the two bytes of program memory immediately following the operation code. The high-order half of the address is in the first byte; this is standard 6809 format. We can illustrate extended direct addressing as follows:



You should note that 6809 manufacturers usually refer to this mode as “extended,” whereas Volume 1 of *An Introduction to Microcomputers* refers to it as “direct” or “extended direct.”

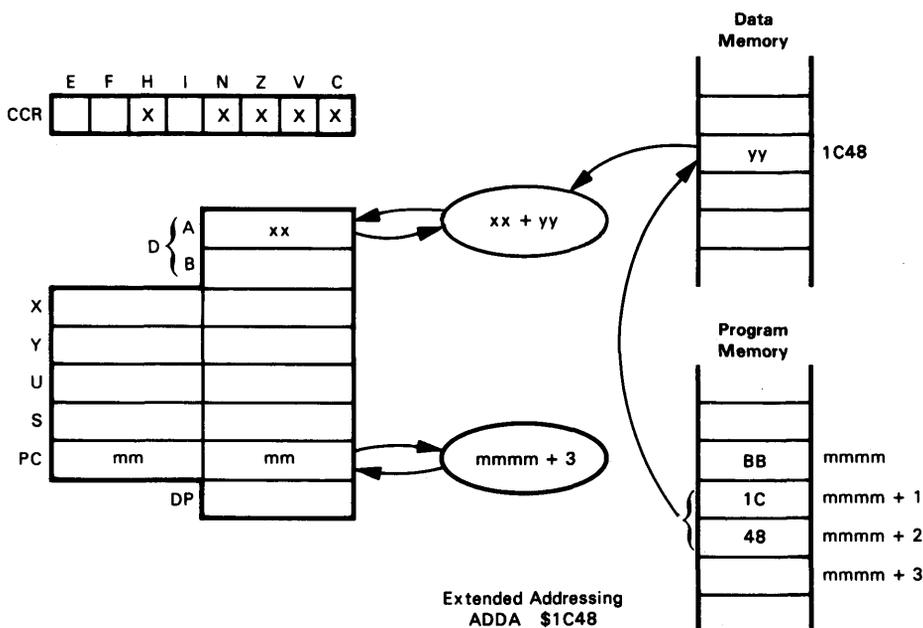
This mode allows the processor to access any specific memory location. Of course, you need not use extended addressing for memory locations that are on the direct page, since the base page direct mode is shorter and faster. However, **extended addressing is the usual approach for handling a fixed address that is not on the direct page.** This mode is often used in performing input and output, since the memory addresses assigned to I/O devices are rarely on the direct page.

The standard 6809 assembler uses extended addressing whenever the mode is available, no other mode is specified, and the address is not on the direct page. Thus extended addressing is the general default mode. The programmer may force the assembler to use extended addressing by preceding the address with the symbol “>”, but this is rarely needed.

For example, the assembler converts the statement

```
ADDA $1C48
```

into an ADD instruction that adds the contents of memory location $1C48_{16}$ to Accumulator A as shown in the following diagram.

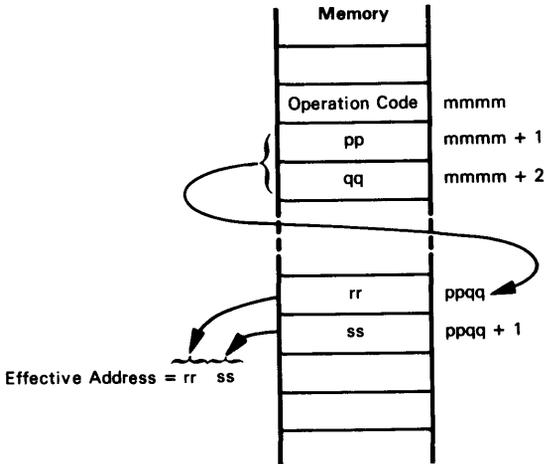


As a specific example, assume the initial contents of Accumulator A are $F4_{16}$ and the contents of memory address $1C48_{16}$ are $3A_{16}$. After the processor executes the instruction `ADDA $1C48`, the sum in Accumulator A will be $F4_{16} + 3A_{16} = 2E_{16}$. The processor increments its program counter three times, once after fetching the operation code and once after fetching each byte of the direct address.

If the instruction (for instance, `CMPX` or `SUBD`) handles 16-bit operands, the addresses used are $ppqq$ and $ppqq + 1$. If the instruction (for example, `CMPY`) requires a two-byte operation code, the extended direct form requires four bytes of program memory.

EXTENDED INDIRECT ADDRESSING

In this mode, the effective address is located at the address in the two bytes of program memory immediately following the operation code. That is, the instruction tells the processor where to find the address, not what its value is. You may compare indirect addressing to a treasure hunt in which one clue tells you where to look for the next clue, not where to find the actual treasure. If, as shown in the next illustration, the two bytes following the operation code contain pp (first byte) and qq (second byte), the effective address is located in addresses $ppqq$ (first byte) and $ppqq + 1$ (second byte). So the effective address in the illustration is $rrss$.



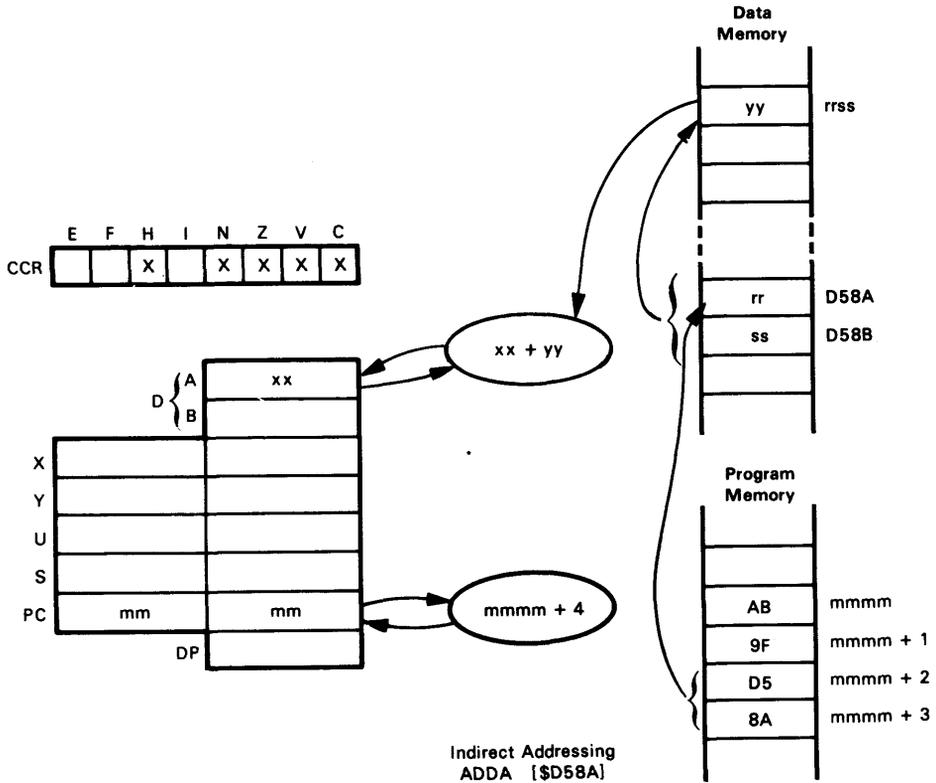
The important point here is to see what this added complication does for us, besides provide some confusion. **Indirection allows a program to use different effective addresses without being changed, since all the program contains is the location of the effective address, not its value.** Why is that useful? Assume, for example, that our application involves printing some results, as most applications do. We write a single routine that takes the results from memory and sends them to an output device. (The 6809 uses memory-mapped input/output, so an output device is addressed using memory addresses.) However, sometimes those results must be sent to a printer (for permanent records), while at other times the results are merely displayed (to the operator) or reported via a remote line (to a central computer). If our routine sends the results using extended indirect addressing, it can send them to any output device. All the main program must do is place the address of the output device in the specified memory locations. The approach is the same as that of television commercials which tell you to call the telephone number that will appear on your screen. The same commercial can be used nationwide; all the local station does is display the correct local number.

In the standard 6809 assembler format, you specify extended indirect addressing by placing the address in square brackets: for example, [D58A]. The address is always interpreted as a 16-bit value and always occupies two bytes of program memory. Instructions that use extended indirect addressing require a post byte after the operation code, and this post byte must contain $9F_{16}$. We will discuss post bytes in more detail as part of the description of the indexed addressing modes.

For example, the assembler converts the statement

```
ADDA [D58A]
```

into an ADD instruction that adds the contents of memory location $rrss$ to Accumulator A, where rr is the contents of address $D58A_{16}$ and ss is the contents of address $D58B_{16}$. The following diagram illustrates the execution of the instruction.



As a specific example, let us assume the initial contents of Accumulator A are $1F_{16}$ and the contents of memory addresses $D58A_{16}$, $D58B_{16}$, and $06E4_{16}$ are 06_{16} , $E4_{16}$, and 35_{16} respectively. After the processor executes the instruction `ADDA [$D58A]`, the sum in Accumulator A will be $1F_{16} + ((D58A_{16}): (D58B_{16})) = 1F_{16} + (06E4_{16}) = 1F_{16} + 35_{16} = 54_{16}$. The processor increments its program counter four times, once after fetching the operation code, once after fetching the post byte, and once after fetching each byte of the indirect address. Clearly this instruction takes extra time to execute (see Appendix B), since the processor must go through a complex sequence to obtain the actual data.

INDEXED MEMORY ADDRESSING MODES

In all the indexed addressing modes, the processor uses a base register. This register may be one of the two index registers, one of the two stack pointers, or the program counter. The instruction tells the processor which base register to use, whether to add an offset to the contents of the base register, where to obtain the offset if one is necessary, whether to change the contents of the base register, and whether to use the indexed address directly or indirectly. Volume 1 of *An Introduction to Microcomputers* describes the use of base registers in detail; their use allows program-

mers to handle data structures that are defined by a base address and to write position-independent code in which the location of the program itself is defined only by a base address.

OBJECT CODE POST BYTE

The 6809's indexed and indirect addressing modes require that the operation code be followed by a byte that differentiates among the various modes. We refer to this extra byte as a "post byte." Figure 3-1 shows the placement of the post byte in the object code. Table 3-4 describes the meanings of the bit positions within the post byte. If you wish more details, Appendix B contains a summary of the indexed modes and Appendix E describes the meanings of all possible post bytes in numerical order.

Information in the Post Byte

Let us summarize the information contained in the post byte:

1. **Which base register to use:** Index Register X, Index Register Y, Stack Pointer U, Stack Pointer S, or the program counter. Of course, **extended indirect addressing uses no base register** at all.
2. **Whether to add an offset** to the base register.
3. **Where to find the offset** if it is necessary. The choices here are: within the post byte itself, in the next one or two bytes of program memory, in Accumulator A, in Accumulator B, or in Double Accumulator D.
4. **Whether to change the base register's contents.** The choices here are to add 1 or 2 to the base register after using it (sometimes called "postincrement") or to subtract 1 or 2 from the base register before using it (sometimes called "predecrement").
5. **Whether to use the address obtained so far directly or indirectly.** That is, whether to use the address to obtain the data or the address of the data. Using an indexed address indirectly is often referred to as "preindexing" or "indirect indexed addressing."

Unimplemented and Illegal Indexed Modes

Not all combinations are implemented. For example, there is no mode that both changes the base register and adds an offset. Nor are there modes that use the program counter as a base register and also change the base register or obtain the offset from within the post byte or from an accumulator. Furthermore, adding 1 to a base register that is used indirectly or subtracting 1 from it is illegal. This is because the base register must point to a 2-byte address, and adding 1 to it or subtracting 1 from it would therefore cause it to point to the middle of an address. We will describe the valid forms and their uses as we proceed.

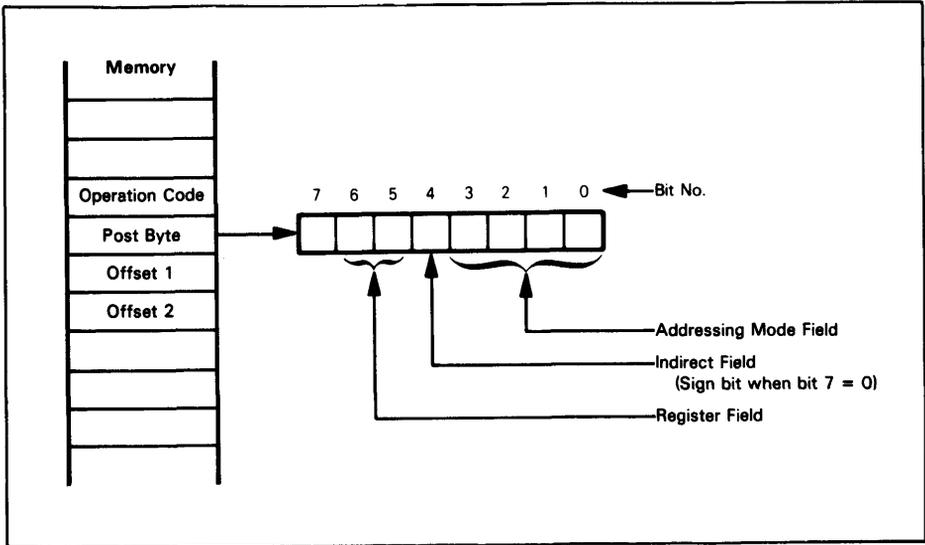


Figure 3-1. 6809 Post Byte for Indexed and Indirect Addressing

Table 3-4. 6809 Post Byte Bit Assignments for Indexed and Indirect Addressing

Bit Number								Addressing Mode
7	6	5	4	3	2	1	0	
0	R	R	X	X	X	X	X	5-Bit Offset
1	R	R	R	0	0	0	0	Autoincrement by One
1	R	R	R	1	0	0	0	Autoincrement by Two
1	R	R	R	0	0	0	1	Autodecrement by One
1	R	R	R	1	0	0	1	Auto Decrement by Two
1	R	R	R	1	0	1	0	Zero Offset
1	R	R	R	1	0	1	0	Accumulator B Offset
1	R	R	R	1	0	1	1	Accumulator A Offset
1	R	R	R	1	1	0	0	8-Bit Offset
1	R	R	R	1	1	0	0	16-Bit Offset
1	R	R	R	1	1	0	1	Accumulator D Offset
1	X	X	1	1	1	0	0	Program Counter 8-Bit Offset
1	X	X	1	1	1	0	1	Program Counter 16-Bit Offset
1	X	X	1	1	1	1	1	Extended Indirect

Addressing Mode Field		
Indirect Field I = 1 for indirect, I = 0 for direct (Sign bit when bit 7 = 0)		
Register Field		
	00	R = X
	01	R = Y
	10	R = U
	11	R = S

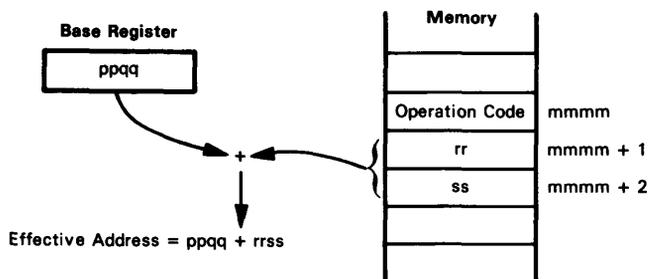
NOTATION FOR INDEXED ADDRESSING MODES

The standard 6809 assembler uses the following special notation in referring to indexed addressing modes:

- **.R** means that the 16-bit register R (X, Y, U, S, or PC) is to be used as the base register.
- **OFFSET,R** means that the number OFFSET is to be added to the contents of base register R. A zero offset can be omitted unless the base register is the program counter.
- **LABEL,PCR** means that the program counter is to be used as the base register and the offset is to be the distance from the location of the instruction to the address LABEL. That is, the address LABEL is specified "program counter relative."
- **,R+** means that the 16-bit base register R (X, Y, U, or S) is to be incremented (once for one plus sign, twice for two plus signs) after its contents are used in determining the effective address.
- **,-R** means that the 16-bit base register R (X, Y, U, or S) is to be decremented (once for one minus sign, twice for two minus signs) before its contents are used in determining the effective address.
- **Square brackets []** around an indexed address indicate that it is to be used indirectly.

CONSTANT OFFSET FROM BASE REGISTER

In this mode, the effective address is the sum of a fixed offset and the contents of a base register. The base register can be any of the following: Index Register X, Index Register Y, Stack Pointer U, Stack Pointer S, or the program counter. Since the purpose of the method is different when the base register is the program counter, we will discuss that option separately. The procedure for obtaining the effective address, however, is always as shown in this diagram:



The offset follows the operation code, which includes the post byte, in program memory. It is a constant since program memory generally does not change during program execution. The contents of the base register may vary; the program can determine the values in the index registers and stack pointers, whereas the program counter contents depend on the placement of the program.

When used with an index register or stack pointer, this addressing mode allows us to refer to a particular element in an array or list. For example, we may have a set of ten temperature readings taken at different points in a tank; to change or display a particular one, we must know where the set of readings starts (base address) and which reading we want (index or offset). If, as is usual, we store the readings in successive memory locations, we can find one by using a constant offset from the base.

Similarly, we may store a record in memory consisting of a person's name, address, identification number, age, and job classification. If we want to send notices of change of wage rate to all people in a particular job classification, we can find the job classification by specifying how far it is from the start of the record (for example, 97 bytes further). This is much like telling all the students who are taking an examination to put their names on the top line, their class levels on the fifth line, and their dates of birth on the tenth line. Each student locates the required lines relative to the top of his or her form. So, in our records, the name might occupy the first 16 bytes, the address the next 70, the identification number the next 9, and the age the next 2. **We can locate a particular field in a particular record (for example, employee #4's identification number) by specifying the base address (in this case, where employee #4's record starts) and how far beyond that we must go for the desired field (in our example, 86 bytes to the identification number).**

Short Constant Offset Modes

Although we have described situations in which the offset could be large, offsets are usually small. We are more likely to want something that is a few locations away than something that is thousands of locations away. **So the 6809 microprocessor provides special short modes to handle the cases where:**

1. **The offset is zero.** We want to use the base register as an implied memory address, as described extensively in *An Introduction to Microcomputers: Volume 1*.
2. **The offset is small enough to fit in the post byte.** Since we need one bit to indicate whether this case holds, and two bits to designate which index register or stack pointer is the base register, the offset must fit in five bits. The 6809 microprocessor interprets these five bits (the least significant bits of the post byte) as a sign (bit 4) and a 4-bit twos complement number (bits 0 through 3). Thus the range is -16_{10} (10000_2) \leq offset $\leq +15_{10}$ (01111_2).

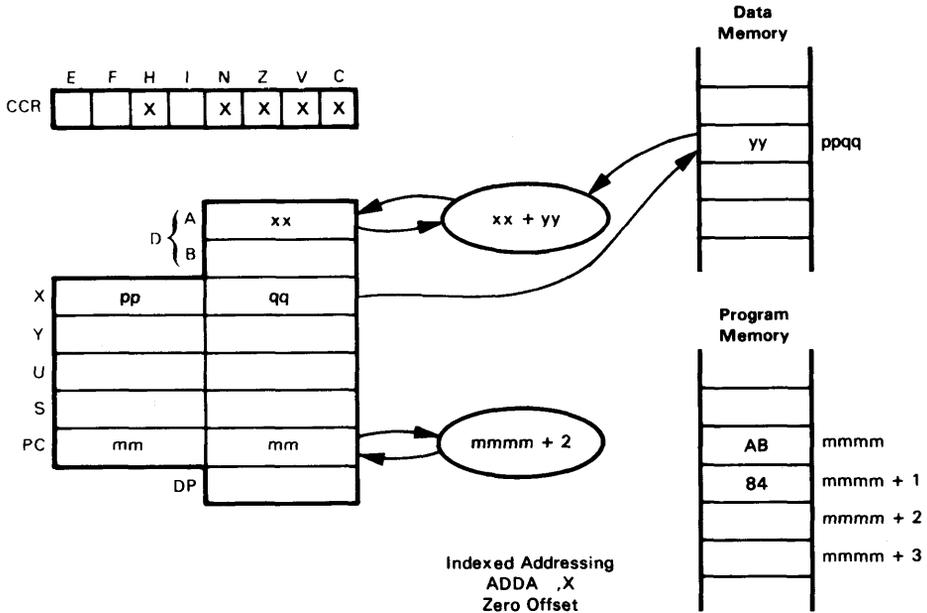
The advantages of these short modes are obvious: they save time and memory, since no additional bytes are needed for the offset. Furthermore, if the offset is zero, the processor does not have to go through the motions of adding it to the base.

Zero Offset (No Offset)

As an example of the zero offset mode, the assembler converts the statement

```
ADDA ,X
```

into an ADD instruction that adds the contents of the address in Index Register X to Accumulator A. The following diagram illustrates the execution of the instruction.



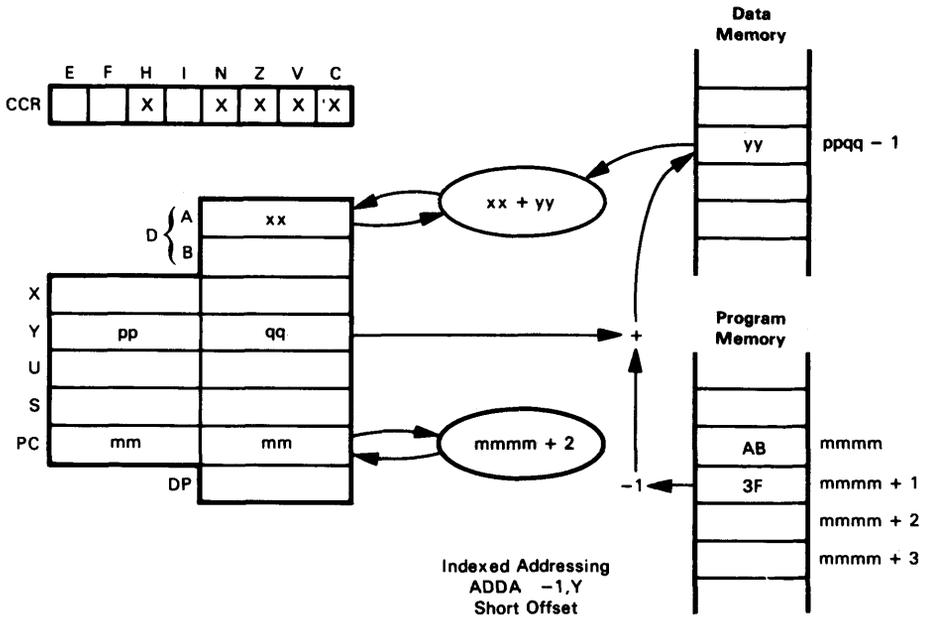
The effective address here is simply the contents of Index Register X. If, for example, Accumulator A contains $B7_{16}$, Index Register X contains $01E1_{16}$, and memory address $01E1_{16}$ contains 15_{16} , then after the processor executes `ADDA ,X` Accumulator A will contain $B7_{16} + ((X)) = B7_{16} + (01E1_{16}) = B7_{16} + 15_{16} = CC_{16}$. The processor increments its program counter twice, once after fetching the operation code and once after fetching the post byte.

Five-Bit Offset

As an example of the short offset mode, the assembler converts the statement

```
ADDA -1,Y
```

into an ADD instruction that adds the contents of the address one less than that specified by Index Register Y to Accumulator A. That is, the effective address is the contents of Index Register Y minus 1. The following diagram illustrates the execution of the instruction.



As a specific example, assume that Accumulator A contains 94_{16} , Index Register Y contains $A048_{16}$, and memory address $A047_{16}$ contains 32_{16} . After the processor executes the instruction `ADD -1, Y`, Accumulator A will contain $94_{16} + ((Y) - 1) = 94_{16} + (A048_{16} - 1) = 94_{16} + (A047_{16}) = 94_{16} + 32_{16} = C6_{16}$.

This mode takes longer for the processor to execute than the zero offset does because of the address addition. That is, the processor must add the offset (-1 in this case) to the contents of the base register (Index Register Y). What if the offset is zero? The processor adds it in anyway, thus wasting some time. In 6809 assembler notation the difference is between

`ADD ,X`

which tells the processor to use the zero offset mode, and

`ADD 0, X`

which tells the processor to use the 5-bit offset mode with a value of zero. Obviously, you should always use the first notation instead of the second because the first executes faster: Both are legal, but the second has no advantage. **Motorola's 6809 assemblers automatically optimize to the zero offset mode; thus, a Motorola assembler would produce the same object code — AB 84 — for both `ADD ,X` and `ADD 0, X`.** Not all 6809 assemblers have this desirable feature, however; you will save yourself potential trouble by using the special zero offset notation exclusively.

Larger Constant Offset Modes

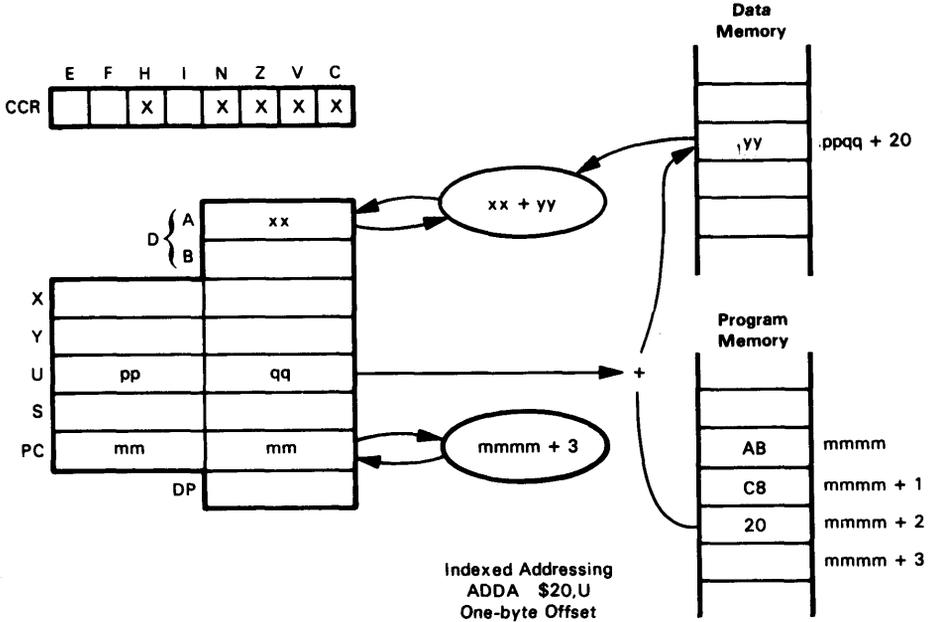
If the offset is not zero or small enough to fit in the post byte, one or two extra bytes of program memory beyond the post byte must be used to hold it. An 8-bit mode and a 16-bit mode allow offsets of any length: Of course, the frequency of use goes down as the length of the offset increases. Furthermore, the longer the offset, the

more extra time and memory the instruction requires (see Appendix B). So we use the longer modes as seldom as possible.

As an example of the 8-bit offset mode, the assembler converts the statement

```
ADDA $20,U
```

into an ADD instruction that adds the contents of the address 20_{16} beyond the address in Stack Pointer U to Accumulator A. The 8-bit offset (20_{16}) is located immediately after the post byte in program memory. The following diagram illustrates the execution of the instruction.



The effective address here is the contents of Stack Pointer U plus 20_{16} . The processor interprets bit 7 of the offset as a sign and the remaining seven bits as a two's complement number. Thus the range of the offset is $-128 = 1000\ 0000_2 \leq \text{offset} \leq +127 = 0111\ 1111_2$. As a specific example, assume that Accumulator A contains $4D_{16}$, Stack Pointer U contains $054E_{16}$, and memory address $056E_{16}$ contains $2A_{16}$. After the processor executes `ADDA $20,U` Accumulator A will contain $4D_{16} + ((U) + 20_{16}) = 4D_{16} + (054E_{16} + 20_{16}) = 4D_{16} + (056E_{16}) = 4D_{16} + 2A_{16} = 77_{16}$.

The extension of this mode to a 16-bit offset occupying two bytes is obvious; we will not discuss it further.

Constant Offset from the Program Counter

The modes that use a constant offset from the program counter help us write position-independent code: that is, programs that work regardless of where they are placed in memory. Such programs can be moved, without changes, to any available memory locations and used with any combination of other programs. The easiest way to make a program position-independent is for it to specify any addresses it uses relative to its own position. How does a program know its own position? By looking at the contents of the program counter. The idea here is the same as a repair manual that first orients

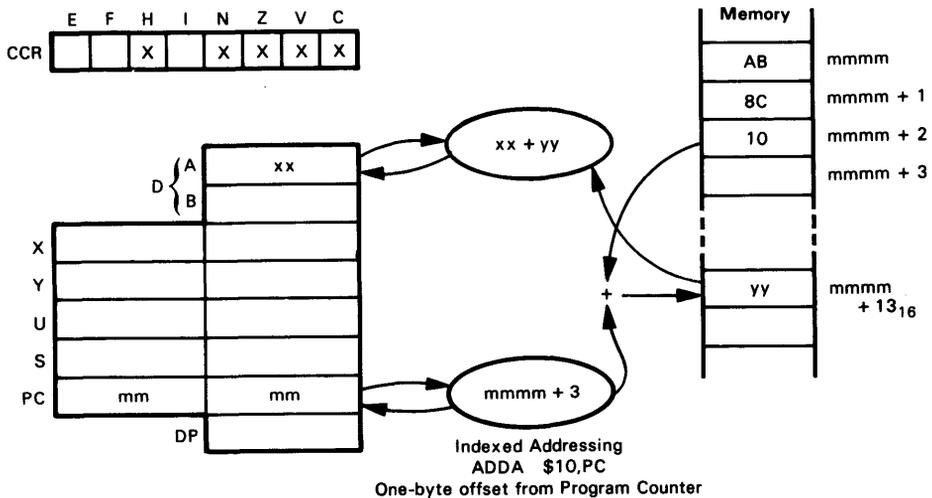
the user properly (for example, by telling the person to face the equipment as shown in a particular picture) and then refers to things as being “in back,” “in the top left-hand corner,” or “fourth from the left in the bottom row.” These descriptions are all relative to the user’s position.

We can move an entire program along with its data if we refer to addresses relative to the program counter. The idea here is to refer to data as being “20 locations from where we are,” rather than at a particular address. If we then move everything, the relative positions of instructions and data remain the same, even though their absolute addresses change. This is like telling someone that the dining car on a train is two cars ahead; the relative positions of the cars remain the same, even though the entire train is moving.

The 6809 microprocessor allows either an 8-bit or a 16-bit offset from the program counter. No special zero or 5-bit modes are provided, as with the index registers and stack pointers. In fact, offsets from the program counter are likely to be large, since data areas are usually separated from program areas. In the 8-bit offset, bit 7 is the sign and bits 0 through 6 are a two’s complement number. As an example of this mode, let us discuss the execution of the instruction

ADDA \$10,PC

which adds the contents of the address 13_{16} beyond the initial value of the program counter to Accumulator A. Why is the offset 13_{16} , not 10_{16} ? The reason is that the processor fetches the entire 3-byte instruction (operation code, post byte, and 8-bit offset) before calculating the effective address. Thus it has already added 3 to the program counter by the time it uses that register for addressing. In the 16-bit offset mode, the extra factor is 4 since the instruction occupies four bytes (operation code, post byte, and 16-bit offset). The following diagram illustrates the execution of the instruction ADDA \$10,PC.



The effective address here is the final contents of the program counter plus 10_{16} . As a specific example, assume that Accumulator A contains CA_{16} , the program counter contains $7B09_{16}$, and memory address $7B1C_{16}$ contains 05_{16} . After the processor executes ADDA \$10,PC Accumulator A will contain $CA_{16} + ((PC) + 3 + 10_{16}) = CA_{16} + (7B09_{16} + 3 + 10_{16}) = CA_{16} + (7B1C_{16}) = CA_{16} + 05_{16} = CF_{16}$. The diagram

and this example show clearly that **the result does not depend on where the instruction is located in program memory, as long as the instruction and the data are moved as a unit.** Here again, the extension to a 16-bit offset is obvious and we will not discuss it further.

Program Counter Relative (PCR) Notation

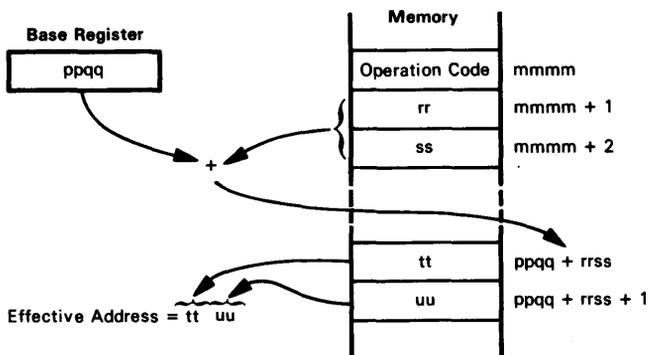
The extra three or four bytes involved in calculating an offset from the program counter are a nuisance, particularly if the offset is negative. Furthermore, if the operation code is two bytes long, the numbers become four or five since the instructions then occupy an extra byte of program memory. We would like to have the assembler handle this for us, since the procedure is simple to explain but difficult to perform correctly. **The standard 6809 assembler will calculate the program counter offset for you if you designate the address as “program counter relative,” or PCR.** For example, if you write

```
ADDA LOCUS,PCR
```

the assembler will figure out the distance from the instruction to address LOCUS (including the proper factor of three or four) and make that distance into an 8-bit or 16-bit offset. **This is the standard approach to writing position-independent 6809 code efficiently.**

INDIRECT WITH CONSTANT OFFSET FROM BASE REGISTER

We can add indirection to the constant-offset modes. The only change is that there is no special 5-bit mode with indirection — because the 5-bit offset occupies the bit used to differentiate between non-indirect and indirect modes — so we must use the 8-bit mode instead. The process of determining the effective address becomes complex here, since it involves an addition followed by two memory accesses. We can illustrate it as follows:



The indirection allows us to handle items in arrays, lists, or records which are addresses rather than data. For example, a microcomputer might be monitoring data from several remote stations. To each station, we assign a block of memory locations that contain:

1. Station number
2. Interval between readings

3. Address in which next reading will be stored
4. Minimum valid reading
5. Maximum valid reading
6. Starting address of routine that handles invalid readings
7. Number of readings taken since last report
8. Number of readings per report
9. Address of output device on which report is printed
10. Starting address of routine that processes readings for a report

Some items in the block are data, whereas others (#3, #6, #9, and #10) are addresses. The use of this block allows the operator to change any of the parameters without affecting the overall program. The operator can vary the time interval between readings (item #2), the data area used for temporary storage (item #3), the procedure for handling invalid readings (item #6), or the output device on which the occasional reports are printed (item #9). We can handle the data with non-indirect indexed addressing, whereas we must handle the addresses with indirect indexed addressing. For example, if the next reading is in Accumulator A and the address in which that reading is to be stored (item #3) occupies bytes 4 and 5 of the block, we can store the reading with the sequence

```
LDX  #BLOCK  GET STARTING ADDRESS OF BLOCK
STA  [4,X]   STORE READING IN MEMORY
```

If later we want to take that reading (its address is item #3) and send it to the output device (its address is item #9), we can use the sequence

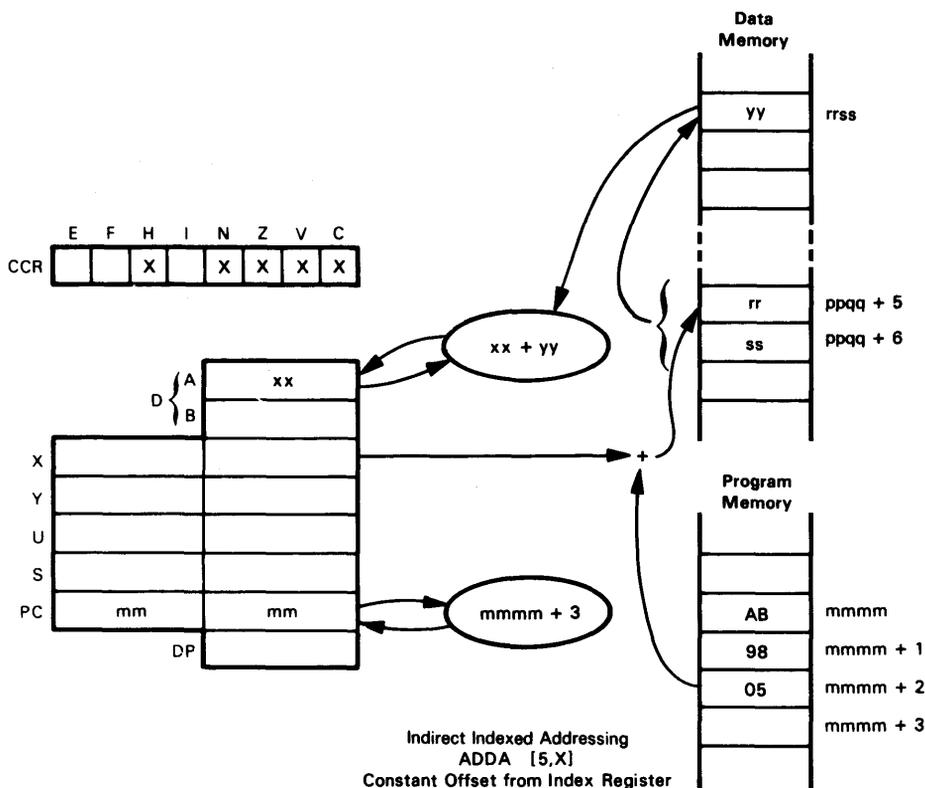
```
LDX  #BLOCK  GET STARTING ADDRESS OF BLOCK
LDA  [4,X]   GET MOST RECENT READING
STA  [OUT,X] REPORT MOST RECENT READING
```

Here OUT is the offset for item #9, the address of the output device.

As an example of the indirect indexed mode with constant offset, let us examine the execution of the instruction

```
ADDA  [5,X]
```

which adds to Accumulator A the contents of the address stored five bytes beyond the address in Index Register X. That is, Index Register X is the base; the instruction adds 5 (the offset) to the base and uses the sum as an indirect address. This mode obviously requires extra execution time (see Appendix B) because of the addition and the subsequent memory accesses. The following diagram illustrates the execution of the instruction.



As a specific example, let us assume that Accumulator A contains $1B_{16}$, Index Register X contains $0C33_{16}$, memory address $0C38_{16}$ contains $A0_{16}$, memory address $0C39_{16}$ contains $D2_{16}$, and memory address $A0D2_{16}$ contains 47_{16} . After the processor executes the instruction `ADDA [5,X]`, Accumulator A will contain $1B_{16} + (((X) + 5) : ((X) + 6)) = 1B_{16} + ((0C33_{16} + 5) : (0C33_{16} + 6)) = 1B_{16} + ((0C38_{16}) : (0C39_{16})) = 1B_{16} + (A0D2_{16}) = 1B_{16} + 47_{16} = 62_{16}$.

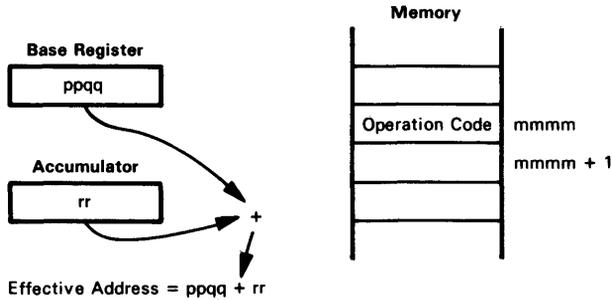
The other indirect modes with a constant offset are:

- Zero offset from an index register or stack pointer
- 16-bit offset from an index register or stack pointer
- 8-bit offset from the program counter
- 16-bit offset from the program counter

The processor executes all these similarly to the 8-bit offset mode described earlier. Note that **there is no special zero offset mode using the program counter**. We can use the PCR (program counter relative) notation to simplify the specification of relative addresses as we discussed previously.

ACCUMULATOR OFFSET FROM BASE REGISTER

This mode allows the offset, as well as the base register contents, to vary. The offset may be in either accumulator or in the double accumulator; the base register may be either index register or either stack pointer. Note, however, that the base register cannot be the program counter. As shown in the following illustration, the instruction does not contain any address at all.



A common use of this mode is to access lookup tables. Let us assume, for example, that we have a lookup table in memory that converts 8-bit ASCII character codes to 8-bit EBCDIC character codes. The table consists of EBCDIC codes, organized according to the ASCII codes to which they correspond. For instance, the 0th entry is the EBCDIC code corresponding to the ASCII code 0, the 15th entry is the EBCDIC code corresponding to the ASCII code 15 ($0F_{16}$), and the 43rd entry is the EBCDIC code corresponding to the ASCII code 43 ($2B_{16}$). To convert an ASCII code to EBCDIC, we need to know where the table starts (let's call it address EBCDIC) and the value of the ASCII code (let's assume it is stored temporarily in address CHAR). Then we can use the accumulator offset mode to fetch the EBCDIC code from the table. A typical program is:

```
LDX #EBCDIC GET BASE ADDRESS OF EBCDIC CODE TABLE
LDB CHAR GET ASCII CODE (ELEMENT NUMBER)
LDA B,X GET CORRESPONDING EBCDIC CODE FROM TABLE
```

For more details on character codes, see Chapter 6 of this book and Chapter 3 of Volume 1 of *An Introduction to Microcomputers*. For further discussions of lookup tables, see Chapters 4, 7, and 8 of this book.

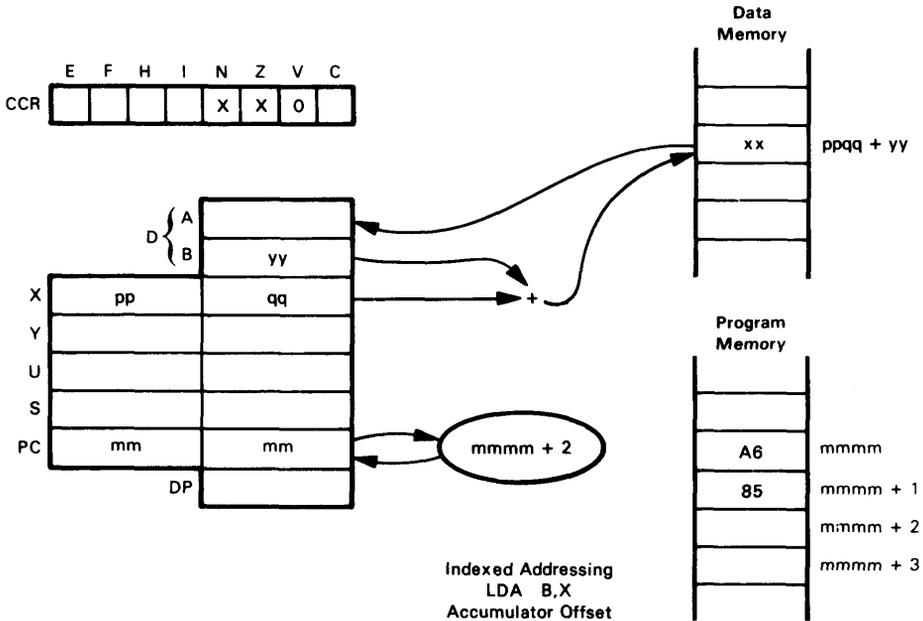
Note the difference between this mode and the constant offset modes. In this mode, the offset is a variable. In the example, the ASCII code could have any value; typically the microprocessor would be receiving a string of ASCII data from an input device and converting it into a string of EBCDIC data for an output device. In the constant offset modes, the offset does not change. In our example of an employee record, a person's identification number or job classification is always located a specific number of bytes from the start of the record.

As an example of the accumulator offset mode, let us consider the instruction

```
LDA B,X
```

which loads Accumulator A from the address obtained by adding Accumulator B and Index Register X. The mode using Accumulator A for the offset clearly works the same way; the double accumulator offset mode is similar except for the offset's length. Note, however, that the processor interprets the contents of a single accumulator as an 8-bit

signed twos complement number. Thus the accumulator offset mode has a slightly different effect than the ABX instruction, which interprets the contents of Accumulator B as an 8-bit unsigned number. Accumulator offset addressing requires extra time because the processor must add the offset and the base; it does not require any extra memory since the offset is in an accumulator or double accumulator. The double accumulator version is necessary when the table occupies more than 256 bytes. The following diagram illustrates the execution of the LDA B,X instruction.



As a specific example, let us assume Accumulator B contains $2B_{16}$ (the ASCII code for +), Index Register X contains $C300_{16}$ (starting address of an ASCII-to-EBCDIC conversion table), and memory address $C32B_{16}$ contains $4E_{16}$ (the EBCDIC code for +). Then after the processor executes LDA B,X Accumulator A will contain $((X) + (B)) = (C300_{16} + 2B_{16}) = (C32B_{16}) = 4E_{16}$. We have converted an ASCII code in Accumulator B into the corresponding EBCDIC code in Accumulator A. If you wish to test this approach on other characters, use the character code tables in Appendix A of *An Introduction to Microcomputers: Volume 1*.

INDIRECT WITH ACCUMULATOR OFFSET FROM BASE REGISTER

We can add indirection to the accumulator offset mode to handle the case in which the table contains addresses rather than data. For example, the table might contain the actual addresses corresponding to numbered input and output devices. The operator of the microcomputer-based system will ask the system to "read data from device #4" or "print results on device #6." The microcomputer will use the table to determine which addresses correspond to devices 4 and 6. This approach (see Chapter 12 for further discussion) allows the operator to change I/O devices by

changing the table. For example, the operator could let device #6 be a CRT display for a test run (thus showing test results without wasting paper), a printer for a run with local output (thus providing a permanent record), and a modem for a run that must be reported to central headquarters (thus sending the data over a communications link).

The procedure for reading data from a numbered input device is:

1. Load the starting address of the device table into an index register or stack pointer.
2. Load the device number (a variable) into an accumulator.
3. Read the data from the address obtained from the device table using indirect addressing with accumulator offset.

For example, if the starting address is IOTBL and the device number (assumed to be even) is in memory address IODEV, a typical program is:

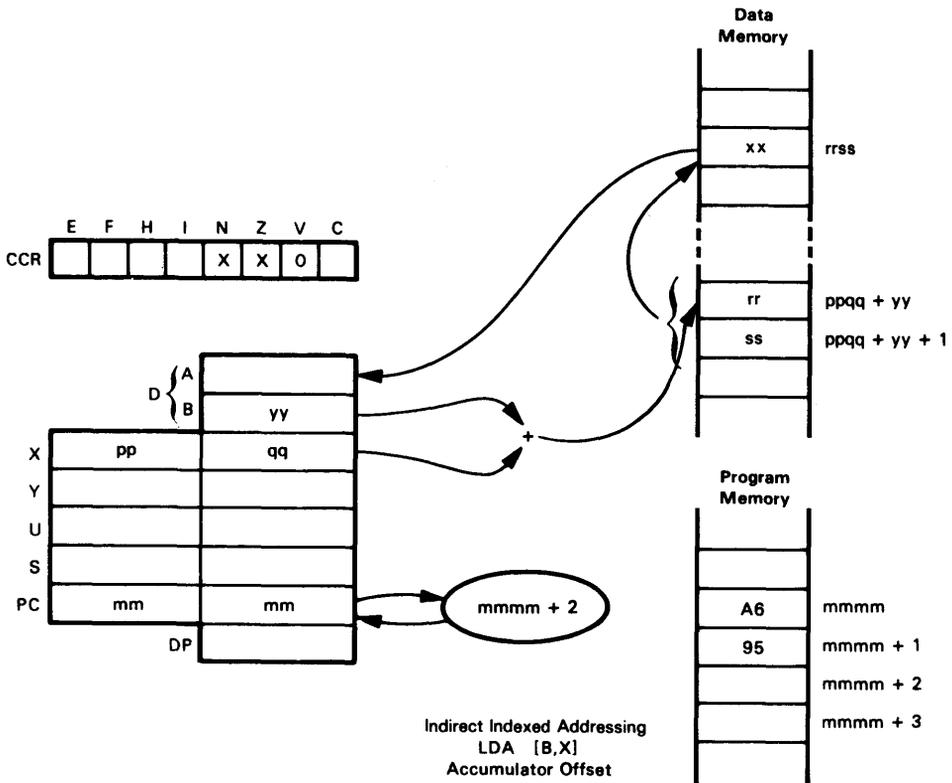
```
LDX    #IOTBL  GET BASE ADDRESS OF DEVICE TABLE
LDB    IODEV   GET I/O DEVICE NUMBER
LDA    [B,X]   GET DATA FROM I/O DEVICE VIA TABLE
```

Remember, the elements in the table are 2-byte addresses and we want to transfer data to or from those addresses, not to or from the table itself. **The entries in the table tell us where to send data or obtain data, not the value of the data** as in the code conversion example shown in the non-indirect case. Here again, the offset is a variable, since the same program must be able to convert various device numbers into actual I/O addresses.

As an example of the indirect accumulator offset mode, we will discuss the instruction

```
LDA [B,X]
```

which loads Accumulator A from the address starting at the address obtained by adding Accumulator B and Index Register X. The next diagram illustrates the execution of the instruction. The Accumulator A and double accumulator offsets are handled similarly, so we will not describe them in detail. The double accumulator offset is used for tables that exceed 256 bytes in length, a relatively infrequent situation. Clearly **this mode takes extra time (see Appendix B) because of the indirection. The processor must calculate where the indirect address is and fetch the indirect address from memory before it can actually execute the instruction. As with the non-indirect version, no additional program memory is necessary since the offset is in an accumulator or double accumulator.**



As a specific example, let us assume that Index Register X contains $03C6_{16}$ (the starting address of the I/O device table), Accumulator B contains 04_{16} (device #4), and memory addresses $03CA_{16}$ and $03CB_{16}$ (which hold the address corresponding to device #4) contain 80_{16} and 12_{16} respectively. Let us further assume that the data currently at address 8012_{16} (the input device port) is 43_{16} (an ASCII C). Then after the processor executes the instruction LDA [B,X] Accumulator A will contain $((X) + (B)) : ((X) + (B) + 1) = ((03C6_{16} + 04_{16}) : (03C6_{16} + 05_{16})) = ((03CA_{16}) : (03CB_{16})) = (8012_{16}) = 43_{16}$ (ASCII C, the data from the input port). The idea here is to use the table to determine where to find the data. The end result is that Accumulator A contains the data read from input device #4, which is accessed through memory address 8012_{16} , the corresponding entry in the device table.

AUTOINCREMENT AND AUTODECREMENT

In processing arrays, strings, or lists, we frequently want to process one byte and then proceed to the next byte which is located at the next higher address (if we are moving forward) or at the next lower address (if we are moving backwards). For example, if we are printing a string of characters (a message such as WATCH OUT - BOILER #6 IS REACHING CRITICAL TEMPERATURE), we must send the characters one by one to the printer (that is, first W, then A, then T, etc.). Similarly, if we are

averaging a set of ten readings, we must add them together one by one (for instance, start with zero, add the first reading, add the second reading, add the third reading, etc.) and finally divide by 10.

Thus to handle one byte and move forward, we must:

- **Reach the byte using the address in an index register or stack pointer.**
- **Add 1 to the index register or stack pointer to make it point to the next byte.**

The effect is like the action of a typewriter, which both prints the character for the key you press and moves the carriage along to the next position. Subtracting 1 from the index register or stack pointer would correspond to backspacing the typewriter's carriage. Unlike the typewriter, the computer does not prefer forward over backwards. Autoincrementing and autodecrementing are the modes most like the indexed addressing described in Volume 1 of *An Introduction to Microcomputers*.

Variations of Autoincrement and Autodecrement

The 6809 offers different step sizes for autoincrementing and autodecrementing. The base address may be:

- Incremented by 1 after it is used.
- Incremented by 2 after it is used.
- Decrement by 1 before it is used.
- Decrement by 2 before it is used.

The increment or decrement by 2 approach is useful when the array consists of 16-bit data or addresses. The processor thus moves on to the next element automatically, even though that element is located two bytes away from the current element. **Applying the increment after using the base but applying the decrement before using the base maintains compatibility with the automatic use of the stack pointers** (in JSR, PSH, PUL, RTI, RTS, and SWI instructions and in interrupt responses). Any access/change-pointer sequence could be implemented, but this is the most popular approach. All the user must remember is to load the base register with the starting address of the array or string for autoincrementing, but with the ending address plus 1 or 2 for autodecrementing (because the first autodecrement will reduce the base register before using it).

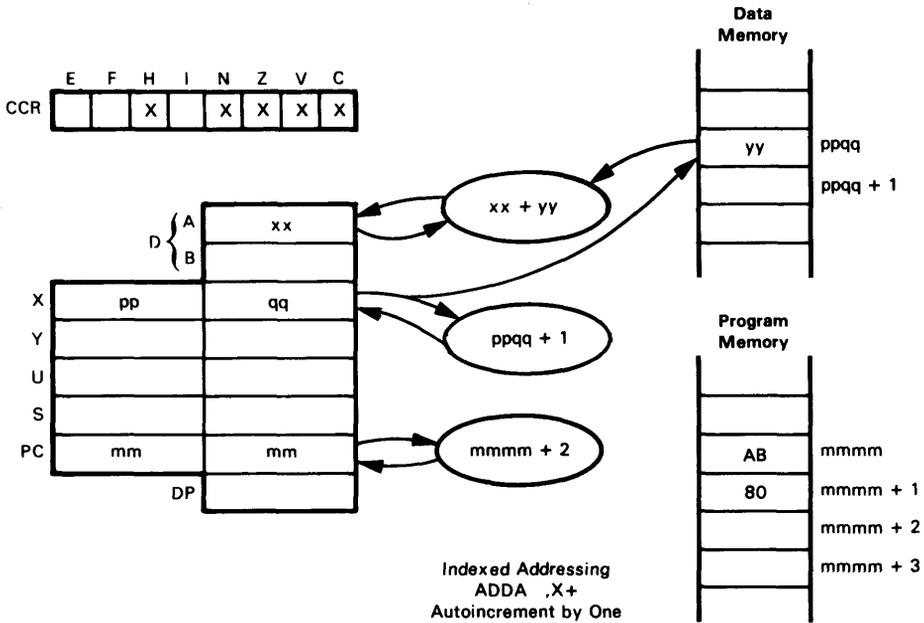
This form of indexed addressing is really a variety of implied memory addressing, since no offset is involved. Instructions using this mode take extra time (see Appendix B), since the processor must update the pointer register as well as execute the instruction. **Autoincrementing or autodecrementing is the simplest way to process arrays or strings since it provides automatic updating of the implied memory address** (or data pointer) as part of instruction execution. See Chapters 5 and 6 for further discussion of autoincrementing and autodecrementing.

Autoincrement with a Step of One

As one example, consider the instruction

```
ADDA ,X+
```

This instruction adds to Accumulator A the contents of the address in Index Register X. It also adds 1 to Index Register X, thus updating that address for the next operation in a summation or averaging program. The following diagram shows how the processor executes the instruction.



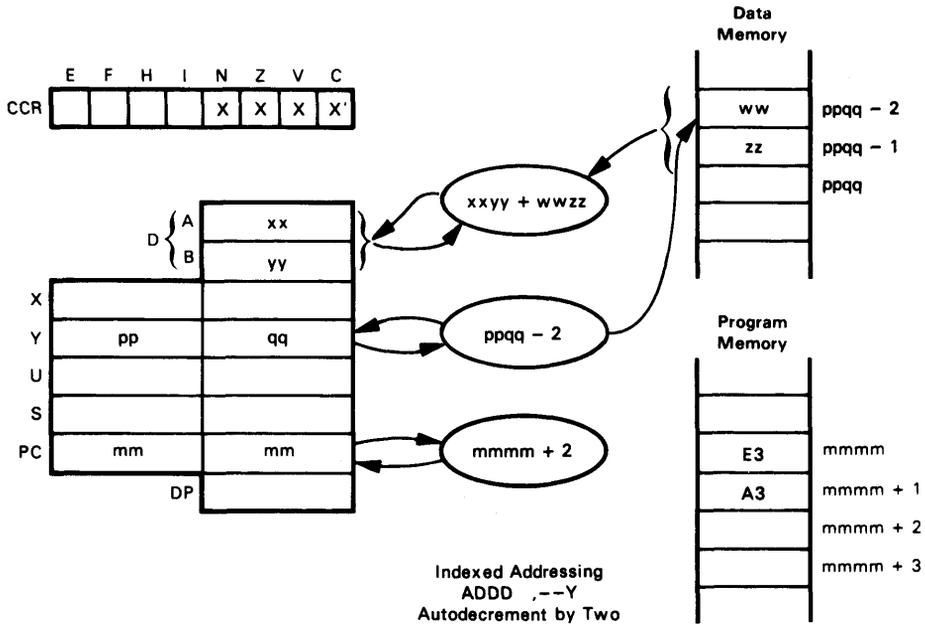
As a specific example, assume that Accumulator A contains 03_{16} , Index Register X contains $07E4_{16}$, and memory location $07E4_{16}$ contains 05_{16} . Then, after the processor executes `ADDA ,X+` Accumulator A will contain $03_{16} + ((X)) = 03_{16} + (07E4_{16}) = 03_{16} + 05_{16} = 08_{16}$. Furthermore, Index Register X will contain $07E4_{16} + 1 = 07E5_{16}$. Thus the instruction both adds an element to Accumulator A and updates Index Register X so it points to the next element.

Autodecrement with a Step of Two

As an example of both autodecrementing and a step of 2, let us show how the processor executes the instruction

`ADDD ,--Y`

This instruction adds to the double accumulator the contents of the address obtained by subtracting 2 from Index Register Y. It also places the result of the subtraction back in Index Register Y. Here the elements are 16 bits long, so a subtraction of 2 is necessary to reach the next element in the array. The step of 2 takes a little extra time (see Appendix B). The processor, of course, has no preference between autoincrementing and autodecrementing, since it lacks human or cultural preferences such as positive over negative, forward over backwards, left-to-right over right-to-left, or top-to-bottom over bottom-to-top. The following diagram illustrates the execution of the instruction.



As a specific example, let us assume that the double accumulator contains $10E8_{16}$, Index Register Y contains $042D_{16}$, and memory locations $042B_{16}$ and $042C_{16}$ contain 09_{16} and $5C_{16}$ respectively. Then, after the processor executes $ADD,--Y$ the double accumulator will contain $10E8_{16} + ((Y) - 2) : ((Y) - 1) = 10E8_{16} + (042B_{16}) : (042C_{16}) = 10E8_{16} + 095C_{16} = 1A44_{16}$. Furthermore, Index Register Y will contain $042D_{16} - 2 = 042B_{16}$. Thus the instruction first updates Index Register Y and then adds the current element to the double accumulator. The update by 2 is essential: decrement by 1 would point Index Register Y to the least significant half of the current element.

INDIRECT WITH AUTOINCREMENT OR AUTODECREMENT

This addressing mode allows us to handle arrays of addresses. For example, we have already described a table of I/O device addresses in which an entry is the actual address corresponding to a particular I/O device number. That is, entry 2 is the address corresponding to I/O device #2; by changing entry 2 (for instance, from a port that controls a machine to a port that is connected to a video display), we can change I/O devices and thus test the system, use the system as a remote terminal, or choose temporary or hard-copy output without making any changes in the underlying program. Let us assume that we want to fetch data from one device after another or test input devices until we find one that has new data available. We can fetch data from the first input device in a table INDEV with the sequence of instructions.

```
LDU    #INDEV  GET BASE ADDRESS OF INPUT DEVICE TABLE
LDA    [,U++]  GET DATA FROM DEVICE #0
```

After the processor executes these instructions, Accumulator A contains the data from device #0 and Stack Pointer U points to the address corresponding to device #2. Thus we can continue through the table of I/O devices, incrementing the pointer by 2 after

fetching data from a particular device. Obviously, we could equally well start two beyond the end of the table and use autodecrementing by 2 to move through the table backwards.

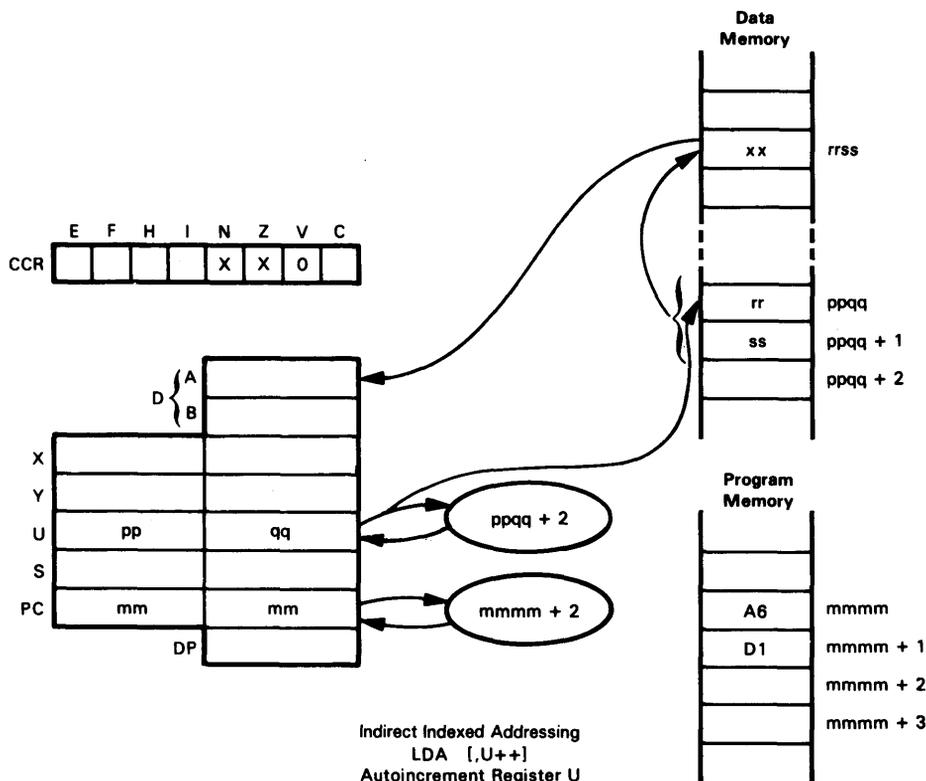
Since an address always occupies two bytes of memory, incrementing or decrementing by 1 makes no sense; it would result in the processor picking up half of one address and half of another. This mode is therefore not allowed with indirection, and the assembler will give you an error message if you try to use it. **The only valid options are:**

1. **Increment the base register by 2 after using it.**
2. **Decrement the base register by 2 before using it.**

As an example, let us show how the processor executes the instruction

LDA [,U++]

This instruction loads Accumulator A from the address starting at the address in Stack Pointer U. It also adds 2 to Stack Pointer U. Here Stack Pointer U points to an address; that is, it tells the processor where the address is, not where the data is. The following diagram illustrates the execution of the instruction:



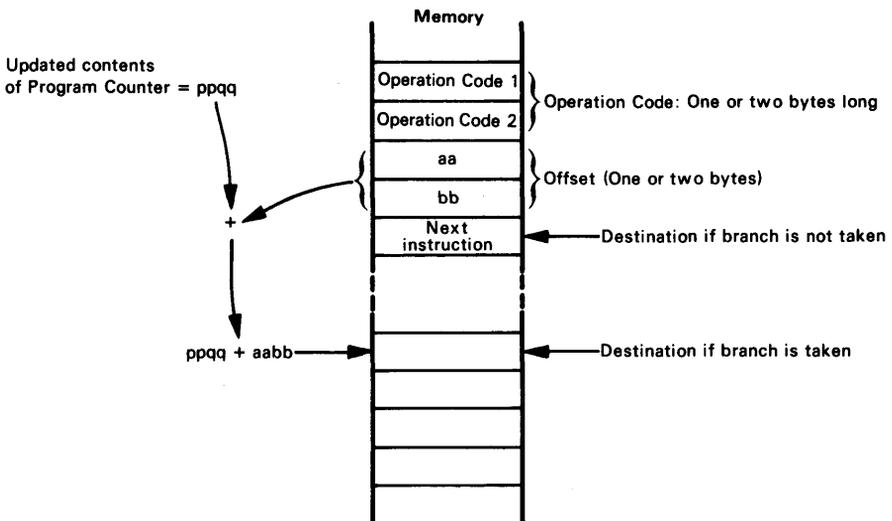
As a specific example, assume that Stack Pointer U contains $27EE_{16}$, memory address $27EE_{16}$ contains $C0_{16}$, memory address $27EF_{16}$ contains 07_{16} , and memory address $C007_{16}$ (the actual I/O port) contains 80_{16} . After the processor executes LDA [,U++] Accumulator A will contain $((U):(U+1)) = ((27EE_{16}):(27EF_{16})) = (C007_{16}) = 80_{16}$. Furthermore, Stack Pointer U will contain $27EE_{16} + 2 = 27F0_{16}$. Thus

the instruction loads Accumulator A from an indirect address obtained from the table and updates Stack Pointer U so it points to the next indirect address. Of course, **this process of picking up an indirect address from the table, loading the data from that address, and updating the pointer takes many extra clock cycles** (see Appendix B). Note, however, that **no extra bytes of program memory are needed, since no offset is involved.**

PROGRAM RELATIVE ADDRESSING FOR BRANCHES

Branch, Branch-on-Condition, and Branch-to-Subroutine instructions use only program relative addressing in which the address value is the offset from the current value of the program counter. Thus branches are specified by “how far from where we are,” rather than by an actual destination address. This mode allows us to relocate an entire program, since such a move does not change any relative addresses. **Relative branches are a key element in producing relocatable or position-independent code.** Furthermore, since most branches in programs are short, **relative addressing allows shorter addresses** (usually eight bits), thus reducing memory usage.

The following illustration shows how the 6809 microprocessor executes relative branch instructions. The value ppqq is the contents of the program counter after the processor has fetched the entire branch instruction from memory. That instruction includes an operation code (one or two bytes long) and an offset (one or two bytes long).



The 6809 microprocessor has two forms of relative addressing: 8-bit offset and 16-bit offset. In both forms, the value following the operation code specifies how many memory locations to skip over from the end of the instruction. **The offset is a twos complement number, so the range for the 8-bit form is**

$$-128 (= 1000\ 0000_2 \text{ or } 80_{16}) \leq \text{offset} \leq +127 (= 0111\ 1111_2 \text{ or } 7F_{16})$$

Since the short relative branches themselves occupy two bytes of program memory, the range from the start of the instruction is

$$-126 \leq \text{offset} \leq +129$$

We do not have to be concerned with this extra factor of 2 if we specify the actual destination in the operand field. If, for example, we use the statement

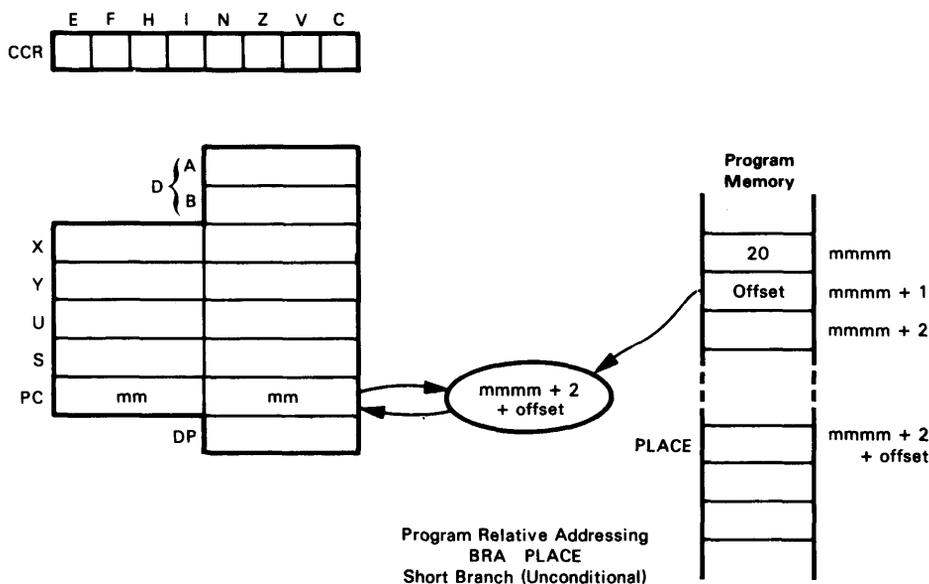
```
BRA CHCNT
```

the assembler will figure how far away label CHCNT is (including the factor of 2) and place that number in the offset. We will discuss calculating relative offsets in more detail in Chapter 4.

As an example of how the processor executes relative branch instructions, consider the instruction

```
BRA PLACE
```

where PLACE is a nearby address. If the program counter contains mmmm originally, the offset is the 8-bit two's complement form of $\text{PLACE} - (\text{mmmm} + 2) = \text{PLACE} - \text{mmmm} - 2$. The next diagram illustrates the execution of the instruction. The 16-bit offset form is similar, except that the offset occupies two bytes and the instructions therefore occupy either three bytes (LBRA and LBSR) or four bytes (all long conditional branches). The extra factor in the address calculation is then either 3 or 4, making hand calculations even more awkward. As we mentioned above, **the assembler will perform the calculation for you if you specify the destination address as a label; you will seldom need to calculate offsets by hand.** The 16-bit offset provides access to any location in memory, but is not commonly needed since few branches are long enough to require its use. **Another approach to providing relative addressing with branches is to use the Jump or Jump-to-Subroutine instructions with the indexed addressing mode that involves an offset from the program counter. The non-indirect versions of these instructions, however, take more time and memory than ordinary relative branches and so are not used.**



As a specific example, assume that $mmmm = C5A1_{16}$ and $PLACE = C5BE_{16}$. The offset is $C5BE_{16} - C5A1_{16} - 2 = 1D_{16} - 2 = 1B_{16}$. After the processor executes the instruction `BRA PLACE`, the contents of the program counter will be (initial PC) + 2 + offset = $C5A1_{16} + 2 + 1B_{16} = C5A3_{16} + 1B_{16} = C5BE_{16} = PLACE$. Note that if we move the entire program forward or backwards by a distance REL, the new offset is $(C5BE_{16} + REL) - (C5A1_{16} + REL) - 2 = C5BE_{16} - C5A1_{16} - 2$, the same as before since the RELs cancel out.

6809 INSTRUCTION SET

Table 3-5 lists the 6809's instruction mnemonics, differentiating between those that are also 6800 mnemonics and those that are new or have been modified. We will discuss compatibility between the 6809 microprocessor and the 6800 microprocessor, as well as compatibility between the 6809 and the 6801 microprocessors, in the next part of this chapter. **For a detailed description of the 6809 instruction set, see the last section of this book.** In Chapter 22, we discuss each instruction's operation; refer to that chapter when you need to understand how a particular instruction works. Appendix A summarizes the available 6809 instructions, grouping them by function. This provides a survey of the 6809's capabilities, and will also be useful when you need a certain kind of operation but are either unsure of the specific mnemonic or not yet familiar with what instructions are available. The rest of the appendices serve as reference tables for calculating program execution time and memory requirements, and for hand assembly and disassembly; Appendix C also displays available addressing modes for each instruction.

Instructions often frighten microcomputer users who are new to programming. Yet taken in isolation, the operations involved in the execution of a single instruction are usually easy to follow. The purpose of the last section of this book is to isolate and explain those operations. Furthermore, you need not attempt to understand all the instructions at once. As you study each of the programs in this book you will learn about the specific instructions involved.

Why are a microprocessor's instructions referred to as an instruction "set?" Because the microprocessor designer selects the instruction complement with great care; it must be easy to execute complex operations as a sequence of simple events, each of which is represented by one instruction from a well-designed instruction "set."

6800/6809 COMPATIBILITY

The 6809 microprocessor is an advanced version of the 6800 microprocessor, produced by the same manufacturers. All assembly language programs written for the 6800 microprocessor can also be assembled for the 6809 microprocessor. In fact, object code produced for the 6800 microprocessor is very similar to that produced for the 6809 microprocessor; in many cases, the processors have direct object code compatibility. **The external support devices designed for use with the 6800 microprocessor can all be used with the 6809 as well.** Chapter 9 of *An Introduction to Microcomputers: Volume 2* discusses the hardware compatibility in more detail.

Table 3-5. 6809 Operation Code Mnemonics

Instruction	Source Forms	Instruction	Source Forms	Instruction	Source Forms	Instruction	Source Forms
ADC	ADCA ADCB ADDA ADDB	BLS	BLS LBLS	DEC	DECA DECB DEC EORA EORB	OR	ORA ORB
ADD	ADDD	BLT	BLT LBLT	EOR		PSH	ORCC PSHS ¹¹ PSHU PULS ¹²
AND	ANDA ANDB	BMI	BMI LBMI	EXGR1,R2 ²		PUL	PULU
ASL ³	ASLA ASLB	BNE	BNE LBNE	INC	INCA INCB INC	ROL	ROLA ROLB ROL
ASR ^{3,6}	ASRA ASRB	BPL	BPL LBPL	JMP		ROR ⁶	RORA RORB ROR
BCC	BCC LBCC	BRA	BRA LBRA	JSR			
BCS	BCS LBCS	BRN	BRN LBRN	LD	LDA ¹⁰ LDB ¹⁰		
BEQ	BEQ LBEQ	BSR	BSR LBSR		LDD LDS LDU	RIT ⁸ RTS SBC ³	SBCA SBCB
BGE	BGE LBGE	BVC	BVC LBVC	LEA	LDY LEAS LEAU LEAX LEAY	SEX ST	STA ¹⁰ STB ¹⁰ STD STS STU STX STY
BGT	BGT LBGT	BVS	BVS LBVS	LSL ³	LSL LSRA LSRB LSR	SUB ³	SUBA SUBB SUBD SUBI
BHI	BHI LBHI	CLR	CLRA CLRB CLR	MUL ⁴		SWI ⁹	SWI SWI2 SWI3
BIT	BITA BITB	CMP ³	CMPA CMPB CMPD CMPX ⁷ CMPY	NEG ³	NEGA NEGB NEG		
BLE	BLE LBLE	COM	COMA COMB COM	NOP		SYNC TFR,R1,R2 ² TST ⁵	TSTA TSTB TST
BLO	BLO LBLO	CWAI DAA					

Notes:

- Shading identifies additions or modifications to the 6800 instruction set. The unshaded instructions are also 6800 instructions with the same operation codes except as noted below.
- R1 and R2 may be any pair of 8-bit or 16-bit registers. The 8-bit registers are A, B, CC, and DP. The 16-bit registers are D, X, Y, U, S, and PC.
- The Half-Carry flag H is undefined after these instructions are executed.
- This MUL affects the Zero flag, whereas 6801 MUL does not.
- This instruction does not affect the Carry flag. On the 6800/6801/6802 it clears the C flag.
- These do not affect the Overflow flag (V). On the 6800/6801/6802 they may.
- This instruction correctly sets all flags. On the 6800/6802 it does not.
- On the 6809, the Entire flag (E) is checked during RTI to determine how much to unstack — the entire register complement or just the Condition Code Register and Return Address.
- SWI sets the F and I flags; SWI2 and SWI3 have no effect on F and I.
- These instructions are implemented on the 6800 with slightly different mnemonics.
- This instruction is implemented on the 6800 as PSH.
- This instruction is implemented on 6800 as PUL.

We will briefly describe and compare the 6809 and 6800 microprocessors with regard to their registers, flags, addressing modes, and instruction sets. The processors are similar, and the manufacturers clearly will encourage migration from the 6800 to the 6809. This description will help you see what problems you would encounter in going from one CPU to the other.

REGISTERS

The 6800 register set is a subset of the 6809 register set. In addition to the 6800 registers — Condition Code, Accumulators A and B, Index Register X, and the Hardware Stack Pointer — the 6809 has another index register (Y), another stack pointer (User Stack Pointer or U register), and a direct page register. Also, the 6809 allows references to the Double Accumulator D, which consists of Accumulator A and Accumulator B, whereas the 6800 does not.

FLAGS

The 6800 and 6809 microprocessors have identical Sign, Zero, Overflow, Carry, Half (or Auxiliary) Carry, and Interrupt Mask flags. The 6809 also has a Fast Interrupt Mask flag (F) and an Entire flag (E) which are not implemented on the 6800, since the 6800 has no Fast Interrupt Request input and always saves all of its registers in response to an interrupt. Bit positions 6 and 7 in the 6800's condition code register always contain ones.

ADDRESSING MODES

The 6809 microprocessor has many more addressing modes than does the 6800. The only indexed addressing mode that is implemented on the 6800 is the non-indirect mode with an 8-bit unsigned offset from Index Register X. All other indexed and indirect addressing modes are unique to the 6809. You should note that 6809 indexed instructions all require an extra (or post) byte that determines the addressing mode. Thus an indexed instruction that required two bytes of code on a 6800 microprocessor will generally require three bytes on a 6809 microprocessor. However, indexed addressing on the 6800 is most often used with an offset that is either zero or less than 16; instructions with such small offsets can be implemented on the 6809 microprocessor using the special forms for zero offset or 5-bit signed offset, thus making them again two bytes in length. You should also note that indexed offsets are signed in the 6809 microprocessor (allowing them to be either positive or negative), whereas they are unsigned in the 6800 microprocessor.

The direct addressing mode on the 6809 microprocessor differs from the direct mode on the 6800 because the 6809 has a Direct Page register which provides the high-order byte of the address. The 6800 microprocessor always sets the high-order byte of the address to zero. Thus 6800 and 6809 direct addressing are the same only when the 6809's Direct Page register contains zero. Compatibility is simplified by the fact that hardware Reset clears the 6809 Direct Page register, so that it contains zero unless the program explicitly changes it. Obviously, the 6809 direct mode is more powerful and more general.

INSTRUCTIONS

The 6800 instruction set is a subset of the 6809 instruction set. Many 6800 and 6809 instructions are identical (see Table 3-6). Some new 6809 instructions (see Table 3-7) are obvious additions to the 6800 set, required to handle the new 6809 registers. Still other 6809 instructions are generalizations of 6800 instructions (see Table 3-8) or entirely new (see Table 3-9).

Table 3-10 describes the implementation of 6800 instructions that no longer exist on the 6809 microprocessor. Note that the 6809 assembler automatically translates these 6800 instructions into their 6809 equivalents. All these one-byte 6800 instructions require at least two bytes (and sometimes as many as four bytes) on the 6809. However, most of them are rarely used in 6800 programs. The only common 6800 instructions in Table 3-10 are PSH and PUL, which have been greatly generalized on the 6809 to handle its larger set of registers, and DEX and INX, which have become far less important on the 6809 with the addition of autoincrementing and autodecrementing.

6800/6809 DIFFERENCES

You should note the following minor differences between the 6800 and 6809 instruction sets:

1. The 6809 regards Accumulators A and B as a Double Accumulator D, with A as the high-order half. It therefore **stacks and unstacks the accumulators with B stacked first and removed last**; this is the opposite order from that implemented on the 6800 microprocessor.
2. The 6809's hardware stack pointer contains the address of the last memory location occupied by the stack, not the address of the next empty location as in the 6800. Thus 6809 instructions that use the hardware stack pointer always decrement it before storing data and increment it after loading data. 6800 instructions that use the stack pointer always decrement it after storing data and increment it before loading data. Thus the hardware stack pointer on the 6809 should be initialized to a value one larger than that used in a comparable 6800 program.
3. The 6800 instructions **TSX (Transfer Stack Pointer to Index Register) and TXS (Transfer Index Register to Stack Pointer)** took account of the fact that the 6800's stack pointer contained the address one beyond the end of the stack. This accounting involved an addition of 1 during TSX and a subtraction of 1 during TXS, thus moving to or from the last occupied address. The 6809 microprocessor does not require this awkward adjustment and therefore does not implement it in instructions.
4. The 6809 TST instruction does not affect the Carry flag, whereas the 6800 TST instruction clears that flag.
5. The 6809 right shifts (ASR, LSR, ROR) do not affect the Overflow flag, whereas the 6800 right shifts do.
6. The 6809 Half-Carry flag is undefined after subtraction, comparison, and related instructions (NEG), whereas the 6800 Half-Carry flag is cleared after such instructions.

Table 3-6. Identical 6800/6809 Instructions

6800 Mnemonic	6809 Mnemonic	Notes
ADCA/ADCB	ADCA/ADCB	
ADDA/ADDB	ADDA/ADDB	
ANDA/ANDB	ANDA/ANDB	
ASL	ASL (also LSL)	1
ASR	ASR	1
BCC	BCC (also BHS)	
BCS	BCS (also BLO)	
BEQ	BEQ	
BGE	BGE	
BGT	BGT	
BHI	BHI	
BIT	BIT	
BLE	BLE	
BLS	BLS	
BLT	BLT	
BMI	BMI	
BNE	BNE	
BPL	BPL	
BRA	BRA	
BSR	BSR	
BVC	BVC	
BVS	BVS	
CLR	CLR	1
CMPA/CMPB	CMPA/CMPB	
COM	COM	1
CPX	CMPX	3
DAA	DAA	
DEC	DEC	1
EOR	EOR	
INC	INC	1
JMP	JMP	1
JSR	JSR	
LDAA/LDAB	LDA/LDB	
LDS	LDS	2
LDX	LDX	2
LSR	LSR	1, 3
NEG	NEG	1
NOP	NOP	2
ORAA/ORAB	ORA/ORB	
PSH	PSHS	2, 4
PUL	PULS	2, 4
ROL	ROL	1
ROR	ROR	1, 3
RTI	RTI	
RTS	RTS	
SBC	SBC	
STAA/STAB	STA/STAB	
STS	STS	2
STX	STX	2
SUBA/SUBB	SUBA/SUBB	
SWI	SWI	
TST	TST	1, 3

Note the minor differences in some of the mnemonics — namely, an extra A in LDAA, LDAB, ORAA, ORAB, STAA, and STAB on the 6800, an extra S in PSHS and PULS on the 6809, and a slightly different version of Compare Index Register X (CMPX on 6809, CPX on 6800).

Notes:

1. Direct addressing is available with this instruction on the 6809 only.
2. 6809 instruction has a different object code.
3. 6809 version has slightly different effects on flags.
4. 6809 Stack Pointer manipulation differs from that of the 6800. See the text for further information.

7. The 6809 sets all flags properly after executing CMPX and similar instructions, whereas the 6800 sets only the Z flag properly after executing its CPX instruction.

Clearly these differences will not affect most programs, unless they perform many stack manipulations. There are slight differences in operations involving the Condition Code register, since the 6800 always has ones in the two most significant bits of that register, whereas the 6809 uses those bits for the Entire flag and the Fast Interrupt Mask bit.

Table 3-7. 6809 Instruction Set Extensions to Handle Additional Registers

6809 Operation	Comparable 6800 Operation
CMPY	CPX
LDU	LDS
LDY	LDX
PSHU	PSH
PULU	PUL
STU	STS
STY	STX

Table 3-8. 6809 Generalizations of 6800 Instructions

6809 Operation	Comparable 6800 Operations
ADDD	ADDA, ADDB
ANDCC	CLC, CLI, CLV
CMPD	CMPA, CMPB
CMPS	CPX
CMPU	CPX
CWAI	WAI
EXG	TAB, TAP, TBA, TPA, TSX, TXS
LBCC (also LBHS)	BCC
LBCCS (also LBLO)	BCS
LBEG	BEQ
LBGT	BGE
LBHI	BGT
LBLE	BHI
LBLS	BLE
LBLT	BLS
LBMI	BLT
LBNE	BMI
LBPL	BNE
LBRA	BPL
LBSR	BRA
LBVC	BSR
LBVS	BVC
LDD	BVS
ORCC	LDAA, LDAB
STD	SEC, SEI, SEV
SUBD	STAA, STAB
SWI2	SUBA, SUBB
SWI3	SWI
TFR	SWI
	TAB, TAP, TBA, TPA, TSX, TXS

Table 3-9. New 6809 Instructions (Without 6800 Equivalents)

Instruction Mnemonic	
ABX	(Also implemented on 6801 microprocessor)
BRN	
LBRN	
LEA	
SEX	
SYNC	(but similar to 6800 WAI)

Table 3-10. 6809 Implementations of Missing 6800 Instructions

6800 Instruction	6809 Equivalent
ABA	PSHS B; ADDA ,S+
CBA	PSHS B; CMPA ,S+
CLC	ANDCC #% 11111110
CLI	ANDCC #% 11101111
CLV	ANDCC #% 11111101
DES	LEAS -1,S
DEX	LEAX -1,X
INS	LEAS 1,S
INX	LEAX 1,X
PSHA	PSHS A *
PSHB	PSHS B *
PULA	PULS A *
PULB	PULS B *
SBA	PSHS B; SUBA ,S+
SEC	ORCC #% 00000001
SEI	ORCC #% 00010000
SEV	ORCC #% 00000010
TAB	TFR A,B; TSTA
TAP	TFR A,CC
TBA	TFR B,A; TSTA
TPA	TFR CC,A
TSX	TFR S,X
TXS	TFR X,S
WAI	CWAI # \$FF or CWAI # \$EF to enable regular interrupt (replaces CLI, WAI)

* 6809 Stack Pointer manipulation differs from that of the 6800. See the text for further information.

6801/6809 COMPATIBILITY

The 6801 microprocessor is a slightly improved version of the 6800 microprocessor that is manufactured by some of the same companies. The 6801 instruction set is almost the same as the 6800's except that the 6801 has a multiplication instruction and an ABX instruction (as on the 6809), as well as 16-bit shifts for the double accumulator that are not implemented on either the 6800 or the 6809. The 6801, like the 6809, does set the flags properly after CMPX (CPX). The only added difference is that the 6801 multiply instruction (MUL) does not affect the Zero flag, whereas the 6809 MUL instruction does.

6502/6809 COMPATIBILITY

The 6809 microprocessor is also similar to the 6502 and related microprocessors, which are produced by a different group of manufacturers. For more details on 6502 compatibility, see the discussions in Chapters 9 and 10 of *An Introduction to Microcomputers: Volume 2* and in Chapter 3 of *6502 Assembly Language Programming*.³

MOTOROLA 6809 ASSEMBLER CONVENTIONS

The standard 6809 assembler is available from 6809 manufacturers and on many major time-sharing networks; it is also included in most development systems. Cross-assembler versions are available for most large computers and many minicomputers.

ASSEMBLER FIELD DELIMITERS

The assembly language instructions have the standard field structure (see Table 2-1). The required delimiters are:

1. **A space after a label.** All labels must start in column 1 and all statements that are not labeled must start with at least one space.
2. **A space after the operation code.** The accumulator, double accumulator, or index register/stack pointer designation can be added to the operation code without a space; for instance, ADDA for "Add to Accumulator A," STD for "Store Double Accumulator," and PSHU for "Push Registers onto User Stack."
3. **A comma between operands in the address field** — that is, between an offset value or register and a base register (X, Y, U, S, or PC). For example, ADDA \$35,X means that an indexed instruction is to be generated with an offset of 35₁₆ from the value in Index Register X. A zero offset can be omitted unless the base register is the program counter.
4. **A comma in front of the symbols for autoincrementing or autodecrementing.** For example, LDA ,X+ tells the assembler to generate an indexed LDA instruction which autoincrements Index Register X by 1. Similarly, ADDB ,--U tells the assembler to generate an indexed ADDB instruction which autodecrements Stack Pointer U by 2. This comma is similar to the one between operands (item 3 above), although no offset is allowed with autoincrementing or autodecrementing.
5. **Square brackets — [] — around addresses to be used indirectly.**
6. **A space before a comment that appears on the same line as an instruction, and an asterisk before an entire line of comments.**

Typical 6809 assembly language instructions are:

```
START LDA [1000,X] GET LENGTH
      LDX TEMPR
      WAI
```

LABELS

Most versions of the assembler allow only six characters in labels and truncate longer labels. The first character must be a letter or the special character period (.). The assembler reserves certain names to refer to CPU registers; these names are A, B, CC, D, DP, PC, PCR (program counter relative), S, U, X, and Y. The use of operation mnemonics as labels is often not allowed and is not good programming practice anyway, because of the obvious confusion.

ASSEMBLER DIRECTIVES

The assembler has the following explicit pseudo-operations:

END	—	End of Source Program
EQU	—	Equate or Define Symbolic Name
FCB	—	Form Constant Byte or Enter Byte-Length Data
FCC	—	Form Constant Character String or Enter Character Data
FDB	—	Form Double Byte Constant or Enter Word-Length Data
ORG	—	Set (Location Counter to) Origin
RMB	—	Reserve Memory Bytes or Allocate Storage
SETDP	—	Set Direct Page Pseudo-Register

FCB, FCC, and FDB

FCB, FCC, and FDB are the data directives used to place constant data in program memory — data such as tables, messages, and numerical factors — that is necessary for the execution of the program but does not consist of instructions. FCB is used for byte-length (8-bit) data, FCC for 7-bit ASCII characters (MSB of each byte is zero), and FDB for word-length (16-bit) data or addresses. Note that FDB stores word-length data in the standard 6800-6809 format with the high-order bits in the first byte and the low-order bits in the following byte.

Examples:

```
ADDR FDB $3155
```

places the numbers 31_{16} and 65_{16} in the next two bytes of program memory and assigns the name ADDR to the address of the first byte; thus $(ADDR) = 31_{16}$ and $(ADDR + 1) = 65_{16}$.

```
TCONV FCB 32
```

places the number 32 (20_{16}) in the next byte of program memory and assigns the name TCONV to the address of that byte.

```
ERROR FCC /ERROR/
```

places the 7-bit ASCII character representations of E, R, R, O, and R (hexadecimal 45, 52, 52, 4F, and 52) in the next five bytes of program memory and assigns the name ERROR to the address of the first byte.

Any single character (not just /) may be used to surround the ASCII text. An alternative is to specify the number of characters in the operand field. For example:

```
ERROR FCC 5,ERROR
```

We will always use the first form shown (with the / character) for consistency.

```
OPERS FDB FADD, FSUB, FMUL, FDIV
```

places the addresses FADD, FSUB, FMUL, and FDIV in the next eight bytes of memory and assigns the name OPERS to the address of the first byte. All addresses (and 16-bit data items) are stored with their high-order bits first.

RMB

RMB is the Reserve directive used to assign locations in memory for specific purposes; it allocates a specified number of bytes.

EQU

EQU is the Equate or Define directive used to define names.

ORG

ORG is the standard Origin directive. 6809 assembly language programs usually have several origins, which are used for the following purposes:

1. To specify the Reset, interrupt service, and software interrupt addresses. These addresses must be placed in the highest memory addresses in the system ($FFF2_{16}$ through $FFFF_{16}$).
2. To specify the starting addresses of the actual Reset, interrupt service, and software interrupt routines. The routines themselves may be placed anywhere in memory.
3. To specify the starting address of the main program.
4. To specify the starting address of subroutines.
5. To define areas of memory for data storage.
6. To define areas of memory for the Hardware and User Stacks.
7. To specify addresses used for I/O ports and special functions.

Examples:

```
RESET EQU $3800
      ORG RESET
      :
      :
      ORG $FFFE
      FDB RESET
```

Note: \$ means 'hexadecimal'.

This sequence places the Reset (or startup) instruction sequence in memory beginning at address 3800_{16} , and places that address in the memory locations (addresses $FFFE_{16}$ and $FFFF_{16}$) from which the 6809 CPU retrieves the Reset address.

```
MAIN EQU SC000
     ORG MAIN
```

This sequence specifies that the instructions following it are to be placed in memory beginning at address $C000_{16}$.

4. **Binary** (must start with % or end with B)

Example: %00101 or 00101B

We will use the “%” symbol to maintain compatibility with the 6800 assembler.

5. **ASCII** (single character preceded by an apostrophe)

Example: 'H

6. **As an offset from the current value of the location counter (*).**

Example: *+7

7. **Relative to the current value of the location counter (DEST, PCR)**

Example: LDA TABLE,PCR

The assembler will generate an indexed LDA instruction using the mode based on a constant offset from the program counter. The value of the offset will be the relative distance between TABLE and the current value of the location counter. Note the difference between LDA TABLE,PCR and LDA TABLE,PC: the latter generates an indexed LDA instruction with TABLE as the value of the offset to be added to the program counter. The assembler automatically calculates the relative distance to the destination when the programmer uses the PCR notation.

Distinguishing Addressing Modes

The various 6809 addressing modes are distinguished as follows:

1. **Direct and Extended are the default modes.** The assembler chooses direct addressing if the address is on the page specified as the direct page. Remember that the direct page is page 0 unless a SETDP directive specifies otherwise. You can force the assembler to use direct addressing by preceding the address with the “<” character and to use extended addressing by preceding the address with the “>” character.
2. **The symbol # precedes the data for immediate mode.**
3. **OFFSET,R specifies indexed non-indirect modes with offsets.** R must be one of the registers PC, S, U, X, or Y. You can force an 8-bit offset mode by preceding the operand with the “<” character and a 16-bit offset mode by preceding the operand with the “>” character. The assembler will automatically choose the zero offset, 5-bit offset, or 8-bit offset mode if the mode is available and the offset is the correct size.
4. **The form DEST,PCR specifies the indexed mode that adds a constant offset to the program counter, and furthermore directs the assembler to calculate the offset as the relative distance to the address labeled DEST.**
5. **Square brackets enclose addresses to be used indirectly.**
6. **The symbol + or ++ after the register name (S, U, X, or Y) specifies autoincrementing, and - or -- before the register name (S, U, X, or Y) specifies autodecrementing.**

Assembler Arithmetic and Logical Expressions

The assembler also allows expressions in the address field. These expressions consist of numbers and names separated by the arithmetic operators +, -, * (multiplication), or / (integer division), or the following special two-character operators:

!^	—	exponentiation	!<	—	shift left
!	—	logical AND	!>	—	shift right
!+	—	logical (inclusive) OR	!L	—	rotate left
!X	—	logical Exclusive OR	!R	—	rotate right

The precedence of the various operators is as follows:

1. Expressions within parentheses are evaluated first.
2. Multiplication, division, and the two-character operators have precedence over addition and subtraction.
3. Operators with the same precedence are evaluated from left to right.

All intermediate results are truncated to 16-bit integers and all fractional results are dropped.

We recommend that you avoid expressions within address fields whenever possible, since there are no standards for calculating such addresses. If you must compute an address, comment any unclear expressions and be sure that the evaluation of the expressions never produces a result which is too large for its ultimate use.

OTHER ASSEMBLER FEATURES

Most 6809 assemblers have additional features, including both macro and conditional assembly capabilities. You should consult your particular assembler's manual for a description of how these features are implemented. We will not use any of these features or refer to them again, although they can be quite convenient in many applications.

REFERENCES

Terry Ritter and Joel Boney, the co-architects of the Motorola 6809, have described its architecture and instruction set in a series of three very interesting articles entitled "A Microprocessor for the Revolution: The 6809" in *BYTE* magazine. These articles describe the philosophy that resulted in the 6809 microprocessor and answer many questions about approaches to problems, design decisions, and tradeoffs. The specific articles in the series are:

- T. Ritter and J. Boney, "A Microprocessor for the Revolution: The 6809. Part 1: Design Philosophy," *BYTE*, January 1979, pp. 14-42.
- T. Ritter and J. Boney, "A Microprocessor for the Revolution: The 6809. Part 2: Instruction Set Dead Ends, Old Trails and Apologies," *BYTE*, February 1979, pp. 32-42.
- T. Ritter and J. Boney, "A Microprocessor for the Revolution: The 6809. Part 3: Final Thoughts," *BYTE*, March 1979, pp. 46-52.

1. A. Osborne and J. Kane, *An Introduction to Microcomputers: Volume 2—Some Real Microprocessors*. Berkeley: Osborne/McGraw-Hill, 1979.
2. A. Osborne, *An Introduction to Microcomputers, Volume 1 — Basic Concepts*, 2nd ed. Berkeley: Osborne/McGraw-Hill, 1980.
3. L. Leventhal, *6502 Assembly Language Programming*. Berkeley: Osborne/McGraw-Hill, 1979.





Introductory Problems

The only way to learn assembly language programming is through experience. The next six chapters of this book contain examples of simple programs that perform actual microprocessor tasks. You should read each example carefully and try to execute the program on a 6809-based microcomputer. Finally, you should work the problems at the end of each chapter and run the resulting programs on your microcomputer to ensure that you understand the material.

GENERAL FORMAT OF EXAMPLES

Each program example contains the following parts:

- **A title** that describes the problem
- **A statement of purpose** that describes the specific tasks the program performs and the memory locations it uses
- **A sample problem** with data and results
- **A flowchart if the program logic is complex**
- **The source program** or assembly language listing
- **The object program** or hexadecimal machine language listing
- **Explanatory notes** that discuss the instructions and methods used in the program

You should use the examples as guides for solving the problems at the end of each chapter. Be sure to run your solutions on a 6809-based microcomputer to ensure that they work correctly.

Program Listing Format

We reproduce Program 4-1 below to illustrate the format for program listings which we will use in this book. This is a common format for assembler output; it shows the object code as well as the source code.

Memory Address	Object Code	Source Program
0000	96 40	LDA \$40 GET DATA
0002	97 41	STA \$41 TRANSFER TO NEW LOCATION
0004	3F	SWI

The 4-digit number starting in the leftmost column of each line is the hexadecimal address of the first byte of object code generated from the line of source code. For example, in the second line 0002 is the address of the object code byte for STA (base page direct addressing form). The digits following the address are the hexadecimal object code for the instruction. Thus, in the second line, 97 41 is the object code for STA \$41, and the byte 97 is in location 0002. The byte 41 is in location 0003; we infer this from the fact that it follows the byte in address 0002. The letters, numbers, and words to the right of the object code are the assembly language fields which we described in Chapter 2. These fields comprise the source program.

If you wish to assemble these examples on your microcomputer, key in the source statements only; do not enter the addresses or object codes, since the assembler program will generate them. You will also need to enter some assembler directives — for example, to tell the assembler where to start program addresses. We may not show all the necessary directives; the ones you use will be determined by your assembler and the requirements of your microcomputer's operating system.

If you wish to execute the program examples without assembling source code, you can key the object code into the specified addresses. Before you do this, however, make sure that you will not be trying to load areas of memory reserved for the monitor or operating system. To avoid such problems, you may need to change addresses before you load the programs. As we will discuss in the seventh guideline below, you may also need to change the instruction at the end of the program.

Guidelines for Examples

We have used the following guidelines in constructing the examples:

- **Standard 6809 assembler notation** as summarized in Chapter 3
- **Use of the clearest possible forms for expressing data and addresses.** We use hexadecimal numbers for memory addresses, instruction codes, and binary-coded decimal (BCD) data; decimal for numeric constants; binary for logical masks; and ASCII (American Standard Code for Information Interchange) for characters
- **Emphasis on frequently used instructions and common programming techniques**
- **Drawing of problems from actual microprocessor applications** in communications, instrumentation, computers and peripherals, business equipment, industrial and process control, and military systems
- **Extensive commenting** for instructional purposes, often more than we would typically include in actual programs

- **Emphasis on simple, clear structure**, while still making programs as efficient as possible within this guideline. The notes often describe more efficient procedures
- **Use of a standard set of memory addresses.** Each program starts in memory location 0000_{16} , uses memory addresses starting at 0040_{16} for temporary data storage, and ends with the SWI (Software Interrupt) instruction. If your microcomputer has no monitor and no interrupts, you may prefer to end programs with an endless loop instruction such as

HERE BRA HERE

Some 6809-based microcomputers require a JMP or JSR instruction with a specific destination address to return control to the monitor. You should consult the User's Manual for your microcomputer to determine the required memory addresses and terminating instruction for your particular system.

- **Use of base page direct memory addressing.** This makes the object code program even shorter and therefore easier to key into memory for testing

Trying the Examples

To test an example program on your microcomputer system, first place the object program in memory. Your assembler program may do this automatically, or it may create an object code file which a separate loader program must then place in memory. Many of the example programs are so short that you can bypass the assembler and simply key the object code into memory using your monitor facility or front panel. Be sure to make any changes your system requires before entering the code; as we mentioned earlier, you may have to change addresses in the program or the terminating instruction.

Once the program is in memory, put the test data in the appropriate locations. Then run the program. After the program terminates, examine the result locations. To test different sets of data, simply change the appropriate data locations before running the program again.

GUIDELINES FOR SOLVING PROBLEMS

Use the following guidelines in solving the problems at the end of each chapter.

1. **Comment each program so that others can understand it.** The comments may be brief and ungrammatical; they should explain the purpose of an instruction or a section of the program. Comments should not describe the operation of instructions; that description is available in manuals. You do not have to comment each statement or explain the obvious. You may follow the format of the examples but provide less detail.
2. **Emphasize clarity, simplicity, and good structure in programs.** While programs should be efficient, do not worry about saving a single byte of program memory or a few microseconds.
3. **Make programs reasonably general.** Do not confuse parameters (such as the number of elements in an array) with fixed constants (such as π or ASCII C).
4. **Load initial values for parameters from the memory area assigned for tem-**

porary storage. Remember that microprocessor applications programs will often execute from ROM or from protected RAM, so you will not be able to vary parameters that are assigned values in the program. The more parameters you can vary, the more likely the program is to be useful in a wide range of tasks.

5. Use **assembler notation** as shown in the examples and defined in Chapter 3.
6. Use **hexadecimal notation for addresses.** Use the **clearest possible form for data.**
7. **If your microcomputer allows it, start all programs in memory address 0000 and use memory addresses starting with 0040₁₆ for data and temporary storage.** Otherwise, establish equivalent addresses for your microcomputer and use them consistently. Again, consult your user's manual.
8. Use **meaningful names for labels and variables** — for example, **SUM** or **CHECK** rather than **X, Y, or Z.**
9. **Execute each program on your microcomputer.** This is ultimately the only way to verify that the program functions correctly. We have provided sample data with each problem, but be sure that the program works for all special cases.

FURTHER PROGRAMMING TIPS

We will now summarize some useful information that will help you in writing your first programs.

Accumulator Operations

Almost all processing instructions (for example, Add, Subtract, AND, OR) use the contents of an accumulator as one operand and place the result back in the same accumulator. In most cases, you will load the initial data into an accumulator with LDA or LDB. You will then store the result (from the same accumulator) with STA or STB.

The Direct Page (Base Page)

You can place data and addresses that you plan to use frequently on the direct (base) page — that is, the page that the processor can access using the Direct Page register. You can then utilize the short direct addressing mode, using one-byte addresses, to reach that data. We assume in our examples that the direct page is page zero, although you can change it easily enough. Remember, however, that the processor initializes the Direct Page register to zero on machine reset, and the assembler assumes the direct page to be page zero unless a SETDP pseudo-operation changes this assumption explicitly.

6809 direct page addressing is a powerful programming tool. Instruction forms with this addressing mode have shorter object codes and execute faster than those using other memory addressing modes, and you can place the direct page anywhere in memory (along 256-byte boundaries). **However, there are disadvantages to using the direct page.** Direct page addressing is a type of absolute addressing; thus programs which use it are limited since the addresses are fixed in the object code. Furthermore, changing the direct page register in a program introduces a new source of potential

errors. The more complex the program or system, the more likely it is that program execution might unexpectedly branch or return to a sequence that assumes the wrong direct page. The 6809 designers intended the direct page to be a tool for program optimization and operating system organization, and discourage its casual use in applications programs. (See the References section of Chapter 3 for further information.)

Memory Operations

Some instructions — shifts, clear, increment (add 1), decrement (subtract 1), and ones or twos complement — can act directly on data in memory. Such instructions allow you to bypass the user registers, but each executes more slowly than the equivalent instruction that acts on a register. A memory operation is slower because the CPU must load the data into a temporary register, perform the operation, and then store the result back into memory. Therefore a sequence of operations on one memory location will execute more slowly than the sequence which operates on the same data in a register, even though the latter sequence must be two instructions (a load register and a store back to memory) longer. Of course, for a single operation the one instruction that operates directly on memory executes faster than the three instructions (load, operate, store) required to obtain the same result through a register operation. Thus, **operating directly on memory is slower than register operation unless the register load and store overhead eliminates the time savings resulting from register use.**

1. The first part of the document discusses the importance of maintaining accurate records of all transactions and activities. It emphasizes the need for transparency and accountability in financial reporting.

2. The second part of the document outlines the various methods and techniques used to collect and analyze data. It includes a detailed description of the experimental procedures and the statistical analysis performed.

3. The third part of the document presents the results of the study, including a comparison of the different methods and techniques used. It discusses the strengths and weaknesses of each method and provides a summary of the findings.

4. The fourth part of the document discusses the implications of the study and provides recommendations for future research. It highlights the need for further investigation into the effectiveness of the different methods and techniques used.

4

Beginning Programs

This chapter contains some very elementary programs. They will introduce some fundamental features of the 6809. In addition, these programs demonstrate some primitive tasks that are common to assembly language programs for many different applications.

PROGRAM EXAMPLES

4-1. 8-BIT DATA TRANSFER

Purpose: Move the contents of memory location 0040 to memory location 0041.

Sample Problem:

(0040) = 6A
Result: (0041) = 6A

Program 4-1:

0000	96	40	LDA	\$40	GET DATA
0002	97	41	STA	\$41	TRANSFER TO NEW LOCATION
0004	3F		SWI		

LDA (Load Accumulator A) and STA (Store Accumulator A) both need an address to determine the memory location that the processor will use in loading or storing the data. In the example, we have used addresses on the direct page (or base page). Remember that we are assuming the Direct Page register contains zero, so all addresses

4-2 6809 Assembly Language Programming

with zeros in their eight most significant bits are on the direct page. Therefore, we can use the direct (or base page) forms of LDA and STA in which the instructions need only specify the eight least significant bits of the memory address in the byte following the operation code. We can omit the leading zeros just as we do in everyday conversation (e.g., we say "sixty cents" rather than "zero dollars and sixty cents"). However, remember that the addresses are really 0040_{16} and 0041_{16} .

Before you execute the example program, you will have to load the data into memory location 0040_{16} . After you execute the program, you can see the result in memory location 0041_{16} (or in Accumulator A — why?).

We use SWI (Software Interrupt) to end all examples and return control to the monitor. You may have to replace this instruction with whatever your microcomputer requires.

4-2. 8-BIT ADDITION

Purpose: Add the contents of memory locations 0040 and 0041, and place the result in memory location 0042.

Sample Problem:

```
(0040) = 38
(0041) = 2B
Result: (0042) = 63
```

Program 4-2:

```
0000 96 40      LDA  $40      GET FIRST OPERAND
0002 9B 41      ADDA $41      ADD SECOND OPERAND
0004 97 42      STA  $42      STORE RESULT
0006 3F         SWI
```

This program uses the direct (base page) forms of LDA, ADDA, and STA, since we have placed all the addresses on the direct page. We will use direct page addressing throughout this book, in order to make the example programs shorter and thus easier to key into memory by hand.

ADDA affects the Carry flag, but LDA and STA do not. Only arithmetic and shift instructions affect the Carry; logical and transfer instructions do not.

LDA and ADDA do not affect the contents of memory, but they do affect the contents of Accumulator A. On the other hand, STA changes the contents of the addressed memory location, but does not affect the contents of Accumulator A.

Before you execute this example program, you will have to load the two operands into memory locations 0040_{16} and 0041_{16} . After you execute the program, you can see the result in memory location 0042_{16} . In a real application, some previous section of the program would store the data in memory and a subsequent section would use the result.

4-3. SHIFT LEFT 1 BIT

Purpose: Shift the contents of memory location 0040 left one bit and place the result in memory location 0041. Clear bit position 0.

Sample Problem:

```
(0040) = 6F = 0110 11112
Result: (0041) = DE = 1101 11102
```

Program 4-3:

```

0000 D6 40          LDB  $40      GET DATA
0002 58           ASLB           SHIFT LEFT
0003 D7 41          STB  $41      STORE RESULT
0005 3F           SWI

```

Unlike the two previous programs, this one uses Accumulator B. There is no compelling reason for these preferences; we could use Accumulator A in this program, and we could use Accumulator B in either of the previous programs. **Accumulators A and B are virtually interchangeable; most instructions can use either one.** We will note a few differences in later chapters.

ASLB shifts Accumulator B left one bit and clears the least significant bit position (bit 0). The previous contents of bit position 7 go into the Carry flag. The result (including the Carry flag) is twice the original data (why?).

We could also shift the contents of memory location 0040 left one bit with the instruction ASL \$40 and then move the result to memory location 0041. However, this method would change the contents of memory location 0040 as well as the contents of memory location 0041. How would you change the program to operate on a memory location without changing the contents of location 0040?

Compare the bit patterns for instructions that use Accumulator A with those that use Accumulator B. How do the bit patterns differ?¹ How does the processor know whether to use Accumulator A or Accumulator B? Remember that two groups of instructions use an accumulator: single-operand instructions such as shifts, clear, increment, and decrement; and double-operand instructions such as ADD, AND, and SUB.

4-4. MASK OFF MOST SIGNIFICANT FOUR BITS

Purpose: Place the least significant four bits of memory location 0040 in the least significant four bits of memory location 0041. Clear the most significant four bits of memory location 0041.

Sample Problem:

```

(0040) = 3D = 0011 11012
Result: (0041) = 0D = 0000 11012

```

Program 4-4:

```

0000 96 40          LDA  $40      GET DATA
0002 84 0F          ANDA #00001111 MASK OUT FOUR MSB'S
0004 97 41          STA  $41      STORE RESULT
0006 3F           SWI

```

The symbol # identifies an immediate operand, and % means binary constant in standard 6809 assembler notation.

ANDA #00001111 logically ANDs the contents of Accumulator A with the binary number 00001111 ($0F_{16}$), not the contents of memory location 000F. Immediate addressing (indicated by # in the operand field) means that the instruction contains the actual data, not its address.

We have written the mask (00001111) in binary to make its purpose clearer to the reader. **Binary masks are easier to understand than hexadecimal ones since the microprocessor performs logical operations bit-by-bit rather than on digits or bytes.** The result, of course, does not depend on the programming notation. **You should use hexadecimal notation for long masks** whenever the binary versions become cumbersome. **The comments should then explain the purpose of the masking operation.**

4-4 6809 Assembly Language Programming

A logical AND instruction may be used to clear bits that are not meaningful. For example, the four least significant bits of the data could be an input from a ten-position switch or an output to a numeric display. Remember that **logically ANDing a bit with '0' always produces a zero result, while logically ANDing a bit with '1' does not change its value.**

4-5. CLEAR A MEMORY LOCATION

Purpose: Clear memory location 0040; that is, reset all the bits in location 0040 to zeros.

Program 4-5:

```
0000 0F 40          CLR  $40      CLEAR MEMORY LOCATION 0040
0002 3F             SWI
```

The CLR instruction can act directly on a memory location, without the need for a user register. Of course, the processor does not really clear the memory location directly; instead, it generates a zero internally (using a register that the programmer cannot access) and writes it into the specified memory location.

CLR always affects the status flags in the same way: it resets the Carry, Sign (Negative), and Overflow flags, and sets the Zero flag.

The 6809 instruction set treats zero as a special number; no other value can be loaded into a memory location as easily.

4-6. BYTE DISASSEMBLY

Purpose: Divide the contents of memory location 0040 into two 4-bit sections (sometimes called “nibbles” or “nybbles”) and place the sections in the low-order four bits of memory locations 0041 and 0042. Place the four most significant bits of 0040 in 0041 and the four least significant bits of 0040 in 0042. Clear the four most significant bits of both 0041 and 0042.

Sample Problem:

```
(0040) = 3F
Result: (0041) = 03
        (0042) = 0F
```

Program 4-6:

```
0000 96 40          LDA  $40      GET DATA
0002 84 0F          ANDA  #%00001111  MASK OFF MSB'S
0004 97 42          STA  $42      STORE LSB'S
0006 96 40          LDA  $40      RELOAD DATA
0008 44             LSR   LSR   SHIFT MSB'S TO LEAST
0009 44             LSR   LSR   SIGNIFICANT POSITIONS
000A 44             LSR   LSR   AND CLEAR OTHER
000B 44             LSR   LSR   POSITIONS
000C 97 41          STA  $41      STORE MSB'S
000E 3F             SWI
```

Each execution of LSR shifts an accumulator or memory location right one position, so four LSRs are required to shift four positions. LSR always clears the most significant bit of the result (a so-called “logical shift”), so four LSRAs clear the four most significant bits of Accumulator A.

Rewrite the program so that it saves a copy of the data in Accumulator B rather than loading it twice. Use the instruction TFR A,B. This instruction moves the contents of A to B without changing A. Which version do you prefer, and why?

The monitor program in your microcomputer must contain a routine similar to this example if it prints or displays the contents of memory locations in hexadecimal. The output device must receive the two hexadecimal digits separately in order to print or display them separately.

4-7. FIND LARGER OF TWO NUMBERS

Purpose: Place the larger of the contents of memory locations 0040 and 0041 in memory location 0042. Assume that memory locations 0040 and 0041 contain unsigned binary numbers.

Sample Problems:

- a. (0040) = 3F
 (0041) = 2B
 Result: (0042) = 3F
- b. (0040) = 75
 (0041) = A8
 Result: (0042) = A8

Program 4-7:

```

0000 96 40          LDA  $40          GET FIRST OPERAND
0002 91 41          CMPA $41          IS SECOND OPERAND LARGER?
0004 24 02          BHS  STRES
0005 96 41          LDA  $41          YES, GET SECOND OPERAND
0008 97 42          STRES STA  $42          STORE LARGER OPERAND
000A 3F             SWI

```

The Compare Instruction and Status Flags

CMPA \$41 subtracts the contents of memory location 0041 from the contents of Accumulator A, but does not save the result anywhere. **All the CMPA instruction does is set the flags for branching; it leaves the value in Accumulator A unchanged**, so that value can be used for later comparisons or other operations.

CMPA affects the flags as follows:

1. The Carry flag (C) is set to 1 if the unsigned subtraction requires a borrow and to 0 if it does not.
2. The Zero flag (Z) is set to 1 if the result of the subtraction is zero and to 0 if it is not.
3. The Sign flag (N) takes the value of the most significant bit of the result of the subtraction.
4. The Overflow flag (V) is set to 1 if the subtraction causes twos complement overflow and to 0 if it does not.

The following cases are particularly important since they are often used for branching:

1. $Z = 1$ if the operands are equal; $Z = 0$ if the operands are not equal. Thus you can use **BEQ** or **BNE** after a **CMP** instruction to check for equality.
2. $C = 1$ if the contents of the memory location are larger (in the unsigned sense) than the contents of the accumulator; $C = 0$ if the contents of the memory location are smaller than or equal to the contents of the accumulator. Remember that **CMPA** calculates $(A) - (M)$, where M is the selected memory location. A borrow is necessary if (M) is larger.

Thus you can use **BLO**, **BHI**, **BLS**, or **BHS** after a **CMP** instruction to compare the magnitude of unsigned numbers. There are four branches so that you can put the equality case on either side; that is, the options are:

- a. $(A) > (M)$ **BHI**, branch if (A) is higher (greater than (M)).
- b. $(A) \geq (M)$ **BHS** (**BCC**), branch if (A) is higher or same (greater than or equal to (M)).
- c. $(A) \leq (M)$ **BLS**, branch if (A) is lower or same (less than or equal to (M)).
- d. $(A) < (M)$ **BLO** (**BCS**), branch if (A) is lower (less than (M)).

Calculating Relative Offsets

All 6809 conditional branch instructions use relative addressing; in this mode, the destination is specified by how far it is from the current instruction. In the short form, the second byte is an 8-bit two's complement number with a range of -128 ($1000\ 0000_2$) to $+127$ ($0111\ 1111_2$). The processor adds this number to the program counter to calculate the destination; the result is

$$\text{NEW PC} = \text{OLD PC} + \text{OFFSET} + 2$$

where **OLD PC** is the original value of the program counter and the extra 2 comes from the two bytes occupied by the branch instruction itself. Rearranging, we can calculate the offset from the equation

$$\text{OFFSET} = \text{NEW PC} - \text{OLD PC} - 2$$

In our latest object program, for example, we have

$$\begin{aligned} \text{OLD PC} &= 0004 \\ \text{NEW PC (destination)} &= 0008 \end{aligned}$$

So

$$\text{OFFSET} = 0008 - 0004 - 2 = 02$$

You can always get the same result by counting bytes. Start counting at 0 at the byte immediately following the last byte of the branch instruction.

Calculating offsets is clearly a rather unpleasant task, unless you are very good at binary or hexadecimal arithmetic or own a calculator (such as the Texas Instruments Programmer) that performs arithmetic in different number systems. The calculations are particularly troublesome if the branch is backwards — that is, the destination address is smaller than the original program counter value plus two. Then you must deal with negative binary or hexadecimal numbers: FF_{16} is -1 , FE_{16} is -2 , and so on. Counting bytes is very tedious, especially for long offsets.

The way to avoid calculating offsets is to let the assembler do it. You can, for example, simply specify how far you want the branch to go by using an expression containing the symbol *****, which refers the assembler to the current value of the location counter. Thus

BHS *+4

will produce a branch to the instruction four bytes further along. The assembler will take care of the extra 2 automatically (that is, it will make the actual offset 2 instead of 4). The problem with this approach is that 6809 instructions vary in length and thus it is often difficult to determine the required numerical value. Furthermore, a much better method of specifying offsets is available.

The better method is to assign a name (referred to as a "label") to the destination address. You can choose whatever name you want (see Chapter 2), but we will try to choose names that have some mnemonic value. The assembler will determine the actual address to which the label refers and will calculate offsets for any branches that use the label. The use of labels not only makes the programmer's job easier, but it also makes programs easier to read and understand.

Conditional Branches

Conditional branches work as follows:

1. If the condition is true, the processor branches. That is, it places the destination address in the program counter and starts executing instructions at that point.
2. If the condition is false, the processor continues its normal sequence as if the branch instruction did nothing at all except advance the program counter.

In our latest source program, the choices are:

1. If $(A) \geq (0041)$, $NEW\ PC = OLD\ PC + OFFSET + 2 = 0004 + 02 + 2 = 0008$ (We have named this location with the label STRES.)
2. If $(A) < (0041)$, $NEW\ PC = OLD\ PC + 2 = 0004 + 2 = 0006$ (The location immediately following the branch instruction.)

Executing a 2-byte instruction advances the program counter by 2 regardless of whether a branch occurs.

BHS causes a branch if $(A) \geq (M)$. In terms of the flags, the branch condition is $C = 0$, meaning $(A) \geq (M)$.

4-8. 16-BIT ADDITION

Purpose: Add the 16-bit number in memory locations 0040 and 0041 to the 16-bit number in memory locations 0042 and 0043. The most significant bytes are in memory locations 0040 and 0042. Store the result in memory locations 0044 and 0045, with the most significant byte in 0044.

Sample Problem:

```
(0040) = 67
(0041) = 2A
(0042) = 14
(0043) = F8
Result: 672A + 14F8 = 7C22
(0044) = 7C
(0045) = 22
```

Program 4-8:

```
0000 DC 40          LDD  $40      GET FIRST 16-BIT NUMBER
0002 D3 42          ADDD $42      ADD SECOND 16-BIT NUMBER
0004 DD 44          STD  $44      STORE 16-BIT RESULT
0006 3F             SWI
```

The Double Accumulator D consists of Accumulator A, which comprises the high-order byte, and Accumulator B, used as the low-order byte. Be careful — D is not a separate register; it is physically the same as A and B.

The 16-bit operations LDD, ADDD, and STD all operate on two bytes of data. For example, LDD \$40 loads the contents of memory location 0040 into Accumulator A and the contents of memory location 0041 into Accumulator B. ADDD \$42 adds the contents of memory location 0043 to Accumulator B and then adds the Carry from that operation and the contents of memory location 0042 to Accumulator A. STD \$44 stores the contents of Accumulator A in memory location 0044 and the contents of Accumulator B in memory location 0045.

The 6809 microprocessor actually performs most 16-bit operations eight bits (one byte) at a time. The advantages of the 16-bit instructions are that they direct the processor through two 8-bit operations instead of one, thus reducing the amount of time spent fetching instructions as well as the amount of program memory that is required.

Remember that 16-bit data (and 16-bit addresses) always occupy two bytes of memory, the one that is actually addressed and the next higher one. For example, LDD \$40 uses memory location 0041 as well as 0040.

The 6809 convention for storing 16-bit data (and 16-bit addresses) is to store the eight most significant bits first (at the lower address). This convention seems natural, but is the opposite of that used in most other microprocessors and minicomputers.

4-9. TABLE OF SQUARES

Purpose: Calculate the square of the contents of memory location 0041 from a table and place the square in memory location 0042. Assume that memory location 0041 contains a number between 0 and 7 inclusive; that is, $0 \leq (0041) \leq 7$. The table occupies memory locations 0050 through 0057.

Hexadecimal Memory Address	Entry	
	Hexadecimal	Decimal
0050	00	0 (0 ²)
0051	01	1 (1 ²)
0052	04	4 (2 ²)
0053	09	9 (3 ²)
0054	10	16 (4 ²)
0055	19	25 (5 ²)
0056	24	36 (6 ²)
0057	31	49 (7 ²)

Sample Problems:

- a. (0041) = 03
 Result: (0042) = 09
- b. (0041) = 06
 Result: (0042) = 24

Remember that the answer is a hexadecimal number.

Program 4-9:

```

0000 D6 41          LDB  $41      GET DATA
0002 8E 0050       LDX  #$50     GET BASE ADDRESS
0005 A6 85         LDA  B,X      GET SQUARE OF DATA
0007 97 42         STA  $42     STORE SQUARE
0009 3F           SWI

0050              ORG  $50      TABLE OF SQUARES
0050          00   SQTAB FCB  0,1,4,9;16,25,36,49
0051          01
0052          04
0053          09
0054          10
0055          19
0056          24
0057          31

```

The assembler directive FCB places the table of squares in memory locations 0050 through 0057. This block of data is essential for the proper execution of the program, even though it does not consist of instructions. The object program may thus include fixed data as well as executable instructions.

LDX # \$50 loads Index Register X from the two bytes of memory immediately following the operation code (addresses 0003 and 0004 in the object program). The processor loads the contents of the first byte into the eight most significant bits of Index Register X, and the contents of the second byte into the eight least significant bits of Index Register X. Always remember that Index Registers X and Y, Stack Pointers S and U, and the Double Accumulator D are all 16 bits long.

Indexed Addressing

The instruction LDA B,X loads Accumulator A from the address calculated by adding the contents of Index Register X (the “base address” of the table) and the contents of Accumulator B (the index of the element that we want). For example, if memory location 0041 contains 03, then

```

(X) = 0050 (base address of the table of squares)
(B) = 03 (data)

```

The calculated or “effective” address is

$$EA = (X) + (B) = 0053$$

Address 0053 contains the square of 3. The result of this procedure (called a “table lookup”) depends only on the organization of the table; it does not depend on the table data value or on the function that the table represents.

As we discussed in Chapter 3, all indexed instructions require an extra object code byte, called the “post byte,” which selects from among the indexed addressing modes. Our example uses the non-indirect mode with an offset in Accumulator B from the indexable register R (referred to as accumulator indexed addressing). The binary form is:

```
1 R R 0 0 1 0 1
```

We have chosen Index Register X, so RR = 00. See Table 3-4 and Appendix B for a complete description of the indexed addressing modes and the assignment of bits in the post bytes.

Indexing takes extra clock cycles whenever the processor must calculate the effective address. Adding Accumulator B to Index Register X takes one cycle beyond the base amount required by any indexed instruction (four cycles for LDA). Appendix B tells how many extra bytes of memory and extra clock cycles each of the indexed modes requires.

Operations on Registers X, Y, S, and U

The 6809 has a few special instructions that operate on the index registers and stack pointers rather than on the accumulators. These are:

CMP(X/Y/S/U)	—	Compare Memory with Index Register or Stack Pointer
LD(X/Y/S/U)	—	Load Memory into Index Register or Stack Pointer
LEA(X/Y/S/U)	—	Load Effective Address into Index Register or Stack Pointer
ST(X/Y/S/U)	—	Store Index Register or Stack Pointer in Memory

The index registers and stack pointers are primarily intended to hold memory addresses, so there are no logical or arithmetic instructions for those registers. As we will see, however, you can occasionally use LEA to perform some arithmetic.

Use of the ORIGIN Directive

The assembler directive ORG simply determines where the loader program will place the next section of code when it is finally entered into the microcomputer's memory for execution. An ORG does not actually result in the generation of any object code.

Arithmetic with Tables

The use of lookup tables is a simple but powerful approach to solving complex arithmetic problems on microprocessors. The lookup table contains all the possible answers to a problem, much as a table of sines or cosines contains all the possible values of a particular function. This approach reduces an arithmetic problem to a problem of obtaining the correct answer from the table. To do that, we need two things: the base (starting) address of the table and the position (called the "index") of the answer. The address of the answer is the sum of the base address and the index.

The base address of a table is a fixed number. The index, however, is not, and we need some way to determine it. In simple cases (such as our Table of Squares example), we can organize the table so that the data itself is the index. In the example, the zeroth entry in the table is zero squared, the first entry is one squared, and so on. In more complex cases, where the input values are irregularly spaced or there are several data items involved (for example, roots of a quadratic equation or number of permutations), we must actually perform some computations (perhaps even involving another table) to determine an index from the data.

The use of tables represents tradeoffs among programming time, execution time, and memory usage. A table lookup executes faster than any but the simplest calculations. For example, even the Table of Squares program executes faster than an equivalent simple squaring program using the 6809 multiplication instruction MUL. Tables can be faster and simpler to program than actual calculations since lookup procedures do not depend on the complexity of the function involved. Furthermore, since a table lookup is fast-executing, it is unlikely to slow down a program intolerably, as a complex calculation might, and thus is less likely than a calculation to require reprogramming to save execution time. On the other hand, tables can occupy a large

amount of memory if there are many possible input values. We can often reduce the required amount of memory by limiting the accuracy of the results, scaling the input data, or organizing the table cleverly.

Common uses of tables include the computation of transcendental and trigonometric functions, the linearization of inputs from thermocouples and other non-linear devices, and code conversions.

4-10. 16-BIT ONES COMPLEMENT

Purpose: Place the ones complement of the 16-bit number in memory locations 0040 and 0041 in memory locations 0042 and 0043. The most significant bytes are in locations 0040 and 0042.

Sample Problem:

```

(0040) = 67 }
(0041) = E2 } 0110 0111 1110 00102
Result: (0042) = 98 }
(0043) = 1D } 1001 1000 0001 11012

```

The ones complement of a number is its logical inverse; that is, each 0 bit in the number is replaced by a 1 and each 1 bit by a 0. The sum of a number and its ones complement is therefore always a number in which all the bit positions contain 1s.

Program 4-10:

```

0000 DC 40          LDD   $40          GET 16-BIT NUMBER
0002 43             COMA          ONES COMPLEMENT MSB'S
0003 53             COMB          ONES COMPLEMENT LSB'S
0004 DD 42          STD   $42          STORE 16-BIT ONES COMPLEMENT
0006 3F             SWI

```

Despite the 6809's 16-bit instructions, you must use the 8-bit instructions to perform many arithmetic and logical operations. The 6809 instruction set does include some common 16-bit operations, such as loading, adding, comparing, subtracting, and storing, but other operations must be performed eight bits at a time.

Manage the accumulators with care; they can hold only one result at a time. If you need an accumulator's contents, be sure to save them before reloading the accumulator.

PROBLEMS

4-1. 16-BIT DATA TRANSFER

Purpose: Move the contents of memory location 0040 to memory location 0042 and the contents of memory location 0041 to memory location 0043.

Sample Problem:

```

(0040) = 3E
(0041) = B7
Result: (0042) = 3E
(0043) = B7

```

4-2. 8-BIT SUBTRACTION

Purpose: Subtract the contents of memory location 0041 from the contents of memory location 0040. Place the result in memory location 0042.

Sample Problem:

(0040) = 77
(0041) = 39
Result: (0042) = 3E

4-3. SHIFT LEFT TWO BITS

Purpose: Shift the contents of memory location 0040 left two bits and place the result in memory location 0041. Clear the two least significant bit positions.

Sample Problem:

(0040) = 5D = 0101 1101₂
Result: (0041) = 74 = 0111 0100₂

4-4. MASK OFF LEAST SIGNIFICANT FOUR BITS

Purpose: Place the four most significant bits of memory location 0040 in memory location 0041. Clear the four least significant bits of memory location 0041.

Sample Problem:

(0040) = C4 = 1100 0100₂
Result: (0041) = C0 = 1100 0000₂

4-5. SET A MEMORY LOCATION TO ALL ONES

Purpose: Set all the bits of memory location 0040 to ones (FF₁₆).

4-6. BYTE ASSEMBLY

Purpose: Combine the four least significant bits of memory locations 0040 and 0041 into a byte and store the result in memory location 0042. Place the four least significant bits of memory location 0040 in the four most significant bit positions of memory location 0042; place the four least significant bits of memory location 0041 in the four least significant bit positions of memory location 0042.

Sample Problem:

(0040) = 6A = 0110 1010₂
(0041) = B3 = 1011 0011₂
Result: (0042) = A3 = 1010 0011₂

4-7. FIND SMALLER OF TWO NUMBERS

Purpose: Place the smaller of the contents of memory locations 0040 and 0041 in memory location 0042. Assume that memory locations 0040 and 0041 contain unsigned binary numbers.

Sample Problems:

a. (0040) = 3F
 (0041) = 2B
 Result: (0042) = 2B

b. (0040) = 75
 (0041) = A8
 Result: (0042) = 75

4-8. 24-BIT ADDITION

Purpose: Add the 24-bit number in memory locations 0040, 0041, and 0042 to the 24-bit number in memory locations 0043, 0044, and 0045. The most significant bytes are in memory locations 0040 and 0043, the least significant bytes in memory locations 0042 and 0045. Store the result in memory locations 0046, 0047, and 0048 with the most significant byte in memory location 0046 and the least significant byte in 0048.

Sample Problem:

	(0040) = 35	}	
	(0041) = 67		35672A
	(0042) = 2A		
	(0043) = 51	}	
	(0044) = A4		51A4F8
	(0045) = F8		
Result:	(0046) = 87	}	
	(0047) = 0C		870C22
	(0048) = 22		

4-9. SUM OF SQUARES

Purpose: Calculate the squares of the contents of memory locations 0040 and 0041 and add them together. Place the result in memory location 0042. Assume that memory locations 0040 and 0041 both contain numbers between 0 and 7 inclusive; that is, $0 \leq (0040) \leq 7$ and $0 \leq (0041) \leq 7$. Use the table of squares from the example entitled Table of Squares.

Sample Problem:

	(0040) = 03
	(0041) = 06
Result:	(0042) = 2D

that is, $3^2 + 6^2 = 9 + 36_{10} = 45_{10} = 2D_{16}$

4-10. 16-BIT TWOS COMPLEMENT

Purpose: Place the two's complement of the 16-bit number in memory locations 0040 and 0041 (most significant bits in 0040) in memory locations 0042 and 0043 (most significant bits in 0042). The two's complement of a number is the number that, when added to the original number, produces a result of zero; the two's complement is also equal to the one's complement plus one, since the sum of a number and its one's complement is the all ones word.

Sample Problems:

a.	(0040) = 00	}	0000	0000	0101	1000 ₂
	(0041) = 58	}				
	Result: (0042) = FF	}	1111	1111	1010	1000 ₂
	(0043) = A8	}				
b.	(0040) = 72	}	0111	0010	0000	0000 ₂
	(0041) = 00	}				
	Result: (0042) = 8E	}	1000	1110	0000	0000 ₂
	(0043) = 00	}				

Since the sum of the original number and its two's complement is zero, we can calculate the two's complement of x as $0 - x$. Which approach (calculating the one's complement and adding one, or subtracting from zero) results in a shorter and faster program? Remember to use the SUBD instruction.

REFERENCES

1. L. A. Leventhal, "Microprogramming," *Kilobaud*, April 1977, pp. 120-23.

5

Simple Program Loops

The program loop is the basic structure that forces the CPU to repeat a sequence of instructions. Loops have four sections:

- 1. The initialization section**, which establishes the starting values of counters, pointers, indexes, and other variables.
- 2. The processing section**, where the actual data manipulation occurs. This is the section that does the work.
- 3. The loop control section**, which updates counters and pointers for the next iteration.
- 4. The concluding section**, which analyzes and stores the results.

The computer performs Sections 1 and 4 only once, while it may perform Sections 2 and 3 many times. Therefore, the execution time of the loop depends mainly on the execution time of Sections 2 and 3. Those sections should execute as quickly as possible, while the execution times of Sections 1 and 4 have little effect on overall program speed.

Figures 5-1 and 5-2 contain two alternative flowcharts for a typical program loop. Following the flowchart in Figure 5-1 results in the computer always executing the processing section at least once. On the other hand, the computer may not execute the processing section in Figure 5-2 at all. The order of operations in Figure 5-1 is more natural, but the order in Figure 5-2 is often more efficient and eliminates the problem of the computer going through the processing sequence once even where there is no data for it to handle.

The computer can use the loop structure to process large sets of data (usually called "blocks" or "arrays"). The simplest way to use one sequence of instructions to handle a block of data is to have the program add 1 to its address register (usually

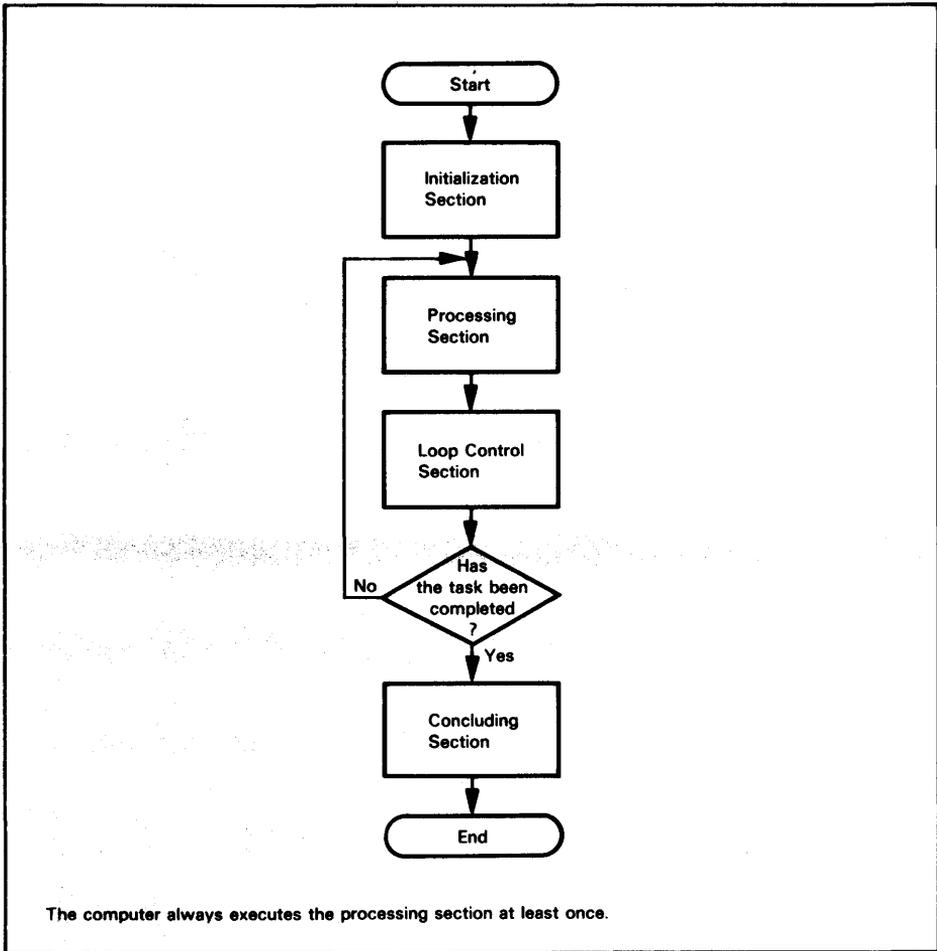


Figure 5-1. Flowchart of a Program Loop

an index register or stack pointer) after each iteration. Then the address register will contain the address of the next element in the block when the computer repeats the sequence of instructions. The computer can then handle blocks of any length with a single program.

Indexed addressing is the key to processing blocks of data with the 6809 microprocessor, since that mode allows you to vary the actual address of the data (the “effective address”) by changing the contents of an address register. In immediate and extended addressing modes, the instruction completely determines the effective address; that address is therefore fixed if program memory is read-only. The direct page mode shares this fixed address limitation even though a register determines part of the effective address.

The 6809’s autoincrementing mode is particularly convenient for processing arrays, since it automatically updates the address register for the next iteration. No

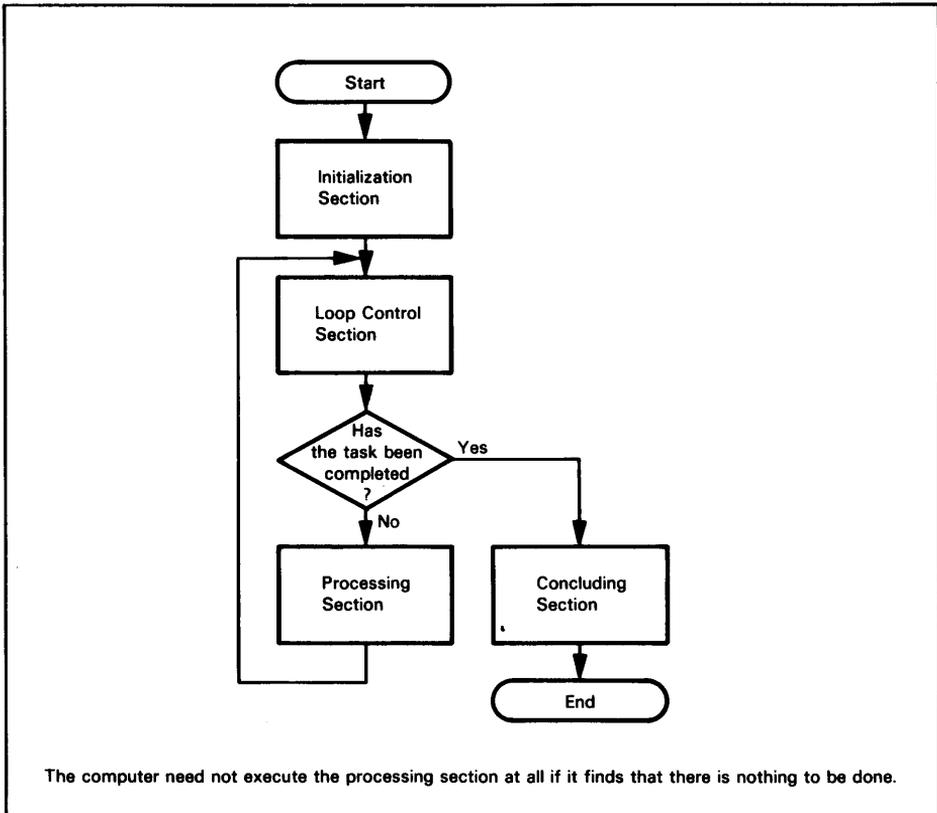


Figure 5-2. An Alternative for a Program Loop

additional instruction is necessary. You can even have an automatic increment by 2 if the array contains 16-bit data or addresses.

Although our examples show the processing of arrays with autoincrementing (adding 1 or 2 after each iteration), **the procedure is equally valid with autodecrementing** (subtracting 1 or 2 before each iteration). Most programmers find moving backwards through an array somewhat awkward and difficult to follow, but it is more efficient in many situations. Clearly, the computer does not know backwards from forward. **The programmer, however, must remember that the 6809 increments an address register after using it but decrements an address register before using it.** This difference affects initialization as follows:

1. When moving forward through an array (autoincrementing), start the address register at the lowest address occupied by the array.
2. When moving backwards through an array (autodecrementing), start the address register one step (1 or 2) beyond the highest address occupied by the array.

You must also remember the difference between autoincrementing and autodecrementing if you use a CMP instruction (CMPX, CMPY, CMPU, or CMPS) to determine if an index register or stack pointer has reached a particular value.

PROGRAM EXAMPLES

5-1. SUM OF DATA

Purpose: Calculate the sum of a series of numbers. The length of the series is in memory location 0041 and the series begins in memory location 0042. Store the sum in memory location 0040. Assume that the sum is an 8-bit number so that you can ignore carries.

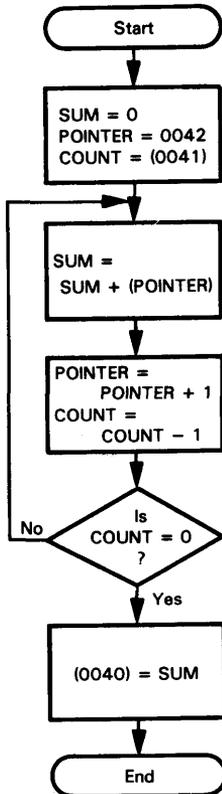
Sample Problem:

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26

Result: (0040) = A3
 = $28_{16} + 55_{16} + 26_{16}$

There are three entries in the sum, since (0041)=03

Flowchart:



(**POINTER**) refers to the contents of the memory location addressed by **POINTER**. Remember that on the 6809 and similar microprocessors, **POINTER** is a 16-bit address, while (**POINTER**) is an 8-bit byte of data.

This flowchart has the same form as that in Figure 5-1; that is, the processing section will execute at least once. What does this form assume about the data, specifically the length of the series (called **COUNT** above)?

Program 5-1a:

0000	4F			CLRA		SUM = ZERO
0001	D6	41		LDB	\$41	COUNT = LENGTH OF ARRAY
0003	8E	0042		LDX	#\$42	POINT TO START OF ARRAY
0006	AB	80	SUMD	ADDA	,X+	ADD NUMBER TO SUM
0008	5A			DECB		
0009	26	FB		BNE	SUMD	
000B	97	40		STA	\$40	
000D	3F			SWI		

The initialization section of the program consists of the first three instructions, which set the sum, counter, and data pointers to their starting values. **LDX** loads the two bytes of memory into Index Register X: 00 and 42 from memory addresses 0004 and 0005 respectively.

The processing section of the program consists of the single instruction **ADDA ,X+** which adds the contents of the memory location addressed by Index Register X to the contents of Accumulator A. This instruction does the real work of the program. The effective address (that is, the address from which the CPU gets the data) is given by the contents of Index Register X.

In the autoincrementing mode, the processor adds 1 to the contents of Index Register X after using it to fetch the data. For example, in the first iteration, Index Register X initially contains 0042. The execution of the instruction **ADDA ,X+** results in the contents of memory location 0042 being added to Accumulator A, and Index Register X being incremented by 1 to 0043.

The loop control section of the program consists of the single instruction **DECB**, since the instruction **ADDA ,X+** updates the pointer automatically. **DECB** decrements the counter that keeps track of how many iterations the computer has left to perform.

The instruction **BNE** causes a branch if the Zero flag is 0 (that is, if the result of decrementing B was *not zero*). The offset is a twos complement number, determined by the distance between the destination and the end of the instruction. In this case, the distance is from memory location 000B (the address following the end of the **BNE** instruction) to memory location 0006 (the destination). So the offset is:

$$\begin{array}{l} 0006 \} \\ -000B \} = \{ \begin{array}{l} 0006 \\ +FFF5 \\ \hline FFFB \end{array} \end{array}$$

The 8-bit offset mode (**BNE** rather than **LBNE**) requires only the two least significant digits of the difference.

If the Zero flag is 1 (that is, if the result of decrementing B was *zero*), the processor continues its normal sequence. Thus the result of executing **BNE** is:

$$(PC) = \begin{cases} \text{SUMD if the result of decrementing B is not zero} \\ (PC) + 2 \text{ if the result of decrementing B is zero} \end{cases}$$

The extra 2, as usual, comes from the two bytes occupied by the BNE instruction itself.

Most programmers make computer loops count down rather than up so that they can use the setting of the Zero flag as an exit condition. Remember that the Zero flag is 1 if the most recent result was zero and 0 if that result was not zero. Rewrite the program so that it loads Accumulator B with zero initially and increments it after each iteration. Which approach is more efficient?

The order in which the processor executes instructions is often very important. DECB must come immediately before BNE SUMD; otherwise, the intervening instruction(s) would probably change the Zero flag. The order of operations within instructions may also be important. In the current program, we must initialize Index Register X to 0042, the lowest address in the array, since the processor increments Index Register X after using its contents in the instruction ADDA ,X+. What initial value would be necessary if the processor incremented Index Register X before using its contents?

Using Register Y

We could easily use Index Register Y, User Stack Pointer U, or Hardware Stack Pointer S instead of Index Register X. The only difference is that LDS and LDY require two-byte operation codes, so a program using one of those registers would occupy one additional byte of memory and would take one extra clock cycle to execute. For example, the following program uses Index Register Y.

Program 5-1b:

0000	4F		CLRA		SUM = ZERO
0001	D6	41	LDB	\$41	COUNT = LENGTH OF ARRAY
0003	108E	0042	LDY	#\$42	POINT TO START OF ARRAY
0007	AB	A0	ADDA	,Y+	ADD NUMBER TO SUM
0009	5A		DECB		
000A	26	FB	BNE	SUMD	
000C	97	40	STA	\$40	
000E	3F		SWI		

In most applications, the slight differences in execution time and memory usage between the two programs do not matter. However, you might as well use Index Register X rather than Index Register Y when both are available, since programs that use Index Register X will be a little shorter and faster. **User Stack Pointer U can also be utilized as an address register, but most programs leave Hardware Stack Pointer S permanently assigned for use with subroutines and interrupts.**

You should verify the hexadecimal value of the relative offset in the last program example. Of course, the final test of any calculation of an offset is whether the program runs correctly. If you must perform hexadecimal calculations frequently, you should consider purchasing either a manual aid or a calculator such as the Texas Instruments Programmer.

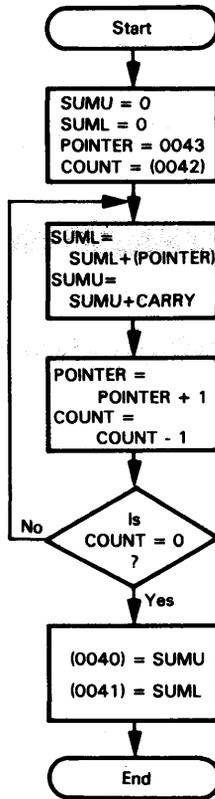
5-2. 16-BIT SUM OF DATA

Purpose: Calculate the sum of a series of 8-bit numbers. The length of the series is in memory location 0042 and the series itself begins in memory location 0043. Store the sum in memory locations 0040 and 0041 (eight most significant bits in 0040).

Sample Problem:

(0042) = 03
 (0043) = C8
 (0044) = FA
 (0045) = 96
 Result: (0040) = 02 } $0258_{16} = C8_{16} + FA_{16} + 96_{16}$
 (0041) = 58 }

Flowchart:



SUMU and SUML are, respectively, the high-order and low-order bytes of the 16-bit sum, SUM.

Program 5-2:

0000 4F		CLRA		MSB'S OF SUM = ZERO
0001 5F		CLRB		LSB'S OF SUM = ZERO
0002 8E	0043	LDX	#\$43	POINT TO START OF ARRAY
0005 EB	80	ADDB	,X+	SUM = SUM + DATA
0007 89	00	ADCA	#0	AND ADD IN CARRY
0009 0A	42	DEC	\$42	
000B 26	F8	BNE	SUMD	
000D DD	40	STD	\$40	SAVE SUM
000F 3F		SWI		

This program has the same structure as the previous example. The only difference is that this program must handle the high-order byte of the sum as well as the low-order byte. The initialization section clears the full 16-bit sum and the processing section now consists of two instructions: `ADDB ,X+` adds the 8-bit data to the low-order byte of the sum and `ADCA #0` adds the carry to the high-order byte.

The only new aspect is that the 16-bit sum occupies both accumulators. Thus we use a memory location on the direct (base) page to hold the counter. Such memory locations are often used as if they were additional registers, since the processor can access them with faster and shorter instructions than those it uses to access other locations.

The instruction `ADCA #0` adds the carry and 0 to Accumulator A:

$$\begin{aligned} (A) &= (A) + 0 + \text{Carry} \\ &= (A) + \text{Carry} \end{aligned}$$

The result is to leave A unchanged if the Carry flag is 0 and to increment A by 1 if the Carry flag is 1.

The 6809 does not have a complete set of 16-bit instructions. For example, there is no Clear Double Accumulator instruction. We can use either the two instructions `CLRA`, `CLRB` (requiring two bytes of memory and four clock cycles) or the immediate instruction `LDD #0` (requiring three bytes of memory and three clock cycles). However, when a 16-bit instruction is available (for instance, `STD $40`), it uses less time and memory than the two equivalent 8-bit instructions (in this case, `STA $40`, `STB $41`).

A single instruction such as `DEC $42` can decrement the contents of a memory location by 1 without changing any registers. Such instructions do affect the flags, however. Note that a memory location is not nearly as useful as an accumulator; there are no instructions that perform general arithmetic or logical operations on data in a memory location. For example, `SUBA #3` subtracts 3 from Accumulator A; try to perform the same operation on the data in memory location 0042.

Long Conditional Branches

Short relative branches are limited to distances that can be specified in an 8-bit signed offset. These limitations are $7F_{16} = 127_{10}$ forward and $80_{16} = -128_{10}$ backwards from the end of the branch instruction. Since short branches are two-byte instructions, the distance from the start of the instruction must be in the range

$$-126_{10} \leq \text{distance} \leq +129_{10}$$

For longer distances, you must use the long form of the branches. A long conditional branch uses the same mnemonic as its short equivalent, with an additional "L" in front: for instance, `LBCC` instead of `BCC`. It requires a two-byte operation code followed by a two-byte relative offset. However, the unconditional branch `LBRA` has a one-byte operation code, although it still requires a two-byte offset. The long relative branches provide access to any memory location in the normal 64K range. In actual practice, most program branches are quite short and you will rarely need the long forms.

5-3. NUMBER OF NEGATIVE ELEMENTS

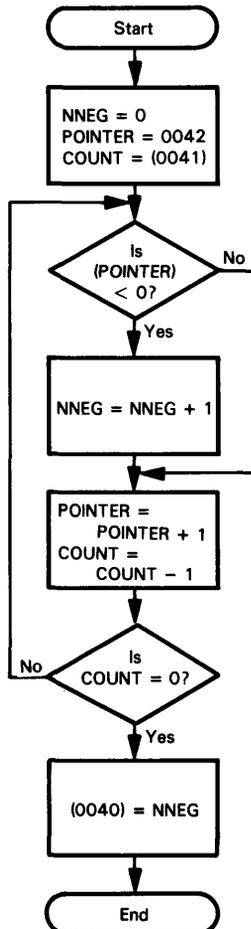
Purpose: Determine the number of negative elements (most significant bit contains 1) in a block. The length of the block is in memory location 0041, and the block itself starts in memory location 0042. Place the number of negative elements in memory location 0040.

Sample Problem:

(0041) = 06
 (0042) = 68
 (0043) = F2
 (0044) = 87
 (0045) = 30
 (0046) = 59
 (0047) = 2A

Result: (0040) = 02, since 0043 and 0044 contain numbers with an MSB of 1

Flowchart:



5-10 6809 Assembly Language Programming

Like the previous flowchart, this one takes the form shown in Figure 5-1; thus it assumes that the input value of COUNT will always be 1 or greater.

Program 5-3:

```
0000 8E 0042          LDX  #542      POINT TO FIRST NUMBER
0003 5F              CLRB          NUMBER OF NEGATIVES = ZERO
0004 A6 80          CHKNEG LDA  ,X+    IS NEXT ELEMENT NEGATIVE?
0006 2A 01          BPL   CHCNT
0008 5C              INCB          YES, ADD 1 TO # OF NEGATIVES
0009 0A 41          CHCNT DEC  $41
000B 26 F7          BNE  CHKNEG
000D D7 40          STB  $40      SAVE NUMBER OF NEGATIVES
000F 3F              SWI
```

LDA affects the Sign (N) and Zero (Z) flags. We can therefore immediately determine if a number that has been loaded into an accumulator is negative or zero.

We could use the Test instruction (TST) to set the Sign flag without using Accumulator A. Accumulator A would then be available to hold a counter. Rewrite the example program to use TST; this instruction is often useful for determining if bit 7 of a memory location is set or if the memory location contains zero.

BPL, Branch if Plus, causes a branch if the Sign flag is 0. The offset for BPL is the distance from the end of the instruction to the destination. Here the distance is a single byte; the result is that the processor skips the INCB instruction if the Sign flag is 0.

The Sign flag simply reflects the value of bit 7 of the most recent result. If you are using signed numbers, bit 7 is, in fact, the sign (0 for positive, 1 for negative); the mnemonics for Branch if Sign = 1 (BMI) and Branch if Sign = 0 (BPL) assume that you are using signed numbers. However, you can equally well use bit 7 for other purposes, such as the status of peripherals or other one-bit data. You can still test bit 7 with BMI or BPL; the mnemonics may no longer make sense, but the operations work. The computer performs its operations without considering whether the user thinks they are sensible or meaningful. The interpretation of the results is the programmer's problem, not the computer's.

Negative signed numbers all have a most significant bit of 1 and thus are actually larger, in the unsigned sense, than positive numbers.

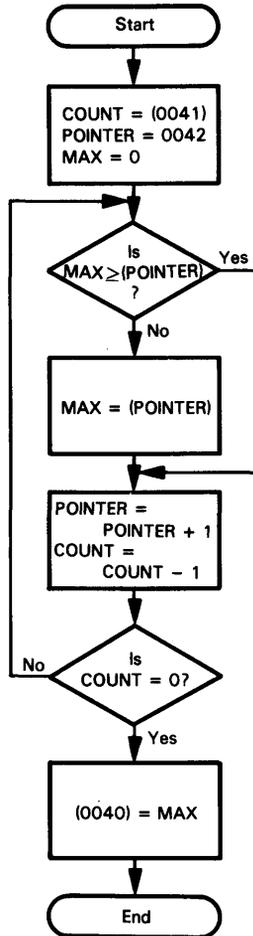
5-4. MAXIMUM VALUE

Purpose: Find the largest element in a block of unsigned binary numbers. The length of the block is in memory location 0041 and the block itself begins in memory location 0042. Store the maximum (largest unsigned element) in memory location 0040.

Sample Problem:

```
(0041) = 05  Number of elements
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72
Result: (0040) = E3, since this is the largest of
          the five unsigned numbers
```

Flowchart:



Program 5-4:

0000 D6	41		LDB	\$41	COUNT = NUMBER OF ELEMENTS
0002 4F			CLRA		MAX = 0 (MINIMUM POSSIBLE)
0003 8E	0042		LDX	#\$42	POINT TO FIRST ENTRY
0005 A1	80	MAXM	CMPA	,X+	IS CURRENT ENTRY GREATER
		*			THAN MAX?
0008 24	02		BHS	NOCHG	
000A A6	1F		LDA	-1,X	YES, REPLACE MAX WITH
		*			CURRENT ENTRY
000C 5A		NOCHG	DECB		
000D 26	F7		BNE	MAXM	
000F 97	40		STA	\$40	SAVE MAXIMUM
0011 3F			SWI		

The first three instructions of this program form the initialization section.

This program takes advantage of the fact that zero is the smallest unsigned binary number. If you make zero the initial estimate of the maximum, the program will set the maximum to a larger value unless all the elements in the array are zeros.

5-12 6809 Assembly Language Programming

The instruction `LDA -1,X` uses the indexed addressing mode with a constant offset. The offset of `-1` is necessary because the autoincrementing in `CMPA ,X+` has added 1 to Index Register X. The object code uses the special 5-bit offset form (signified by a 0 in bit 7 of the post byte). In this form, the offset is a twos complement number in the five least significant bits; bit 4 is thus the sign of the offset, and the processor automatically extends (copies) that bit into the more significant positions before performing the addition. The processor thus extends 11111_2 to $1111\ 1111_2$, an 8-bit number. This form requires no additional bytes of memory (since the post byte contains the offset) and only one additional clock cycle. The range of the offset is

$$-16_{10} = 10000_2 \leq \text{Offset} \leq +15_{10} = 01111_2$$

The relative offsets in the branch instructions are:

1. BHS NOCHG		
Destination address	=	000C
- Address at end of instruction	=	<u>000A</u>
		02
2. BNE MAXM		
Destination address	=	0006 = 0006
- Address at end of instruction	=	000F + <u>FFF1</u>
		F7

The program works correctly if the array has two elements, but not if it has only one element or none at all. Why? How could you eliminate this problem?

The instruction `CMPA ,X+` affects the Carry flag as follows (ELEMENT is the contents of the effective address and MAX is the contents of Accumulator A):

$$\begin{aligned} \text{Carry} &= 0 && \text{if } \text{MAX} \geq \text{ELEMENT} \text{ ("Higher or Same")} \\ \text{Carry} &= 1 && \text{if } \text{MAX} < \text{ELEMENT} \text{ ("Lower")} \end{aligned}$$

If $\text{Carry} = 0$, the program branches to address NOCHG and does not replace the current maximum. If $\text{Carry} = 1$, the program replaces the maximum with the current element using the instruction `LDA -1,X`.

The program does not work properly if the numbers are signed, because negative numbers all appear to be larger than positive numbers. You must then use the Sign (Negative) flag instead of the Carry in the comparison. However, you must also consider the fact that twos complement overflow can affect the sign; that is, the magnitude of a signed result could overflow into the sign bit. **The 6809 has special branch instructions — BGT, BGE, BLE, and BLT — which perform the branches indicated by their mnemonics after signed comparisons and handle twos complement overflow automatically.**

5-5. JUSTIFY A BINARY FRACTION

Purpose: Shift the contents of memory location 0040 until the most significant bit of the number is 1. Store the result in memory location 0041 and the number of left shifts required in memory location 0042. If the contents of memory location 0040 are 0, clear both 0041 and 0042.

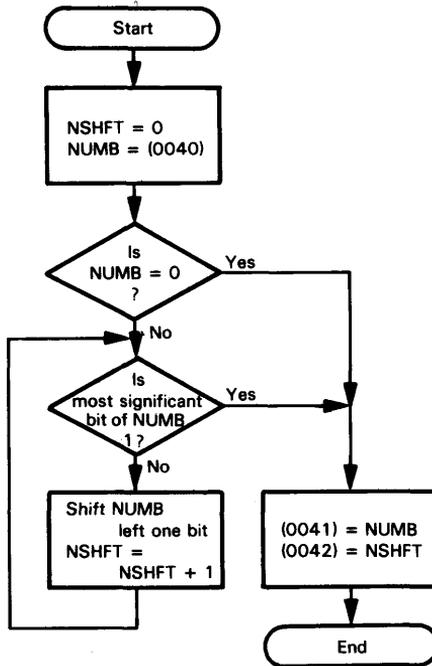
The process is just like converting a number to a scientific notation; for example:

$$0.0057 = 5.7 \times 10^{-3}$$

Sample Problems:

- a. (0040) = 22
Result: (0041) = 88
(0042) = 02
- b. (0040) = 01
Result: (0041) = 80
(0042) = 07
- c. (0040) = CB
Result: (0041) = CB
(0042) = 00
- d. (0040) = 00
Result: (0041) = 00
(0042) = 00

Flowchart:



Program 5-5a:

0000 5F		CLRB		NUMBER OF SHIFTS = ZERO
0001 96	40	LDA	\$40	GET DATA
0003 27	06	BEQ	DONE	THROUGH IF DATA IS ZERO
0005 2B	04	CHKMS	BMI	DONE
0007 5C			INCB	THROUGH IF MSB OF DATA IS 1
0008 48			ASLA	ADD 1 TO NUMBER OF SHIFTS
0009 20	FA		BRA	CHKMS
000B DD	41	DONE	STD	\$41
		*		SAVE JUSTIFIED DATA AND
				NUMBER OF SHIFTS
000D 3F			SWI	

The relative offsets are:

1. BEQ DONE

Destination address	= 000B
- Address at end of instruction	= <u>0005</u>
	06

2. BMI DONE

Destination address	= 000B
- Address at end of instruction	= <u>0007</u>
	04

3. BRA CHKMS

Destination address	= 0005 = 0005
Address at end of instruction	= 000B = <u>+FFF5</u>
	FA

ASL (Arithmetic Shift Left) shifts the contents of the specified accumulator or memory location left one bit and clears the least significant bit. The most significant bit ends up in the Carry flag and the old Carry value is lost. ASLA is equivalent to adding Accumulator A to itself; the result is, of course, twice the original number (try it!).

BMI DONE causes a branch to address DONE if the Sign flag is 1. This condition may mean that the result was a negative number, or it may just mean that the most significant bit of that result was 1. The computer only performs the operations; the programmer must provide the interpretation.

BRA is an unconditional branch; that is, it always adds the offset to the program counter. The 6809 also has the unconditional jump instruction JMP, which can use direct (base page), extended, or indexed addressing. BRA, like the conditional branch instructions, always uses relative addressing.

Reorganizing the Program

We can often reorganize programs to eliminate unconditional branches. The reorganization usually makes the initial conditions less obvious, but may save a little memory and some execution time, particularly if the processor repeats a loop many times. For example, we can reorganize the justification program as follows.

Program 5-5b:

0000 5F		CLRB		NUMBER OF SHIFTS = 0
0001 96	40	LDA	\$40	GET DATA
0003 27	06	BEQ	DONE	THROUGH IF DATA IS ZERO
0005 5A		DECB		NUMBER OF SHIFTS = -1
0006 5C		CHKMS	INCB	ADD 1 TO NUMBER OF SHIFTS
0007 48			ASLA	SHIFT DATA LEFT ONE BIT
0008 24	FC		BCC	CHKMS
000A 46			RORA	CONTINUE UNTIL CARRY BECOMES 1
000B DD	41	DONE	STD	\$41
		*		THEN SHIFT DATA BACK ONCE
				SAVE JUSTIFIED DATA AND
				NUMBER OF SHIFTS
000D 3F			SWI	

This version initializes the number of shifts to -1 and shifts the data until the Carry becomes 1. Then it shifts the data back once since the last shift was not really necessary. Show that this version is also correct. What are its advantages and disadvantages as compared to the other version? You might wish to try some other organizations to see how they compare in terms of execution time and memory usage.

PROBLEMS

5-1. CHECKSUM OF DATA

Purpose: Calculate the checksum of a series of numbers. The length of the series is in memory location 0041, and the series itself begins in memory location 0042. Store the checksum in memory location 0040. The checksum is formed by Exclusive-ORing all the numbers in the series together.

Such checksums are often used in paper tape and cassette systems to ensure that the data has been read correctly. The calculated checksum is compared to the one stored with the data — if the two checksums do not agree, the system will usually either indicate an error to the operator or automatically read the data again.

Sample Problem:

```

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26
Result: (0040) = (0042) ⊕ (0043) ⊕ (0044)
           = 28 ⊕ 55 ⊕ 26
           = 0010 1000
           ⊕ 0101 0101
           ⊕ 0111 1101
           ⊕ 0010 0110
           = 0101 1011
           = 5B

```

5-2. SUM OF 16-BIT DATA

Purpose: Calculate the sum of an array of 16-bit numbers. The length of the array is in memory location 0042 and the array itself begins in memory location 0043. Store the sum in memory locations 0040 and 0041 with the eight most significant bits in 0040. Each 16-bit number occupies two bytes of memory, with the eight most significant bits first (in the lower address). Assume that the summation does not result in any carries (i.e., the sum is a 16-bit number).

Sample Problem:

```

(0042) = 03 Length of the Array
(0043) = 28 } 28F1, First Number in Array
(0044) = F1 }
(0045) = 30 } 301A, Second Number in Array
(0046) = 1A }
(0047) = 4B } 4B89, Third Number in Array
(0048) = 89 }
Result: (0040) = A4 } A494 = 28F1 + 301A + 4B89
        (0041) = 94 }

```

Hint: Use the indexed addressing mode with autoincrementing by 2.

5-3. NUMBER OF ZERO, POSITIVE, AND NEGATIVE NUMBERS

Purpose: Determine the number of zero, positive (most significant bit = 0 but entire number not zero), and negative (most significant bit = 1) elements in a block. The length of the block is in memory location 0043, and the block itself starts in memory location 0044. Place the number of negative elements in memory location 0040, the number of zero elements in memory location 0041, and the number of positive elements in memory location 0042.

Sample Problem:

```
(0043) = 06
(0044) = 68
(0045) = F2
(0046) = 87
(0047) = 00
(0048) = 59
(0049) = 2A
```

Result: 2 negative, 1 zero, and 3 positive, so

```
(0040) = 02
(0041) = 01
(0042) = 03
```

5-4. FIND MINIMUM

Purpose: Find the smallest element in a block of data. The length of the block is in memory location 0041, and the block itself begins in memory location 0042. Store the minimum in memory location 0040. Assume that the numbers in the block are 8-bit unsigned binary numbers.

Sample Problem:

```
(0041) = 05
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72
```

Result: (0040) = 15, since this is the smallest of the five unsigned numbers

5-5. COUNT 1 BITS

Purpose: Determine how many bits in memory location 0040 are ones and place the result in memory location 0041.

Sample Problem:

```
(0040) = 3B = 0011 10112
```

Result: (0041) = 05

6

Character-Coded Data

Microprocessors often handle data which represents printed characters rather than numeric quantities. Not only do keyboards, teletypewriters, communications devices, displays, and computer terminals expect or provide character-coded data, but many instruments, test systems, and controllers also require data in this form. ASCII (American Standard Code for Information Interchange) is the most commonly used code; others include Baudot (telegraph) and EBCDIC (Extended Binary-Coded-Decimal Interchange Code).

Throughout this book, we will assume all of our character coded data to be seven-bit ASCII, as shown in Table 6-1; the character code occupies the low-order seven bits of the byte, and the most significant bit of the byte holds a 0.

HANDLING DATA IN ASCII

Here are some principles to remember in handling ASCII-coded data:

- 1. The codes for the numbers and letters form ordered subsequences.** Since the ASCII codes for the numbers 0 through 9 are 30_{16} through 39_{16} , you can convert a decimal digit to the equivalent ASCII character (and ASCII to decimal) by means of a simple additive factor: $30_{16} = \text{ASCII } 0$. Since the codes for the upper-case letters (41_{16} through $5A_{16}$) are ordered alphabetically, you can alphabetize strings by sorting them according to their numerical values.
- 2. The computer does not distinguish between printing and non-printing characters.** Only I/O devices make that distinction.

Table 6-1. Hexadecimal ASCII Character Codes

MSBs LSBs	Control Characters									
	0	1	2	3	4	5	6	7		
0	NUL	DLE	SP	0	@	P	`	p	NUL Null	DC1 Device control 1
1	SOH	DC1	!	1	A	Q	a	q	SOH Start of heading	DC2 Device control 2
2	STX	DC2	"	2	B	R	b	r	STX Start of text	DC3 Device control 3
3	ETX	DC3	#	3	C	S	c	s	ETX End of text	DC4 Device control 4
4	EOT	DC4	\$	4	D	T	d	t	EOT End of transmission	NAK Negative acknowledge
5	ENQ	NAK	%	5	E	U	e	u	ENQ Enquiry	SYN Synchronous idle
6	ACK	SYN	&	6	F	V	f	v	ACK Acknowledge	ETB End of transmission block
7	BEL	ETB	'	7	G	W	g	w	BEL Bell, or alarm	CAN Cancel
8	BS	CAN	(8	H	X	h	x	BS Backspace	EM End of medium
9	HT	EM)	9	I	Y	i	y	HT Horizontal tabulation	SUB Substitute
A	LF	SUB	*	:	J	Z	j	z	LF Line feed	ESC Escape
B	VT	ESC	+	:	K	[k	[VT Vertical tabulation	FS File separator
C	FF	FS	<	<	L	\	l	\	FF Form feed	GS Group separator
D	CR	GS	=	=	M]	m]	CR Carriage return	RS Record separator
E	SO	RS	>	>	N	^	n	^	SO Shift out	US Unit separator
F	SI	US	/	/	O	_	o	_	SI Shift in	SP Space
			?	?				DEL	DLE Data link escape	DEL Delete

3. **An ASCII I/O device handles data only in ASCII.** For example, if you want an ASCII printer to print the digit 7, you must send it 37_{16} as the data; 07_{16} is the "bell" character. Similarly, if an operator presses the "9" key on an ASCII keyboard, the input data will be 39_{16} ; 09_{16} is the "horizontal tab" character.
4. **Many ASCII devices do not use the entire character set.** For example, devices may ignore meaningless control characters and may not print lower-case letters.
5. **ASCII control characters often have widely varying interpretations.** Each ASCII device typically uses control characters in a special way to provide features such as cursor control on a CRT, and to allow software control of characteristics such as rate of data transmission, print width, and line length.
6. **Some widely used ASCII characters are:**
 - $0A_{16}$ — line feed (LF)
 - $0D_{16}$ — carriage return (CR)
 - 20_{16} — space
 - $3F_{16}$ — question mark (?)
 - $7F_{16}$ — rubout or delete character (DEL)
7. **Each ASCII character occupies eight bits.** This allows a large character set but is wasteful when only a few characters are actually being used. If, for example, the data consists entirely of decimal numbers, the ASCII format (allowing one digit per byte) requires twice as much storage, communications capacity, and processing time as does the BCD format (allowing two digits per byte).

PROGRAM EXAMPLES

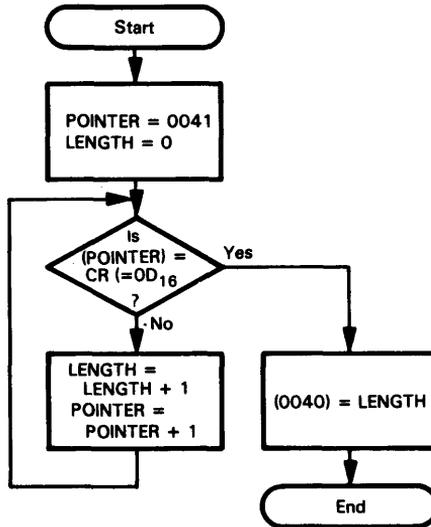
6-1. LENGTH OF A STRING OF CHARACTERS

Purpose: Determine the length of a string of characters. The string starts in memory location 0041; the end of the string is marked by an ASCII carriage return character ('CR', 0D₁₆). Place the length of the string (excluding the carriage return) into memory location 0040.

Sample Problems:

- a. (0041) = 0D
Result: (0040) = 00 since the beginning character is a carriage return
- b. (0041) = 52 'R'
(0042) = 41 'A'
(0043) = 54 'T'
(0044) = 48 'H'
(0045) = 45 'E'
(0046) = 52 'R'
(0047) = 0D CR
Result: (0040) = 06

Flowchart:



Program 6-1a:

0000 5F		CLRB		STRING LENGTH = ZERO
0001 8E 0041		LDX #S41		POINT TO START OF STRING
0004 86 0D		LDA #S0D		GET ASCII CARRIAGE RETURN
				(STRING TERMINATOR)
				IS NEXT CHARACTER
0006 A1 80	*	CHKCR CMA ,X+		A CARRIAGE RETURN?
				YES, END OF STRING
0008 27 03	*	BEQ DONE		NO, ADD 1 TO STRING LENGTH
000A 5C		INCB		
000B 20 F9		BRA CHKCR		
000D D7 40	DONE	STB S40		SAVE STRING LENGTH
000F 3F		SWI		

As far as the computer is concerned, the carriage return (CR) is just another ASCII code (0D₁₆). The fact that the carriage return causes the output device to perform a control function rather than print a symbol does not affect the computer.

The Compare instruction CMP performs a subtraction and sets the flags, but does not change the contents of the accumulator. In Program 6-1a, CMPA leaves the carriage return character in Accumulator A for later use. In this program, the CMPA instruction affects the Zero flag as follows:

Z = 1 if the character in the string is a carriage return
 Z = 0 if it is not a carriage return

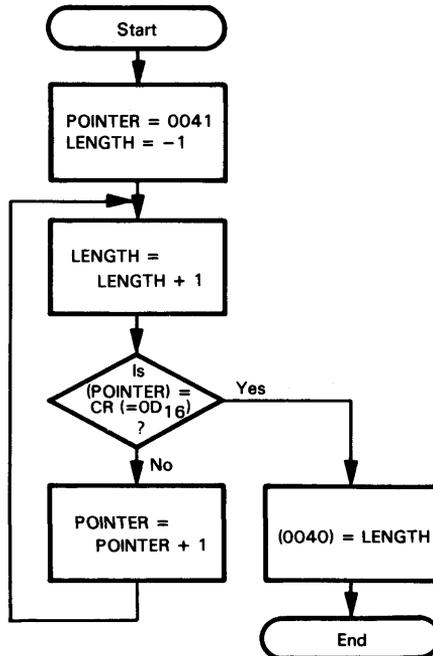
The instruction INCB adds 1 to the string length counter in Accumulator B. CLRB initializes this counter to zero before the loop begins. You must remember to initialize variables before using them in a loop; failure to do so is a common programming error.

This loop does not terminate by decrementing a counter to zero. In fact, the computer will simply continue examining characters until it finds a carriage return. Obviously, this creates problems if the string, because of an error or omission, does not contain a carriage return. It is good programming practice to place a maximum count in a loop like this, even though it does not appear to be necessary. What happens if you use the example program on a string that does not contain a carriage return?

Rearranging the Program

By rearranging the logic and changing the initial conditions, you can decrease the execution time of the program. If we rearrange the flowchart so that the program increments the string length before it checks for the carriage return, only one branch instruction is necessary instead of two.

Flowchart:



Program 6-1b:

0000 C6	FF	LDB	#\$FF	STRING LENGTH = -1
0002 8E	0041	LDX	#\$41	POINT TO START OF STRING
0005 86	0D	LDA	#\$0D	GET ASCII CARRIAGE RETURN
		*		(STRING TERMINATOR)
0007 5C		CHKCR	INCB	ADD 1 TO STRING LENGTH
0008 A1	80	*	CMPA ,X+	IS NEXT CHARACTER
				A CARRIAGE RETURN?
000A 26	FB	BNE	CHKCR	NO, KEEP CHECKING
000C D7	40	STB	\$40	YES, SAVE STRING LENGTH
000E 3F		SWI		

This program, like the previous one, has no provision for stopping if a maximum string length is reached before a carriage return is found.

6-2. FIND FIRST NON-BLANK CHARACTER

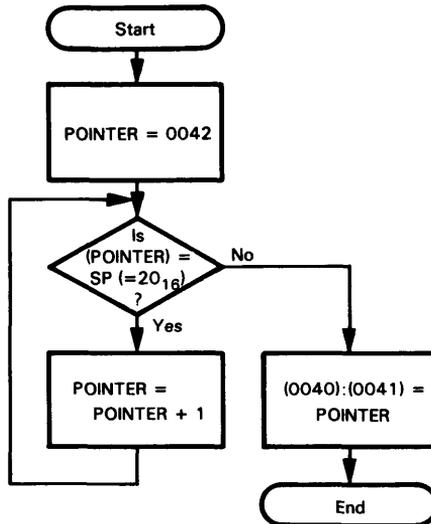
Purpose: Search a string of ASCII characters for a non-blank character. The string starts in memory location 0042. Place the address of the first non-blank character in memory locations 0040 and 0041 (most significant bits in 0040).

A blank character is exactly the same as a space, and is referred to as 'B' or 'SP'; the ASCII code for this character is 20₁₆.

Sample Problem:

- a. (0042) = 37 '7'
- Result: (0040) = 00 } since memory location 0042 contains a non-blank character
- (0041) = 42 }
- b. (0042) = 20 SP
- (0043) = 20 SP
- (0044) = 20 SP
- (0045) = 46 'F'
- (0046) = 20 SP
- Result: (0040) = 00 } since the three previous memory locations all contain blanks
- (0041) = 45 }

Flowchart:



Program 6-2:

0000	8E	0042		LDX	#\$42	POINT TO START OF STRING
0003	86	20		LDA	#'	GET ASCII SPACE FOR COMPARISON
0005	A1	80	CHBLK	CMPA	,X+	IS CHARACTER AN ASCII SPACE?
0007	27	FC		BEQ	CHBLK	YES, KEEP EXAMINING CHARS
0009	30	1F		LEAX	-1,X	NO, MOVE POINTER BACK ONE
000B	9F	40		STX	\$40	SAVE ADDRESS OF FIRST
			*			NON-BLANK CHARACTER
000D	3F			SWI		

Note the use of an apostrophe (') or single quotation mark before an ASCII character.

Looking for spaces in strings is a common task in microprocessor applications. Programs often reduce storage requirements by removing spaces that only serve to increase readability or fit data into particular formats. Storing and transmitting extra space characters obviously wastes memory, communications capacity, and processor time. However, operators find it easier to enter data and programs when the computer accepts extra spaces; the entry is then said to be in free, rather than fixed, format. **One of the most popular uses of microcomputers is to convert data and commands between the forms that are easy for people to handle and the forms that are most efficient for computers and communications systems.**

The LEA instruction has many uses in 6809 programming. This instruction calculates an effective address using one of the indexed addressing modes (see Chapter 22 for a complete description), but then simply places that address in an index register or stack pointer rather than using it to transfer data. The effective address is available for later use and need not be recalculated. This can save execution time. Remember that instructions using most of the indexed addressing modes, particularly the more complicated modes, require many additional clock cycles to execute. Furthermore, the programmer can later use the effective address in any of the indexed modes, thus providing additional levels of indirection and more flexibility.

LEA can perform many simple functions. For example, you can (as in Program 6-2) subtract 1 from Index Register X with the instruction

```
LEAX -1,X
```

In this case, the processor first calculates the effective address by adding -1 to the contents of Index Register X. It then places that result back in Index Register X. A more complex example is one that adds 8 to User Stack Pointer U and places the result in Index Register Y; the required instruction is

```
LEAY 8,U
```

The earlier 6800 microprocessor had no autoincrementing or autodecrementing. Instead, the instruction DEX subtracted 1 from Index Register X and INX added 1 to it. The 6809 assembler will accept these operation mnemonics (as well as DEY, INY, DES, and INS) and will generate the appropriate LEA instructions. The use of these operation codes saves typing and makes programs somewhat clearer (and more familiar to 6800 programmers), but we will stick with the actual 6809 operation codes.

The autoincrement in CMPA ,X+ provides us with a fast and simple way to step to the next character. However, it is a bit of a nuisance once we have found the first non-blank character, since it has then added 1 to the address that we want to save. We must explicitly subtract the extra 1 with the instruction LEAX -1,X. This instruction would not be necessary if we were working backwards instead of forward, since the 6809 autodecrements before using the address. As we noted earlier, however, you must start the index register one beyond the end of the array when autodecrementing.

6-3. REPLACE LEADING ZEROS WITH BLANKS

Purpose: Edit a string of numeric characters by replacing all leading zeros with blanks. The string starts in memory location 0041; assume that it consists entirely of ASCII-coded decimal digits. Memory location 0040 contains the length of the string in bytes.

Sample Problems:

- a. (0040) = 02 Length of the string in bytes
 (0041) = 36 '6'
 (0042) = 39 '9'

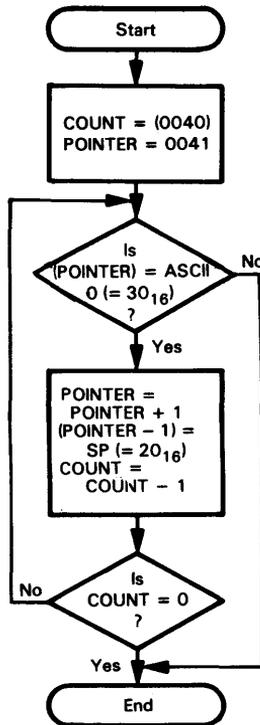
The program leaves the string unchanged, since the leading digit is not zero

- b. (0040) = 08 Length of the string in bytes
 (0041) = 30 '0'
 (0042) = 30 '0'
 (0043) = 38 '8'
 .
 .
 .

Result: (0041) = 20 SP
 (0042) = 20 SP

The program replaces the two leading zeros with ASCII spaces.
 The printed result would be " 8..." instead of "008..."

Flowchart:



Program 6-3:

0000	86	30	LDA	#'0	GET ASCII ZERO FOR COMPARISON
0002	C6	20	LDB	#'	GET ASCII SPACE FOR STORAGE
0004	8E	0041	LDX	#\$41	POINT TO START OF ARRAY
0007	A1	80	CHKZ	CMPA ,X+	IS LEADING DIGIT ZERO?
0009	26	06		BNE DONE	NO, DONE
000B	E7	1F		STB -1,X	YES, REPLACE ZERO WITH SPACE
000D	0A	40		DEC \$40	
000F	26	F6		BNE CHKZ	
0011	3F		DONE	SWI	

Editing strings of decimal digits to improve their appearance is a common task in microprocessor programs. Typical procedures include the removal of leading zeros, justification, the addition of signs (+ or -) and other delimiters or symbols for units (such as \$, ¢ %, or #), and rounding. **The program should print numbers in the form that the user wants and expects;** results like "0006," "\$27.34382," or "135000000" are annoying and difficult to interpret.

This loop has two exits — one if the processor finds a non-zero digit and the other if it works through the entire string. In an actual application, you would have to be careful to leave one zero if all the digits in the string are zero. How would you modify the program to do this?

We have assumed that the length of the string (the contents of location 0040) will be greater than zero. What will happen if (0040) = 00 when the program starts execution?

The instruction STB -1, X places an ASCII space (20₁₆) in a memory location that previously contained ASCII zero. Here again we need the offset of -1 to make up for the +1 that was added to Index Register X by the instruction CMPA ,X+.

We have assumed that all the digits in the string are in the ASCII form; that is, the digits used are 30₁₆ through 39₁₆ rather than the ordinary decimal 0 to 9. Converting a digit from BCD to ASCII is simply a matter of adding 30₁₆ (ASCII zero), while converting from ASCII to decimal involves subtracting the same number.

You can place a single ASCII character in a 6809 assembly language program by preceding it with an apostrophe ('). You can place a string of ASCII characters in program memory by using the FCC (Form Constant Characters) directive on the 6809 assembler. There are two acceptable forms of this directive:

```
MSG    FCC    5,ERROR
MSG    FCC    /ERROR/
```

In the first form, the user must specify the number of characters, followed by a comma and the character string. In the second form, the user may place any single character delimiter (we will always use /) at both ends of the string.

Each ASCII digit requires eight bits of storage, as compared to four bits for a BCD digit. Therefore, ASCII is a relatively expensive format in which to store or transmit numerical data.

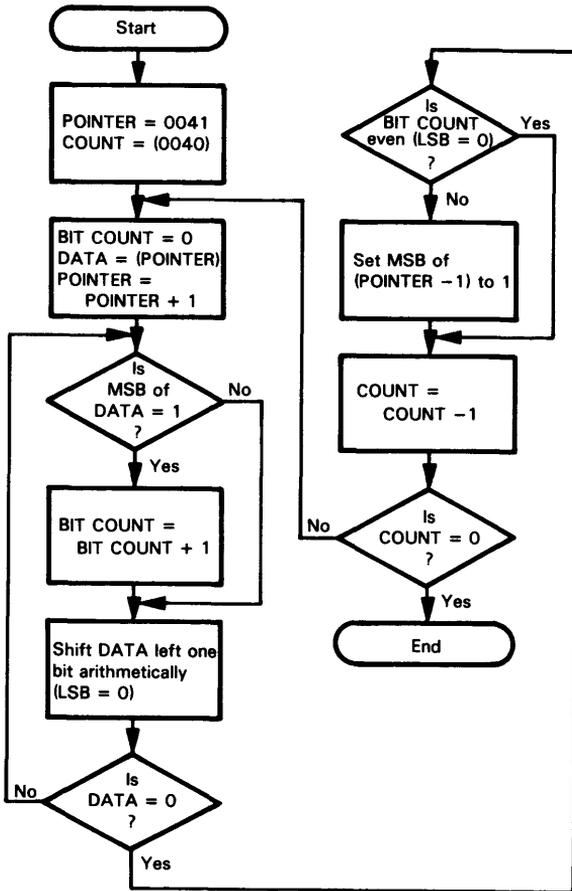
6-4. ADD EVEN PARITY TO ASCII CHARACTERS

Purpose: Add even parity to a string of 7-bit ASCII characters. The length of the string is in memory location 0040 and the string itself begins in memory location 0041. Place even parity in the most significant bit of each character by setting the most significant bit to 1 if that makes the total number of 1 bits in the byte an even number.

Sample Problem:

	(0040) = 06	Length of string
	(0041) = 31	= 0011 0001
	(0042) = 32	= 0011 0010
	(0043) = 33	= 0011 0011
	(0044) = 34	= 0011 0100
	(0045) = 35	= 0011 0101
	(0046) = 36	= 0011 0110
Result:	(0041) = B1	= 1011 0001
	(0042) = B2	= 1011 0010
	(0043) = B3	= 0011 0011
	(0044) = B4	= 1011 0100
	(0045) = B5	= 0011 0101
	(0046) = B6	= 0011 0110

Flowchart:



Program 6-4:

0000	8E	0041		LDX	#\$41	POINT TO START OF DATA BLOCK
0003	A6	80	GTBYTE	LDA	,X+	GET A BYTE OF DATA
0005	5F			CLRB		BIT COUNT = ZERO INITIALLY
0006	48		CHBIT	ASLA		SHIFT A DATA BIT TO CARRY
0007	C9	00		ADCB	#0	IF BIT IS 1, INCREMENT
			*			BIT COUNT
0009	4D			TSTA		KEEP COUNTING UNTIL
			*			DATA BECOMES ZERO
000A	26	FA		BNE	CHBIT	
000C	54			LSRB		DID DATA HAVE EVEN NUMBER
			*			OF '1' BITS?
000D	24	06		BCC	NEXTE	
000F	A6	1F		LDA	-1,X	NO, SET EVEN PARITY BIT
			*			IN DATA
0011	8A	80		ORA	##10000000	
0013	A7	1F		STA	-1,X	
0015	0A	40	NEXTE	DEC	\$40	
0017	26	EA		BNE	GTBYTE	
0019	3F			SWI		

Parity provides a simple means of checking for errors on noisy communications lines. If the transmitter sends parity along with the actual data, the receiver can then compare that parity with the parity of the data that it receives. If the two parities do not agree, the receiver can request retransmission of the data. If there is a single bit in error, the two parities will never agree, since the number of '1' bits in the data will clearly change from even to odd or odd to even. However, two bit errors will just as obviously result in the same parity as the original data. Thus we say that **parity detects single but not double bit errors.** Of course, single bit errors are usually more common than are double bit errors, so this is not a major drawback.

A more serious problem with **parity** is that it **provides no way to correct errors.** An error in any bit position will produce the same change in parity, so the receiver cannot determine which bit is wrong. **More advanced coding techniques provide for error correction as well as error detection.** Parity, however, is easy to calculate and adequate in situations in which retransmission of data is tolerable.

The procedure for calculating parity is to count the number of '1' bits in each byte of data. If that number is odd, the program sets the most significant bit (MSB) of the data byte to 1 to make the parity even. One of the advantages of the 7-bit ASCII code is that it leaves the most significant bit available for parity; the 8-bit EBCDIC code does not.

ASL clears the least significant bit of the accumulator or memory location that it is shifting. Therefore, **the result of a series of ASL instructions will eventually be zero, regardless of the original data** (try it!). The bit counting procedure in the example program does not use a counter for termination since it stops as soon as all the remaining data bits are zero. This procedure is simple and reduces execution time in most cases.

The program sets the MSB of the data byte to '1' by logically ORing it with a pattern that has a '1' in its most significant bit and zeros elsewhere. **Logically ORing a bit with '1' always produces a result of '1', while logically ORing a bit with '0' leaves the bit unchanged.**

6-5. PATTERN MATCH

Purpose: Compare two strings of ASCII characters to see if they are the same. The length of the strings is in memory location 0041; one string starts in memory location 0042 and the other in memory location 0052. If the two strings match, clear memory location 0040; otherwise, set memory location 0040 to all ones (FF_{16}).

Sample Problems:

- a.
- | | | | |
|--------|---|----|-------------------------------------|
| (0041) | = | 03 | Number of characters in each string |
| (0042) | = | 43 | 'C' |
| (0043) | = | 41 | 'A' |
| (0044) | = | 54 | 'T' |
| (0052) | = | 43 | 'C' |
| (0053) | = | 41 | 'A' |
| (0054) | = | 54 | 'T' |
- Result: (0040) = 00, since the two strings are the same
- b.
- | | | | |
|--------|---|----|-------------------------------------|
| (0041) | = | 03 | Number of characters in each string |
| (0042) | = | 52 | 'R' |
| (0043) | = | 41 | 'A' |
| (0044) | = | 54 | 'T' |
| (0052) | = | 43 | 'C' |
| (0053) | = | 41 | 'A' |
| (0054) | = | 54 | 'T' |
- Result: (0040) = FF, since the first characters in the strings differ

Note: The matching process ends as soon as the CPU finds a difference — the rest of the strings need not be examined.

Program 6-5:

0000	86	FF	LDA	#\$FF	MARK = FF HEX FOR INEQUALITY
0002	97	40	STA	\$40	
0004	8E	0042	LDX	#\$42	POINTER1 = START OF STRING 1
0007	108E	0052	LDY	#\$52	POINTER2 = START OF STRING 2
000B	D6	41	LDB	\$41	COUNT = LENGTH OF STRINGS
000D	A6	80	CHBYTE LDA	,X+	GET A CHARACTER FROM STRING 1
000F	A1	A0	CMPA	,Y+	DOES IT MATCH WITH STRING 2?
0011	26	05	BNE	DONE	NO, DONE
0013	5A		DECB		
0014	26	F7	BNE	CHBYTE	CHECK NEXT PAIR IF ANY LEFT
0016	0F	40	CLR	\$40	IF NONE ARE LEFT, MARK = 0
			*		FOR EQUALITY
0018	3F		DONE	SWI	

Matching strings of ASCII characters is an essential part of recognizing names or commands, identifying variables or operation codes in assemblers and compilers, accessing named files, and many other tasks.

Program 6-5 uses different index registers for the two strings, so they can be located anywhere in memory. **We could use a single index register if the two strings were always located a constant distance apart.** If that distance were DIST, the comparison procedure would be

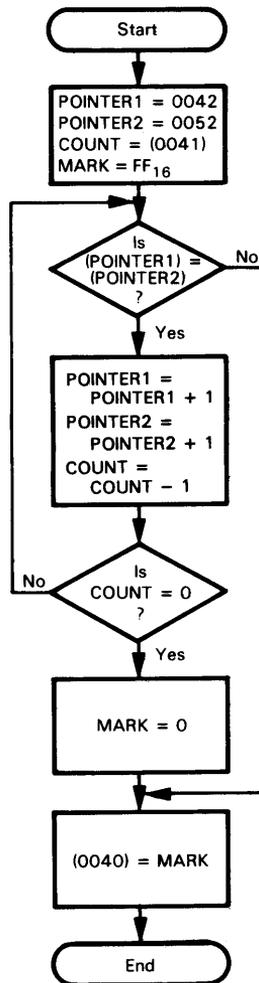
CHBYTE	LDA	,X+
	CMPA	DIST-1,X

6-12 6809 Assembly Language Programming

Program 6-5 autoincrements both Index Register X and Index Register Y. Note that LDY requires a 2-byte operation code, while LDX requires only a 1-byte code. In fact, the operation code for LDY is the operation code for LDX preceded by the byte 10_{16} . The prefix byte 10_{16} apparently tells the 6809 processor that this instruction falls in a special group and the next byte will actually describe the operation to be performed.

We could replace CLR \$40 with INC \$40 or STB \$40 (why?). Which of these alternatives executes faster? Which do you think is clearer?

Flowchart:



PROBLEMS

6-1. LENGTH OF A TELETYPEWRITER MESSAGE

Purpose: Determine the length of an ASCII message. All characters are 7-bit ASCII with MSB = 0. The string of characters in which the message is embedded starts in memory location 0041. The message itself starts with an ASCII STX character (02₁₆) and ends with ETX (03₁₆). Place the length of the message (the number of characters between the STX and the ETX but including neither) into memory location 0040.

Sample Problem:

```
(0041) = 40
(0042) = 02 STX
(0043) = 47 'G'
(0044) = 4F 'O'
(0045) = 03 ETX
```

Result: (0040) = 02, since there are two characters between the STX in location 0042 and ETX in location 0045

6-2. FIND LAST NON-BLANK CHARACTER

Purpose: Search a string of ASCII characters for the last non-blank character. The string starts in memory location 0042 and ends with a carriage return character (0D₁₆). Place the address of the last non-blank character in memory locations 0040 and 0041 (most significant bits in 0040).

Sample Problems:

```
a. (0042) = 37 '7'
    (0043) = 0D CR
```

Result: (0040) = 00 } since the last (and only) non-blank
(0041) = 42 } character is in memory location 0042

```
b. (0042) = 41 'A'
    (0043) = 20 SP
    (0044) = 48 'H'
    (0045) = 41 'A'
    (0046) = 54 'T'
    (0047) = 20 SP
    (0048) = 20 SP
    (0049) = 0D CR
```

Result: (0040) = 00
(0041) = 46

6-3. TRUNCATE DECIMAL STRING TO INTEGER FORM

Purpose: Edit a string of ASCII decimal characters by replacing all digits to the right of the decimal point with ASCII blanks (20_{16}). The string starts in memory location 0041 and is assumed to consist entirely of ASCII-coded decimal digits and a possible decimal point ($2E_{16}$). The length of the string is in memory location 0040. If no decimal point appears in the string, assume that the decimal point is implicitly at the far right.

Sample Problems:

a. (0040) = 04 Length of string
 (0041) = 37 '7'
 (0042) = 2E '.'
 (0043) = 38 '8'
 (0044) = 31 '1'

Result: (0041) = 37 '7'
 (0042) = 2E '.'
 (0043) = 20 SP
 (0044) = 20 SP

b. (0040) = 03 Length of string
 (0041) = 36 '6'
 (0042) = 37 '7'
 (0043) = 31 '1'

Result: Unchanged, as number is assumed to be 671

6-4. CHECK EVEN PARITY IN ASCII CHARACTERS

Purpose: Check even parity in a string of ASCII characters. The length of the string is in memory location 0041, and the string itself begins in memory location 0042. If the parity of all the characters in the string is correct, clear memory location 0040; otherwise, place all ones (FF_{16}) into memory location 0040.

Sample Problems:

a. (0041) = 03 Length of string
 (0042) = B1 = 1011 0001
 (0043) = B2 = 1011 0010
 (0044) = 33 = 0011 0011

Result: (0040) = 00, since all the characters have even parity

b. (0041) = 03 Length of string
 (0042) = B1 = 1011 0001
 (0043) = B6 = 1011 0110
 (0044) = 33 = 0011 0011

Result: (0040) = FF, since the character in memory location 0043 does not have even parity

6-5. STRING COMPARISON

Purpose: Compare two strings of ASCII characters to see which is larger (that is, which follows the other in alphabetical ordering). The length of the strings is in memory location 0041; one string starts in memory location 0042 and the other in memory location 0052. If the string starting in memory location 0042 is greater than or equal to the other string, clear memory location 0040; otherwise, set memory location 0040 to all ones (FF_{16}).

Sample Problems:

- a.
- | | | | |
|--------|---|----|-----------------------|
| (0041) | = | 03 | Length of each string |
| (0042) | = | 43 | 'C' |
| (0043) | = | 41 | 'A' |
| (0044) | = | 54 | 'T' |
| (0052) | = | 42 | 'B' |
| (0053) | = | 51 | 'A' |
| (0054) | = | 54 | 'T' |
- Result: (0040) = 00, since CAT is 'larger' than BAT
- b.
- | | | | |
|--------|---|----|-----------------------|
| (0041) | = | 03 | Length of each string |
| (0042) | = | 43 | 'C' |
| (0043) | = | 41 | 'A' |
| (0044) | = | 54 | 'T' |
| (0052) | = | 43 | 'C' |
| (0053) | = | 41 | 'A' |
| (0054) | = | 54 | 'T' |
- Result: (0040) = 00, since the two strings are equal
- c.
- | | | | |
|--------|---|----|-----------------------|
| (0041) | = | 03 | Length of each string |
| (0042) | = | 43 | 'C' |
| (0043) | = | 41 | 'A' |
| (0044) | = | 54 | 'T' |
| (0052) | = | 43 | 'C' |
| (0053) | = | 55 | 'U' |
| (0054) | = | 54 | 'T' |
- Result: (0040) = FF, since CUT is 'larger' than CAT

7

Code Conversion

Code conversion is a continual problem in microcomputer applications. Peripherals provide data in ASCII, BCD, or various special codes. The microcomputer must convert the data into some standard form for processing. Output devices may require data in ASCII, BCD, seven-segment, or other codes. Therefore, the microcomputer must convert the results to the proper form after it completes the processing.

There are several ways to approach code conversion:

- 1. Some conversions can easily be handled by algorithms involving arithmetic or logical functions.** The program may, however, have to handle special cases separately.
- 2. More complex conversions can be handled with lookup tables.** The lookup table method requires little programming and is easy to apply. However, the table may occupy a large amount of memory if the range of input values is large.
- 3. Hardware is readily available for some conversion tasks.** Typical examples are decoders for BCD to seven-segment conversion and Universal Asynchronous Receiver/Transmitters (UARTs) for conversion between parallel (ASCII) and serial (teletypewriter) formats.

In most applications, the program should do as much as possible of the code conversion work. This approach reduces parts count and power dissipation, saves board space, and increases reliability. Furthermore, most code conversions are easy to program and require little execution time.

PROGRAM EXAMPLES

7-1. HEXADECIMAL TO ASCII

Purpose: Convert the contents of memory location 0040 to an ASCII character. Memory location 0040 contains a single hexadecimal digit (the four most significant bits are zero). Store the ASCII character in memory location 0041.

Sample Problems:

- a. (0040) = 0C
Result: (0041) = 43 'C'
- b. (0040) = 06
Result: (0041) = 36 '6'

Program 7-1:

0000 96 40	LDA \$40	GET DATA
0002 81 09	CMPA #9	IS DATA 9 OR LESS?
0004 23 02	BLS ASCZ	
0006 8B 07	ADDA #'A-'9-1	NO, ADD OFFSET FOR LETTERS
0008 8B 30	ASCZ ADDA #'0	CONVERT DATA TO ASCII
000A 97 41	STA \$41	STORE ASCII DATA
000C 3F	SWI	

The basic idea of this program is to add ASCII 0 (30_{16}) to all the hexadecimal digits. This addition converts the digits 0 through 9 to ASCII correctly. However, the letters A through F do not follow immediately after the digit 9 in the ASCII code; instead, there is a break between the ASCII code for 9 (39_{16}) and the ASCII code for A (41_{16}), so the conversion must add a further constant to the nondecimal digits (A, B, C, D, E, and F) to account for the break. The first ADD instruction does this by adding 'A - '9 - 1 to Accumulator A. Can you explain why the extra factor for letter digits has the value 'A - '9 - 1?

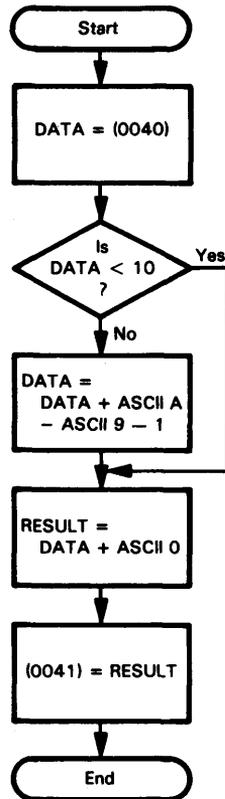
We have used the ASCII forms for the addition factors in the source program; a single quotation mark (apostrophe) before a character indicates the ASCII equivalent. We have also left the offset for the letters as an arithmetic expression to make its meaning as clear as possible. The extra assembly time is a small price to pay for the great increase in clarity. A routine like this is necessary in many applications; for example, monitor programs must convert hexadecimal digits to their ASCII equivalents in order to display the contents of memory locations in hexadecimal on an ASCII printer or CRT display.

The following program, described by Allison¹, provides a less obvious conversion method that requires no conditional branches.

0000 96 40	LDA \$40	GET HEX DIGIT
0002 8B 90	ADDA #\$90	DECIMAL ADD 90 BCD
0004 19	DAA	
0005 89 40	ADCA #\$40	DECIMAL ADD 40 BCD + CARRY
0007 19	DAA	
0008 97 41	STA \$41	
000A 3F	SWI	

Try this program on some hexadecimal digits. Can you explain why it works?

Flowchart:



7-2. DECIMAL TO SEVEN-SEGMENT

Purpose: Convert the contents of memory location 0041 to a seven-segment code in memory location 0042. If memory location 0041 does not contain a single decimal digit, clear memory location 0042.

Figure 7-1 illustrates the seven-segment display and our representation of it as a binary code. The segments are usually assigned the letters a through g as shown in Figure 7-1. We have organized the seven-segment code as shown: segment g is in bit position 6, segment f in bit position 5, e in bit position 4, and so on. Bit position 7 is always zero. The segment names are standard, but the assignment of segments to bit positions is arbitrary; in actual applications, this assignment is a hardware function.

The table in Figure 7-1 is a typical example of those used to convert decimal numbers to seven-segment code; it assumes positive logic, that is, 1 = on and 0 = off. Note that the table uses 7D for 6 rather than the alternative 7C (top bar off) to avoid confusion with lower-case b, and 6F for 9 rather than 67 (bottom bar off) for symmetry with the 6.

7-4 6809 Assembly Language Programming

Sample Problems:

- a. (0041) = 03
Result: (0042) = 4F
- b. (0041) = 28
Result: (0042) = 00

Program 7-2:

```

0000 5F          CLRB          GET ERROR CODE
                                TO BLANK DISPLAY
0001 96 41      * LDA $41      GET DATA
-0003 81 09    CMPA #9       IS DATA A DECIMAL DIGIT?
0005 22 05      BHI DONE     NO, KEEP ERROR CODE
0007 8E 0020    LDX #SSEG'    YES, GET SEVEN-SEGMENT
                                CODE FROM TABLE
000A E6 86      * LDB A,X
000C D7 42      DONE STB $42
000E 3F          * SWI
0020            * ORG $20
0020            *
0020            SSEG FCB $3F,$06,$5B,$4F,$66
0021            06
0022            5B
0023            4F
0024            66
0025            6D      FCB $6D,$7D,$07,$7F,$6F
0026            7D
0027            07
0028            7F
0029            6F

```

The program calculates the memory address of the seven-segment code by adding an index — the digit to be converted — to the base address of the seven-segment code table. This procedure is known as a “table lookup.” The addition does not require any explicit instructions, since the processor performs it automatically as part of the calculation of the effective address in the indexed addressing mode. We have used the accumulator indexed mode in which the effective address is the sum of Accumulator A and Index Register X.

The assembler directive FCB (Form Constant Byte) places constant byte-length data in program memory. Such data may include tables, headings, error messages, priming messages, format characters, thresholds, and mathematical constants. The label attached to an FCB pseudo-operation is assigned the value of the address in which the assembler places the first byte of data.

The assembler assigns the data from the FCB directive to consecutive memory addresses, with no changes other than numerical conversions. One FCB directive can fill many bytes of memory; all the programmer must do is separate the entries with commas.

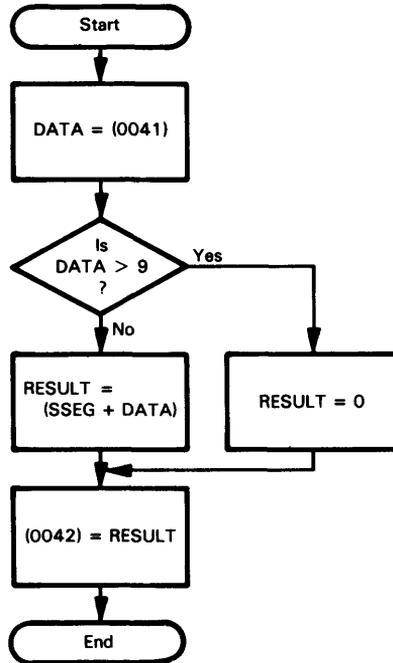
We have left some memory space between the program and the table to allow for later additions and to emphasize that they need not be located consecutively. In fact, we could place the table anywhere in memory.

Tables are a simple, fast, and convenient approach to code conversion problems that are more complex than our hexadecimal-to-ASCII example. The required lookup tables simply contain all the possible results organized by input value; that is, the first entry is the code for the number zero and so on.

Seven-segment displays provide recognizable forms of the decimal digits and a few letters and other characters. They are relatively inexpensive and easy to handle

with 8-bit microprocessors. However, many people find seven-segment coded digits somewhat difficult to read, although their widespread use in calculators and watches has made them more familiar.

Flowchart:



Note that the addition of base address (SSEG) and index (DATA) produces the address that contains the answer.

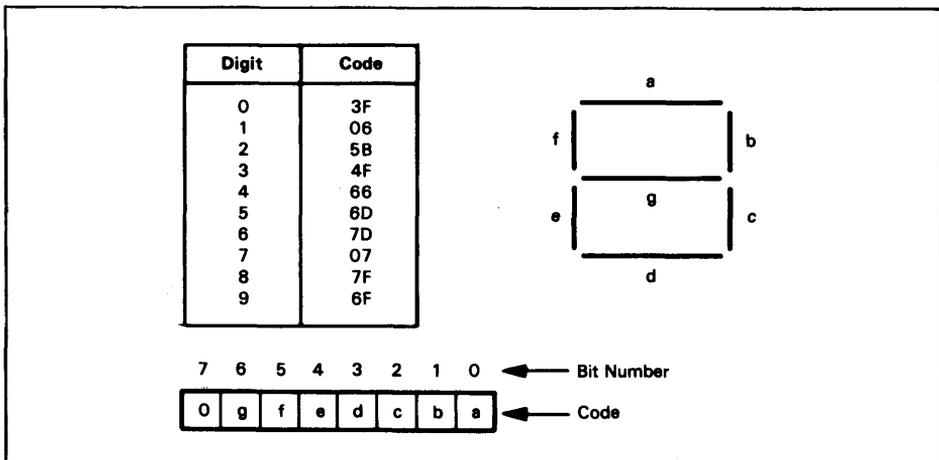


Figure 7-1. Seven-Segment Arrangement

7-3. ASCII TO DECIMAL

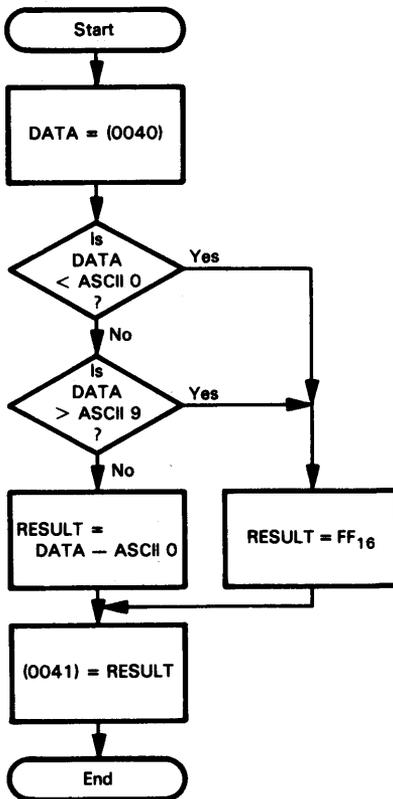
Purpose: Convert the contents of memory location 0040 from an ASCII character to a decimal digit and store the result in memory location 0041. If the contents of memory location 0040 are not the ASCII representation of a decimal digit, set the contents of memory location 0041 to FF₁₆.

Sample Problems:

- a. (0040) = 37 '7'
Result: (0041) = 07

- b. (0040) = 55 an invalid code, since it is not an
 ASCII decimal digit
Result: (0041) = FF

Flowchart:



Program 7-3:

0000 C6	FF	LDB	#\$FF	GET ERROR MARKER
0002 96	40	LDA	\$40	GET DATA
0004 80	30	SUBA	#'0	IS DATA BELOW ASCII ZERO?
0006 25	06	BLO	DONE	YES, NOT A DIGIT
0008 81	09	CMPA	#9	IS DATA ABOVE ASCII NINE?
000A 22	02	BHI	DONE	YES, NOT A DIGIT
000C 1F	89	TFR	A,B	SAVE VALID DECIMAL DIGIT
000E D7	41	DONE STB	\$41	SAVE DIGIT OR ERROR MARKER
0010 3F		SWI		

This program handles ASCII-coded characters just like ordinary numbers. Since ASCII assigns an ordered sequence of codes to the decimal digits, we can identify an ASCII character as a digit by determining if it falls within the proper range of numerical values. We could use the order of ASCII codes similarly to determine if a character is in a particular group of letters or symbols, such as A through F. This approach assumes detailed knowledge of a particular code and would not necessarily be valid for other codes.

Subtracting ASCII 0 (30_{16}) from any ASCII decimal digit gives the decimal value of that digit. An ASCII character is a decimal digit if its value lies between 30_{16} and 39_{16} (including the endpoints); how would you determine if an ASCII character is a valid hexadecimal digit? ASCII-to-decimal conversion is necessary in applications in which decimal data is entered from an ASCII device such as a teletypewriter or terminal.

The program performs one comparison — to the lower limit — with an actual subtraction (SUBA #'0), since the subtraction is necessary for the ASCII-to-decimal conversion. It performs the other comparison with an implied subtraction (CMPA #9) to avoid destroying the possible decimal digit in Accumulator A. Implied subtractions (CMP) are far more common than actual subtractions (SUB) in programs, since the numerical value of the result is seldom of interest.

The instruction TFR can transfer the contents of any 8- or 16-bit register to any other 8- or 16-bit register. TFR copies the source register into the destination register; the source register is not changed. The only restriction is that the source and destination registers must be the same length (both eight bits long or both 16 bits long). TFR instructions always require one byte besides the operation code; the high-order four bits of that byte specify the source register and the low-order four bits specify the destination register. See the description of TFR in Chapter 22 for more details.

One special use of TFR is to load the direct page register, since there is no LD instruction for that register. A typical sequence that loads the direct page register with the constant value PGNO is:

LDA	#PGNO	DIRECT PAGE = PGNO
TFR	A,DP	

An alternative to TFR is EXG (Exchange Registers). This instruction swaps the source and destination registers, thus preserving both values. For example, the following sequence will load the direct page register with the constant PGNO and save the old direct page register in memory location OLDPG.

LDA	#PGNO	DIRECT PAGE = PGNO
EXG	A,DP	
STA	OLDPG	SAVE OLD DIRECT PAGE NUMBER

7-4. BCD TO BINARY

Purpose: Convert two BCD digits in memory locations 0040 and 0041 to a binary number in memory location 0042. The most significant BCD digit is in memory location 0040.

Sample Problems:

a. (0040) = 02
(0041) = 09
Result: (0042) = $1D_{16} = 29_{10}$

b. (0040) = 07
(0041) = 01
Result: (0042) = $47_{16} = 71_{10}$

Program 7-4:

0000 96	40	LDA	\$40	GET MOST SIGNIFICANT DIGIT
0002 C6	0A	LDB	#10	MULTIPLY BY 10
0004 3D		MUL		
0005 DB	41	ADDB	\$41	ADD LEAST SIGNIFICANT DIGIT
0007 D7	42	STB	\$42	STORE BINARY EQUIVALENT
0009 3F		SWI		

The **MUL** instruction performs an unsigned 8-bit by 8-bit multiplication of the contents of Accumulators A and B; the result occupies the double accumulator D, with the high-order byte in A.

In this case, we know that the result is 90_{16} or less, so only the low-order eight bits of the product (in Accumulator B) are relevant.

Converting BCD entries to binary saves storage and simplifies calculations. BCD numbers require about 20% more memory space than do binary numbers; for example, representing the numbers 0 to 999 requires three BCD digits — 12 bits — but only 10 bits in binary since $2^{10} = 1024 \approx 1000$.

Since **MUL** requires 11 clock cycles, it is sometimes faster to multiply by small decimal numbers using repeated additions.² The instruction **ASLA** multiplies the contents of Accumulator A by 2, so multiplications by powers of 2 can be implemented as arithmetic shifts.

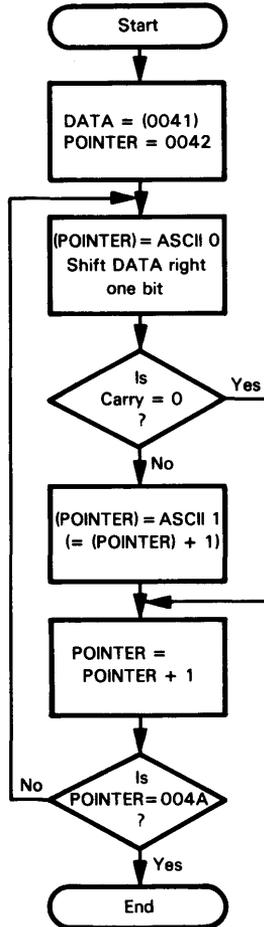
7-5. BINARY NUMBER TO ASCII STRING

Purpose: Convert the 8-bit binary number in memory location 0041 to eight ASCII characters (each either ASCII 0 or ASCII 1) in memory locations 0042 through 0049. (Place the most significant bit in location 0042.)

Sample Problem:

(0041) = D2 = 1101 0010
Result: (0042) = 31 '1'
(0043) = 31 '1'
(0044) = 30 '0'
(0045) = 31 '1'
(0046) = 30 '0'
(0047) = 30 '0'
(0048) = 31 '1'
(0049) = 30 '0'

Flowchart:



Program 7-5:

0000 C6 30	*	LDB	#'0	GET ASCII ZERO
				TO STORE IN STRING
0002 96 41		LDA	\$(41	GET DATA
0004 8E 0042		LDX	\$(42	POINT TO START OF ASCII STRING
0007 E7 80	CONV	STB	,X+	STORE ASCII ZERO IN STRING
0009 48		LSLA		IS BIT ACTUALLY 1?
000A 24 02		BCC	COUNT	
000C 6C 1F		INC	-1,X	YES, MAKE STRING ELEMENT
	*			INTO ASCII ONE
000E 8C 004A	COUNT	CMPL	\$(4A	CHECK FOR END OF CONVERSION
0011 26 F4		BNE	CONV	
0013 3F		SWI		

Since the decimal digits form a sequence in ASCII, $\text{ASCII } 1 = \text{ASCII } 0 + 1$.

The **CMP(X/Y/U/S/D)** instructions compare 16-bit quantities. The flags are set according to the result of the entire 16-bit subtraction, even though the microprocessor actually performs it eight bits at a time. CMPX takes two cycles longer

than CMPA or CMPB, while CMPD, CMPY, CMPS, and CMPU all require two-byte operation codes and take three cycles longer than CMPA or CMPB.

Single-operand instructions like INC, DEC, COM, or ASL can all use any of the indexed addressing modes. Be careful of the fact that such instructions **affect memory locations (the effective address), not the specified index register or stack pointer (except through autoincrementing or autodecrementing)**. For example, CLR ,X+ clears the byte of memory located at the address in Index Register X (and autoincrements X); it does not clear Index Register X.

Assembly-time arithmetic often comes in handy for performing address comparisons. If, for example, we established that the ASCII binary string started in the location named BINSTR, the required comparison instruction would be:

```
COUNT CMPX #BINSTR+8
```

This form is clearer and easier to change than is an explicit address. Furthermore, the programmer does not have to perform any hexadecimal arithmetic.

Binary-to-ASCII conversion is necessary if numbers are to be printed in binary on an ASCII device. Binary outputs are helpful in debugging and testing when each bit has a separate meaning; typical examples are inputs from a set of panel switches or outputs to a set of LEDs. If the programmer can only obtain the value in some other number system (such as octal or hexadecimal), he or she must perform an error-prone hand conversion to check the bits.

PROBLEMS

7-1. ASCII TO HEXADECIMAL

Purpose: Convert the contents of memory location 0040 to a hexadecimal digit and store the result in memory location 0041. Assume that memory location 0040 contains the ASCII representation of a hexadecimal digit (7 bits with MSB 0).

Sample Problems:

- a. (0040) = 43 'C'
 Result: (0041) = 0C
- b. (0040) = 36 '6'
 Result: (0041) = 06

7-2. SEVEN-SEGMENT TO DECIMAL

Purpose: Convert the contents of memory location 0040 from a seven-segment code to a decimal number in memory location 0041. If memory location 0040 does not contain a valid seven-segment code, set memory location 0041 to FF₁₆. Use the seven-segment table given in Figure 7-1 and try to match codes.

Sample Problems:

- a. (0040) = 4F
 Result: (0041) = 03
- b. (0040) = 28
 Result: (0041) = FF

7-3. DECIMAL TO ASCII

Purpose: Convert the contents of memory location 0040 from a decimal digit to an ASCII character and store the result in memory location 0041. If the number in memory location 0040 is not a decimal digit, set the contents of memory location 0041 to an ASCII space (20_{16}).

Sample Problems:

- a. (0040) = 07
 Result: (0041) = 37 '7'
- b. (0040) = 55
 Result: (0041) = 20 SP

7-4. BINARY TO BCD

Purpose: Convert the contents of memory location 0040 to two BCD digits in memory locations 0041 and 0042 (most significant digit in 0041). The number in memory location 0040 is unsigned and less than 100.

Sample Problems:

- a. (0040) = 1D = 29_{10}
 Result: (0041) = 02
 (0042) = 09
- b. (0040) = 47 = 71_{10}
 Result: (0041) = 07
 (0042) = 01

7-5. ASCII STRING TO BINARY NUMBER

Purpose: Convert the eight ASCII characters in memory locations 0042 through 0049 to an 8-bit binary number in memory location 0041 (the most significant bit is in 0042). Clear memory location 0040 if all the ASCII characters are either ASCII 1 or ASCII 0 and set it to FF_{16} otherwise.

Sample Problems:

a. (0042) = 31 '1'
 (0043) = 31 '1'
 (0044) = 30 '0'
 (0045) = 31 '1'
 (0046) = 30 '0'
 (0047) = 30 '0'
 (0048) = 31 '1'
 (0049) = 30 '0'
Result: (0041) = D2 = 1101 0010
 (0040) = 00

b. Same as above except:
 (0045) = 37 '7'
Result: (0040) = FF

REFERENCES

1. D. R. Allison, "A Design Philosophy for Microcomputer Architectures," *Computer*, February 1977, pp. 35-41. This is an excellent article which we highly recommend.
2. Other BCD-to-binary conversion methods are discussed in J. A. Tabb and M. L. Roginsky, "Microprocessor Algorithms Make BCD-Binary Conversions Superfast," *EDN*, January 5, 1977, pp. 46-50, and in J. B. Peatman, *Microcomputer-Based Design*, (New York: McGraw-Hill, 1977), pp. 400-406.

8

Arithmetic Problems

Most arithmetic in microprocessor applications consists of multi-byte binary or decimal manipulations. A decimal correction (decimal adjust) or some other means for performing decimal arithmetic is frequently the only arithmetic instruction provided beyond binary addition and subtraction. The 6809 microprocessor represents a significant advance over earlier devices in that, besides the operations mentioned, it has instructions for 16-bit addition and subtraction, 8-bit unsigned multiplication, and sign extension.

Multiple-precision binary arithmetic requires simple repetitions of the basic single-byte instructions. The Carry flag transfers information between bytes. Add with Carry and Subtract with Carry (Borrow) are the instructions that use the information from the previous arithmetic operations. You must be careful to clear the Carry before operating on the least significant bytes, since there is obviously never any carry into them or borrow from them.

Decimal arithmetic is a common enough task for microprocessors that most have special instructions for this purpose. These instructions may either perform decimal operations directly or correct the results of binary operations to the proper decimal forms. Decimal arithmetic is essential in such applications as point-of-sale terminals, calculators, check processors, order entry systems, and banking terminals. It is necessary in other applications as well (such as instrumentation, test equipment, process control, and industrial control) to allow input and output of data in the form familiar to human operators.

The 6809 microprocessor has a multiplication instruction MUL that can easily be extended to handle data that is more than 8 bits in length. You can implement division as a series of subtractions and shifts much as you ordinarily perform long division by hand. Double-byte operations are essential since division reduces the bit length of the result. Of course, multiplying or dividing by a power of 2 is simple since such operations can be implemented with an appropriate number of left or right arithmetic shifts.

PROGRAM EXAMPLES

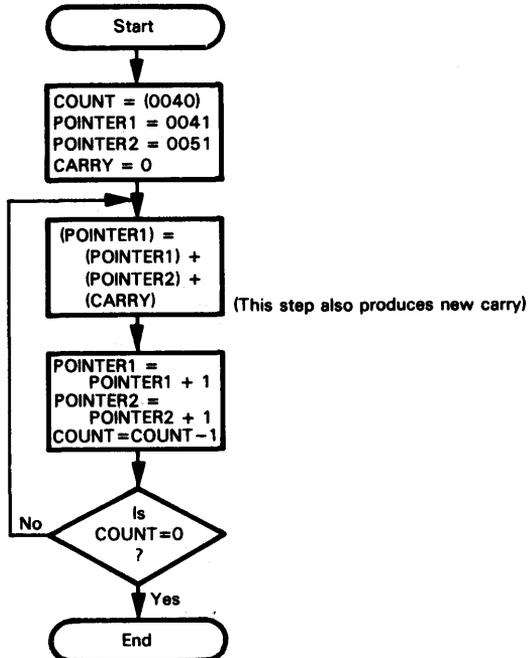
8-1. MULTIPLE-PRECISION BINARY ADDITION

Purpose: Add two multi-byte binary numbers. The length of the numbers (in bytes) is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051 respectively, and the sum replaces the number starting in memory location 0041.

Sample Problem:

(0040)	= 04	Number of bytes in each number	
(0041)	= C3	}	
(0042)	= A7		
(0043)	= 5B		
(0044)	= 2F		
2F5BA7C3 ₁₆ is first number			
(0051)	= B8	}	
(0052)	= 35		
(0053)	= DF		
(0054)	= 14		
14DF35B8 ₁₆ is second number			
Result:	(0041)	= 7B	}
	(0042)	= DD	
	(0043)	= 3A	
	(0044)	= 44	
443ADD7B ₁₆ is sum			

Flowchart:



Program 8-1:

```

0000 D6 40          LDB  $40          COUNT=LENGTH OF NUMBER IN BYTES
0002 8E 0041        LDX  #$41          POINT TO LSB'S OF FIRST NUMBER
0005 108E 0051       LDY  #$51          POINT TO LSB'S OF SECOND NUMBER
0009 1C FE          ANDCC #11111110 CLEAR CARRY TO START
000B A6 84          ADBYTE LDA ,X          GET BYTE FROM FIRST NUMBER
000D A9 A0          ADCA ,Y+         ADD BYTE FROM SECOND NUMBER
000F A7 80          STA  ,X+         STORE RESULT IN FIRST NUMBER
0011 5A             DECB
0012 26 F7          BNE  ADBYTE     CONTINUE UNTIL ALL BYTES ADDED
0014 3F             SWI

```

Clearing and Setting Flags

The instruction **ANDCC** logically ANDs the next byte of program memory with the condition code register, clearing those flags that are ANDed with '0's and leaving unchanged those flags that are ANDed with '1's. **ANDCC #11111110** thus clears bit 0 of the condition code register (the Carry flag) and leaves the other bits unchanged. The 6800 mnemonic for this operation is much clearer — **CLC** (CLEAR CARRY); of course, the 6809 **ANDCC** is more general. The program must clear the carry since there is never a carry into the least significant bytes.

The instruction **ORCC** is similar to **ANDCC**, except that it logically ORs the next byte of program memory with the condition code register, setting those flags that are ORed with '1's and leaving unchanged those flags that are ORed with '0's. **ORCC #1** thus sets bit 0 of the condition code register (the Carry flag) and leaves the other bits unchanged. As with **ANDCC**, the 6800 version of this operation mnemonic is much clearer — **SEC** (SET CARRY).

Add With Carry

The instruction **ADC** (ADD WITH CARRY) adds in the carry from the previous byte. **ADC** is the only instruction in the loop that affects the Carry flag. Note, in particular, that instructions such as **INC**, **DEC**, and **LEA** perform counting and arithmetic functions without affecting the Carry flag.

Positioning Data

This program uses two index registers so that the two numbers can be positioned independently in memory. If we used a single index register, the numbers could be located anywhere but would always have to be separated by a constant distance. We could take advantage of the User Stack Pointer **U** to store the result in a third independent set of memory locations. You might try modifying the program so that it stores the sum starting in memory location **0061**.

Decimal Accuracy in Binary Representation

This procedure can add binary numbers of any length. **Ten bits correspond to approximately three decimal digits** since $2^{10} = 1024 \approx 1000$. So you can calculate the number of bits required to give a certain accuracy in decimal digits from the formula:

$$\text{Number of bits} = (10 \div 3) \times \text{Number of decimal digits}$$

For example, twelve decimal digit accuracy requires:

$$12 \times 10 \div 3 = 40 \text{ bits}$$

One shortcoming of the 16-bit instruction **ADDD** is that it cannot be extended easily. There is no 16-bit equivalent of the **ADD WITH CARRY** instruction.

8-2. DECIMAL ADDITION

Purpose: Add two multi-byte decimal (BCD) numbers. The length of the numbers (in bytes) is in memory location 0040, the numbers themselves start (least significant digits first) in memory locations 0041 and 0051 respectively, and the sum replaces the number starting in memory location 0041.

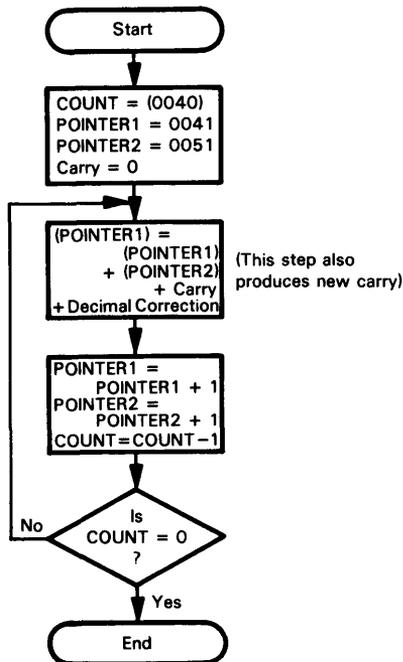
Sample Problem:

(0040) = 04	Number of bytes in each number	
(0041) = 85	}	36701985 is first number
(0042) = 19		
(0043) = 70		
(0044) = 36		
(0051) = 59	}	12663459 is second number
(0052) = 34		
(0053) = 66		
(0054) = 12		
Result: (0041) = 44	}	49365444 is decimal sum
(0042) = 54		
(0043) = 36		
(0044) = 49		

that is,

36701985	
+ 12663459	
49365444	

Flowchart:



Program 8-2:

0000 D6	40	LDB	\$40	COUNT=LENGTH OF NUMBERS IN BYTES
0002 8E	0041	LDX	#\$41	POINT TO LSB'S OF FIRST NUMBER
0005 108E	0051	LDY	#\$51	POINT TO LSB'S OF SECOND NUMBER
0009 1C	FE	ANDCC	##11111110	CLEAR CARRY TO START
000B A6	84	ADDIGS	LDA ,X	GET TWO DIGITS OF FIRST NUMBER
000D A9	A0		ADCA ,Y+	ADD TWO DIGITS OF SECOND NUMBER
000F 19			DAA	DECIMAL CORRECTION
0010 A7	80		STA ,X+	STORE RESULT IN FIRST NUMBER
0012 5A			DECB	
0013 26	F6		ADDIGS	CONTINUE UNTIL ALL DIGITS ADDED
0015 3F			SWI	

The Decimal Adjust Instruction

The Decimal Adjust (DAA) instruction uses the Carry (C) and Half-Carry (H) flags to recognize and change the following situations in which binary and BCD addition differ:

1. **The sum of two digits is between 10 and 15 inclusive.** In this case, six must be added to the sum to give the right result, e.g.,

	0101	(5)
	+ 1000	(8)
	<hr/>	
	1101	(D)
	+ 0110	
	<hr/>	
0001	0011	(BCD 13, which is correct)

2. **The sum of two digits is 16 or more.** In this case, the result is a proper BCD digit but six less than it should be, e.g.,

	1000	(8)
	+ 1001	(9)
	<hr/>	
0001	0601	(BCD 11)
	+ 0110	
	<hr/>	
0001	0111	(BCD 17, which is correct)

An extra factor of 6 is necessary in both cases. However, the processor can recognize Case 1 by determining that the sum is not a BCD digit, i.e., it is between 10 and 15 (or A and F hexadecimal). On the other hand, the processor must check the digit carry (H for the lower digit, C for the upper digit) to recognize Case 2, since the result is a valid BCD number. DAA is the only instruction that actually needs the H (Half-Carry) flag. Note that DAA only operates on Accumulator A and only works correctly after an ADDA or ADCA instruction.

You cannot use DAA after such instructions as:

1. **ADDD or SUBD**, since neither affects the H flag. Correcting the result of ADDD to decimal would obviously require three digit carry flags.
2. **ASL, ASR, NEG, SBC, or SUBA (B)**, since all of these leave the H flag in an undefined state. In particular, you can only perform decimal subtraction in a rather roundabout way (see Problem 2 at the end of the chapter). This approach involves transforming a subtraction operation into an addition

operation; if, for example, X and Y are each two digits from a string of decimal numbers, then

$$X - Y = X + 99 - Y + \overline{\text{BORROW}}$$

where BORROW is the borrow from the previous (less significant) digits.

Calculating 99 - Y is simple, since any decimal number can be subtracted from 99 without producing a borrow from either digit. You can then use DAA to add X in decimal form. Note, however, that this operation produces a carry if the result is positive but not if the result is negative. Thus the Carry has the opposite meaning from its usual significance as a borrow in subtraction operations.

3. **INC or DEC, since neither affects the Half-Carry or the Carry.** You can, however, perform a decimal increment of Accumulator A with the sequence:

```

ADD A, #1          INCREMENT ACCUMULATOR
DAA                RETAINING DECIMAL FORM
    
```

or a decimal decrement by adding 99 (hex or BCD):

```

ADD A, #99         DECREMENT ACCUMULATOR
DAA                RETAINING DECIMAL FORM
    
```

The decimal increment produces a carry if the result is 100, while the decimal decrement produces a carry unless the result is 99. Thus you can recognize either a carry or a borrow by examining the Carry flag.

4. **LEA, since it produces a 16-bit result and does not affect either the Half-Carry flag or the Carry flag.**

Binary and BCD Accuracy

The decimal addition procedure works for decimal (BCD) numbers of any length. Since each decimal digit requires four bits, twelve-digit accuracy requires

$$12 \times 4 = 48 \text{ bits}$$

as compared to 40 bits using binary addition. This is six bytes instead of five, a 20% increase.

8-3. 8-BIT BY 16-BIT BINARY MULTIPLICATION

Purpose: Multiply the 8-bit unsigned number in memory location 0040 by the 16-bit unsigned number in memory locations 0041 and 0042 (MSB's in 0041). Place the result in memory locations 0043, 0044, and 0045, with the MSB's in 0043 and the LSB's in 0045.

Sample Problems:

a. (0040) = 03 multiplier
 (0041) = 00 }
 (0042) = 05 } 0005 is multiplicand

 Result: (0043) = 00 }
 (0044) = 00 } 00000F is product
 (0045) = 0F }
 or in decimal: $3 \times 5 = 15$.

b. (0040) = 64 multiplier
 (0041) = 75 }
 (0042) = 30 } 7530 is multiplicand

 Result: (0043) = 2D }
 (0044) = C6 } 2DC6C0 is product
 (0045) = C0 }
 or in decimal: $100 \times 30,000 = 3,000,000$.

Program 8-3:

0000 96	40	LDA	\$40	GET MULTIPLIER
0002 D6	42	LDB	\$42	GET LSB'S OF MULTIPLICAND
0004 3D		MUL		MULTIPLY LSB'S
0005 DD	44	STD	\$44	SAVE PARTIAL PRODUCT
0007 96	40	LDA	\$40	GET MULTIPLIER
0009 D6	41	LDB	\$41	GET MSB'S OF MULTIPLICAND
000B 3D		MUL		MULTIPLY MSB'S
000C DB	44	ADDB	\$44	ADD LSB'S TO MSB'S
				OF PREVIOUS PARTIAL PRODUCT
000E 89	00	ADCA	#0	ADD CARRY TO MSB'S
0010 DD	43	STD	\$43	SAVE SUM OF PARTIAL PRODUCTS
0012 3F		SWI		

Extending the MUL instruction to handle longer operands works much like ordinary long multiplication. You must be careful to align the partial products correctly before adding them together. Each successive partial product is shifted 8 bits to the left from the previous product. The ADCA #0 instruction provides a convenient way to handle carries that may result from adding partial products.

Besides its obvious uses in calculators and point-of-sale terminals, multiplication is also a key part of almost all signal processing and control algorithms. The speed at which a processor can perform multiplication determines its usefulness in process control, adaptive control, signal detection, and signal analysis.

Multi-Dimensional Arrays

Another common use of multiplication is in locating elements in multi-dimensional arrays. For example, if we have an array of sensor readings organized by remote station number and sensor number, we generally refer to the reading from the 7th sensor at station number 5 as $R(5,7)$, where R is the name of the entire array. The usual method of storing such an array is to start at address RBASE with $R(0,0)$ and continue

with $R(0,1)$, $R(0,2)$, etc. If there are 3 stations (0, 1, and 2) and 4 sensors at each station (0, 1, 2, and 3), we keep the readings in the following memory locations:

Memory Location	Reading
RBASE	$R(0,0)$
RBASE + 1	$R(0,1)$
RBASE + 2	$R(0,2)$
RBASE + 3	$R(0,3)$
RBASE + 4	$R(1,0)$
RBASE + 5	$R(1,1)$
RBASE + 6	$R(1,2)$
RBASE + 7	$R(1,3)$
RBASE + 8	$R(2,0)$
RBASE + 9	$R(2,1)$
RBASE + 10	$R(2,2)$
RBASE + 11	$R(2,3)$

In general, if we know the station number I and the sensor number J , the reading $R(I,J)$ is located at address

$$\text{RBASE} + N \times I + J$$

where N is the number of sensors at each station. Thus locating a particular reading in order to update it, display it, or perform some mathematical operations on it requires a multiplication. For example, the operator might want an instrument to print the current reading of sensor #3 at station #2. To find that reading, the processor must calculate the address

$$\text{RBASE} + 4 \times 2 + 3 = \text{RBASE} + 11$$

Even more multiplications are necessary if the array has more dimensions. For example, we might organize the sensors by station number, position in the X direction, and position in the Y direction (each station thus has sensors at regular positions on a two-dimensional surface). Now we can describe a reading $R(2,3,1)$, which refers to the reading of the sensor at station #2, X position #3, and Y position #1. We can add even more dimensions, such as vertical position, type of sensor, or time of reading. Each added dimension means that the processor must perform more multiplications to locate elements in the essentially one-dimensional memory.

Execution Time

This algorithm takes 54 clock cycles (or 27 microseconds if the clock is 2 MHz) to multiply on a 6809 microprocessor. Higher speed would require additional hardware, such as one of the multiplier chips described in the References at the end of this chapter.

8-4. BINARY DIVISION

Purpose: Divide the 16-bit unsigned number in memory locations 0040 and 0041 (most significant bits in 0040) by the 8-bit unsigned number in memory location 0042. The numbers are normalized so that 1) the most significant bits of both the dividend and the divisor are zero and 2) the number in memory location 0042 is greater than the number in memory location 0040, i.e., the quotient is an 8-bit number. Store the remainder in memory location 0043 and the quotient in memory location 0044.

Sample Problems:

- a. (0040) = 00 }
 (0041) = 40 } 0040₁₆ = 64 is dividend
 (0042) = 08 divisor
 Result: (0043) = 00 remainder
 (0044) = 08 quotient
 that is, $64 \div 8 = 8$
- b. (0040) = 32 }
 (0041) = 6D } 326D₁₆ = 12,909 is dividend
 (0042) = 47 71₁₀ is divisor
 Result: (0043) = 3A 58₁₀ is remainder
 (0044) = B5 181₁₀ is quotient
 that is, $12,909 \div 71 = 181$ with a remainder of 58

Division Algorithm

You can perform division on the computer just as you would perform division with pen and paper, i.e., using trial subtractions. Since the numbers are binary, the only question is whether the bit in the quotient is 0 or 1, i.e., whether the divisor can be subtracted from what is left of the dividend. **Each step in a binary division can be reduced to the following operation:**

If the divisor can be subtracted from the eight most significant bits of the dividend without a borrow, the corresponding bit in the quotient is 1; otherwise, it is 0.

The only remaining problem is to line up the dividend and quotient properly. You can do this by shifting the dividend and quotient logically left one bit before each trial subtraction. The dividend and quotient can share a 16-bit register, since the procedure clears one bit of the dividend at the same time as it determines one bit of the quotient.

The complete process for binary division is

STEP 1 — Initialization

QUOTIENT = 0
 COUNT = 8

STEP 2 — Shift DIVIDEND and QUOTIENT to align them properly

DIVIDEND = 2 × DIVIDEND
 QUOTIENT = 2 × QUOTIENT

STEP 3 — Perform trial SUBTRACTION. If no BORROW, add 1 to QUOTIENT

If 8 MSBs of DIVIDEND \geq DIVISOR then
 MSBs of DIVIDEND = MSBs of DIVIDEND - DIVISOR
 QUOTIENT = QUOTIENT + 1

STEP 4 — Decrement counter and check for zero

COUNT = COUNT - 1
 If COUNT \neq 0, GO TO STEP 2
 REMAINDER = 8 MSBs of DIVIDEND

In the case of sample problem b, where the dividend is 326D₁₆ and the divisor is 47₁₆, the process works as follows.

Initialization:

DIVIDEND	326D
DIVISOR	47
QUOTIENT	00
COUNT	00

After fifth iteration:

DIVIDEND	33A0
DIVISOR	47
QUOTIENT	16
COUNT	03

After first iteration of STEPS 2-4. Note that the dividend is shifted prior to the trial subtraction):

DIVIDEND	1DDA
DIVISOR	47
QUOTIENT	01
COUNT	07

After sixth iteration:

DIVIDEND	2040
DIVISOR	47
QUOTIENT	2D
COUNT	02

After second iteration of STEPS 2-4:

DIVIDEND	3BB4
DIVISOR	47
QUOTIENT	02
COUNT	06

After seventh iteration:

DIVIDEND	4080
DIVISOR	47
QUOTIENT	5A
COUNT	01

After third iteration:

DIVIDEND	3068
DIVISOR	47
QUOTIENT	05
COUNT	05

After eighth iteration:

DIVIDEND	3A00
DIVISOR	47
QUOTIENT	B5
COUNT	00

After fourth iteration:

DIVIDEND	19D0
DIVISOR	47
QUOTIENT	0B
COUNT	04

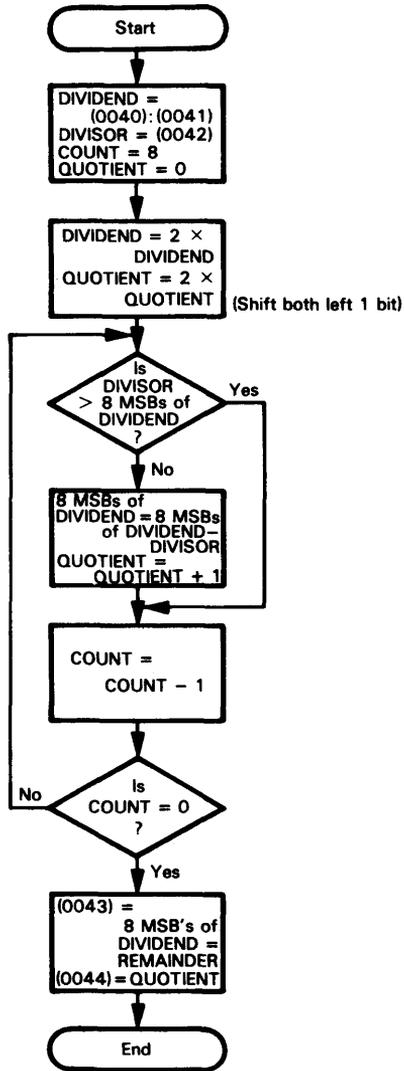
So the quotient is B5 and the remainder is 3A.

The MSBs of dividend and divisor are assumed to be zero to simplify calculations (the shift prior to the trial subtraction would otherwise place the MSB of the dividend in the Carry). Problems that are not in this form must be simplified by removing parts of the quotient that would overflow an 8-bit word. For example:

$$\frac{1024}{3} = \frac{400_{16}}{3} = 100_{16} + \frac{100_{16}}{3}$$

The last problem is now in the proper form. An extra division may be necessary.

Flowchart:



Program 8-4:

0000 86 08	LDA #8	COUNT=8
0002 97 43	STA \$43	
0004 DC 40	LDD \$40	GET DIVIDEND
0006 58	DIVIDE ASLB	SHIFT DIVIDEND, QUOTIENT
0007 49	ROLA	
0008 91 42	CMPA \$42	IS TRIAL SUBTRACTION SUCCESSFUL?
000A 25 03	BCS CHKCNT	
000C 90 42	SUBA \$42	YES, SUBTRACT AND SET BIT IN QUOTIENT
	*	
000E 5C	INCB	
000F 0A 43	CHKCNT DEC \$43	
0011 26 F3	BNE DIVIDE	
0013 DD 43	STD \$43	STORE REMAINDER, QUOTIENT
0015 3F	SWI	

Many applications, such as calculators, terminals, communications error checking, and control algorithms, involve division, but it is not nearly as common as multiplication. This is why the 6809 instruction set includes a multiplication instruction, but no division instruction. In particular, locating elements in multi-dimensional arrays requires multiplication but not division.

The algorithm takes between 170 and 230 clock cycles to divide. That corresponds to between 85 and 115 microseconds at a 6809 clock frequency of 2 MHz. The precise time depends on how many times the trial subtraction succeeds, resulting in an actual subtraction and the setting of bit 0 of the quotient. Other algorithms can reduce the execution time somewhat, but 200 clock cycles will still be typical for a software division. Higher speed requires additional hardware as described in the References at the end of this chapter.

The instructions ASLB and ROLA together produce a 16-bit arithmetic left shift of the Double Accumulator D. ASLB shifts bit 7 of Accumulator B into the Carry, and ROLA picks it up and places it in bit 0 of Accumulator A. The 6801 microprocessor has instructions that shift the Double Accumulator left logically (LSLD) and right logically (LSRD).

Accumulators A and B hold both the dividend and the quotient. The quotient simply replaces the dividend in Accumulator B as the dividend is shifted left logically.

8-5. SELF-CHECKING NUMBERS

Double Add Double Mod 10

Purpose: Calculate a checksum digit from a string of BCD digits. The length of the string (number of bytes) is in memory location 0041, and the string of digits (2 in each byte) starts in memory location 0042. Calculate the checksum digit by the Double Add Double Mod 10 technique¹ and store it in memory location 0040.

The Double Add Double Mod 10 technique works as follows:

1. Clear the checksum to start.
2. Multiply the leading digit by two and add the result to the checksum.
3. Add the next digit to the checksum.
4. Continue the alternating process until you have used all the digits.
5. The least significant digit of the checksum is the self-checking digit.

Self-Checking Numbers

Self-checking digits are commonly added to identification numbers on credit cards, inventory tags, luggage, parcels, etc. when they are handled by computerized systems. They may also be used in routing messages, identifying files, and other applications. The purpose of the digits is to minimize entry errors such as transposing digits (69 instead of 96), shifting digits (7260 instead of 3726), missing digits by one (65 instead of 64), etc. You can check the self-checking number automatically for correctness upon entry and can eliminate many errors immediately.

The analysis of self-checking methods is quite complex. For example, a plain

checksum will not find transposition errors ($4 + 9 = 9 + 4$). The Double Add Double algorithm will find simple transposition errors ($2 \times 4 + 9 = 17 \neq 2 \times 9 + 4$), but will miss some errors, such as transpositions across even numbers of digits (367 instead of 763). However, this method will find many common errors! **The value of a method depends on what errors it will detect and on the probability of particular errors in an application.**

For example, if the string of digits is:

549321

the result will be:

$$\text{Checksum} = 5 \times 2 + 4 + 9 \times 2 + 3 + 2 \times 2 + 1 = 40$$

$$\text{Self-checking digit} = 0 \text{ (least significant digit of checksum)}$$

Note that an erroneous entry like 543921 would produce a different self-checking digit (4), but erroneous entries like 049321 or 945321 would not be detected.

Sample Problems:

- a. (0041) = 03 Number of bytes
 (0042) = 36
 (0043) = 68
 (0044) = 51

Result: Checksum = $3 \times 2 + 6 + 6 \times 2 + 8 + 5 \times 2 + 1 = 43$
 (0040) = 03

- b. (0041) = 04 Number of bytes
 (0042) = 50
 (0043) = 29
 (0044) = 16
 (0045) = 83

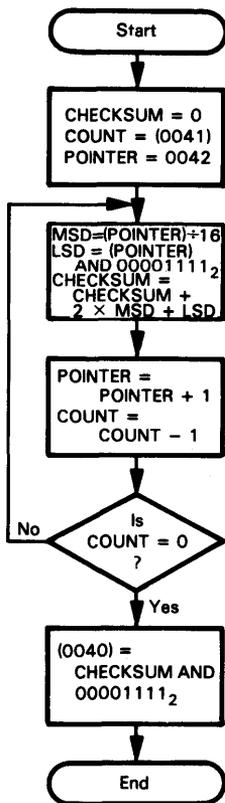
Result: Checksum = $5 \times 2 + 0 + 2 \times 2 + 9 + 1 \times 2 + 6 + 8 \times 2 + 3 = 50$
 (0040) = 00

Program 8-5:

```

0000 8E 0042      LDX  #S42      POINT TO START OF STRING
0003 0F 40        CLR  S40        CHECKSUM=ZERO
0005 A6 84        CHKDG LDA  ,X        GET NEXT 2 DIGITS OF DATA
0007 44           LSRA           SHIFT OFF LEAST SIGNIFICANT
                        *                DIGIT
0008 44           LSRA
0009 44           LSRA
000A 44           LSRA
000B 1F 89        TFR  A,B        COPY MOST SIGNIFICANT DIGIT
000D 9B 40        ADDA  S40        ADD MSD TO CHECKSUM
000F 19           DAA           RETAINING DECIMAL FORM
0010 97 40        STA  S40
0012 1F 98        TFR  B,A        AND ADD MSD TO CHECKSUM AGAIN
0014 9B 40        ADDA  S40
0016 19           DAA           RETAINING DECIMAL FORM
0017 AB 80        ADDA  ,X+      ADD IN LEAST SIGNIFICANT DIGIT
0019 19           DAA           RETAINING DECIMAL FORM
001A 97 40        STA  S40
001C 0A 41        DEC  S41        CONTINUE UNTIL ALL DIGITS ADDED
001E 26 E5        BNE  CHKDG
0020 84 0F        ANDA  #00001111 SAVE LSD OF CHECKSUM
0022 97 40        STA  S40
0024 3F           SWI
    
```

Flowchart:



Four logical right shifts move the most significant digit to the least significant bit positions. There is no reason to mask out the most significant digit before adding the least significant digit, since we do not care what happens to the most significant digit of the checksum anyway.

A decimal adjust (DAA) must follow each addition to produce the proper decimal result. A single DAA after a series of additions will not work (try it!). Remember that DAA only operates on Accumulator A.

There is no problem with carries from the various decimal sums, since the algorithm only uses the least significant digit of the checksum anyway.

Doubling and Halving Decimal Numbers

You can double a decimal number in Accumulator A by adding it to itself and then performing a decimal correction. The following sequence uses memory location 0040 for temporary storage:

```

STA    $40
ADDA   $40      DOUBLE NUMBER (ADD IT TO ITSELF)
DAA
SWI
    
```

You cannot use ASLA, because it leaves the Half-Carry flag undefined. Only ADCA, ADCB, ADDA, and ADDB set the Half-Carry flag correctly.

You can divide a decimal number by 2 simply by shifting it right logically and then subtracting 3 from any digit that has a value of 8 or larger (since 10 BCD is 16_{10}). The following program divides a decimal number in memory location 0040 by 2 and places the result in memory location 0041.

```

LDA $40      GET DECIMAL NUMBER
LSRA        DIVIDE BY 2 IN BINARY
TFR A,B     MOVE QUOTIENT TO B FOR TESTING
ANDB #$0F   MASK OFF MSD
CMPB #8     IS LSD 8 OR MORE?
BLO DONE   YES, SUBTRACT 3 FROM LSD FOR DECIMAL
SUBA #3     YES, SUBTRACT 3 FROM LSD FOR DECIMAL
STA $41     STORE RESULT
SWI
DONE

```

Try this program (and the method) on the decimal numbers 28, 30, and 37. Do you understand why it works?

Binary Rounding

Rounding numbers is simple, regardless of whether they are binary or decimal. You can round a binary number as follows:

If the most significant bit to be dropped is 1, add 1 to the remaining bits. Otherwise, do not change the remaining bits.

This rule works because 1 is halfway between 0 and 10 in binary, much as 5 is halfway in decimal (0.5 decimal = 0.1 binary).

So the following program will round a 16-bit number in memory locations 0040 and 0041 (MSB's in 0040) to an 8-bit number in memory location 0040:

```

TST $41     IS MSB OF EXTRA BYTE 1?
BPL DONE   YES, ROUND UP
INC $40     YES, ROUND UP
SWI
DONE

```

The TST instruction sets the flags according to the contents of the specified accumulator or memory location (by subtracting zero from those contents), thus allowing you to change the flags without using any registers or changing any values.

If the number is longer than 16 bits, the rounding must ripple through the other bytes as needed. Of course, the only time the rounding affects the more significant bytes is when it causes a carry. Since incrementing a memory location with INC does not affect the Carry flag, we can only recognize a carry by checking to see if the result of INC is zero. The following program increments a 16-bit number in memory locations 0040 and 0041 (MSB's in 0040).

```

INC $41     ADD 1 TO LSB'S
BNE DONE   AND CARRY TO MSB'S IF NECESSARY
INC $40     AND CARRY TO MSB'S IF NECESSARY
SWI
DONE

```

An alternative for 16-bit numbers is to use an index register as in:

```

LDX $40     GET 16-BIT DATA
LEAX 1,X    INCREMENT IT BY 1
STX $40     STORE INCREMENTED DATA

```

This approach is more general, since the step size can have any value.

Decimal Rounding

Decimal rounding is a bit more difficult, because the crossover point is now BCD 50 and the rounding must produce a decimal result. The rule is:

If the most significant digit to be dropped is 5 or more, add 1 to the remaining digits.

The following program will round a four-digit BCD number in memory locations 0040 and 0041 (MSD's in 0040) to a two-digit BCD number in memory location 0040.

```

LDA    $41          IS BYTE TO BE DROPPED 50 OR MORE?
CMPA   #550
BLO   DONE
LDA    $40          YES, ADD 1 TO MSD'S
ADDA   #1          KEEPING THEM IN DECIMAL FORM
DAA
STA    $40
DONE   SWI

```

Remember that you cannot use INC to add 1 because INC does not affect the Half-Carry flag (which could have any value). As in the binary case, rounding longer numbers requires that the carries ripple through the more significant digits as needed.

PROBLEMS

8-1. MULTIPLE-PRECISION BINARY SUBTRACTION

Purpose: Subtract one multi-byte binary number from another. The length of the numbers (in bytes) is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051 respectively, and the difference replaces the number starting in memory location 0041. Subtract the number starting in 0051 from the one starting in 0041.

Sample Problem:

```

(0040) = 04   Number of bytes
(0041) = C3
(0042) = A7   } 2F5BA7C316 is minuend
(0043) = 5B
(0044) = 2F
(0051) = B8
(0052) = 35   } 14DF35B816 is subtrahend
(0053) = DF
(0054) = 14
Result: (0041) = 0B
        (0042) = 72   } 1A7C720B16 is difference
        (0043) = 7C
        (0044) = 1A

```

8-2. DECIMAL SUBTRACTION

Purpose: Subtract one multi-byte decimal (BCD) number from another. The length of the numbers (in bytes) is in memory location 0040, the numbers themselves

start (least significant digits first) in memory locations 0041 and 0051 respectively, and the difference replaces the number starting in memory location 0041. Subtract the number starting in 0051 from the one starting in 0041.

Sample Problem:

(0040) = 04	=	04	Number of bytes	
(0041) = 85	=	85	}	
(0042) = 19	=	19		
(0043) = 70	=	70		
(0044) = 36	=	36		
			36701985 is minuend	
(0051) = 59	=	59	}	
(0052) = 34	=	34		
(0053) = 66	=	66		
(0054) = 12	=	12		
			12663459 is subtrahend	
Result:	(0041) = 26	=	26	}
	(0042) = 85	=	85	
	(0043) = 03	=	03	
	(0044) = 24	=	24	
			24038526 is decimal difference	

Hint: Remember that $X - Y = X + 99 - Y + \overline{\text{BORROW}}$

where X and Y are each two digits from the decimal strings and BORROW is the borrow from the less significant digits. The right-hand side of this equation has an extra factor of 100, but that factor has no effect on a two-digit number. Note, however, that the operations on the right-hand side produce an overall carry if $X - Y + \overline{\text{BORROW}}$ is positive but not if it is negative or zero.

8-3. 16-BIT BY 16-BIT BINARY MULTIPLICATION

Purpose: Multiply the 16-bit unsigned number in memory locations 0040 and 0041 (MSB's in 0040) by the 16-bit unsigned number in memory locations 0042 and 0043 (MSB's in 0042). Store the result in memory locations 0044 through 0047, with the most significant bits in memory location 0044.

Sample Problems:

a.	(0040) = 00	=	00	}	0003 is multiplier
	(0041) = 03	=	03		
	(0042) = 00	=	00	}	0005 is multiplicand
	(0043) = 05	=	05		
Result:	(0044) = 00	=	00	}	0000000F is product
	(0045) = 00	=	00		
	(0046) = 00	=	00		
	(0047) = 0F	=	0F		

or in decimal: $3 \times 5 = 15$.

b.	(0040) = 27	=	27	}	2710 is multiplier
	(0041) = 10	=	10		
	(0042) = 75	=	75	}	7530 is multiplicand
	(0043) = 30	=	30		
Result:	(0044) = 11	=	11	}	11E1A300 is product
	(0045) = E1	=	E1		
	(0046) = A3	=	A3		
	(0047) = 00	=	00		

or in decimal: $10,000 \times 30,000 = 300,000,000$.

8-4. SIGNED BINARY DIVISION

Purpose: Divide the 16-bit signed number in memory locations 0040 and 0041 (most significant bits in 0040) by the 8-bit signed number in memory location 0042. The numbers are normalized so that the magnitude of memory location 0042 is greater than the magnitude of memory location 0040. Store the quotient (signed) in memory location 0044 and the remainder (always positive) in memory location 0043.

Sample Problems:

a. (0040) = FF }
 (0041) = C0 } dividend is -64_{10}
 (0042) = 08 divisor

Result: (0043) = 00 remainder
 (0044) = F8 quotient is -8

or in decimal: $-64 \div 8 = -8$.

b. (0040) = ED }
 (0041) = 93 } dividend is $-4,717_{10}$
 (0042) = 47 divisor is 71_{10}

Result: (0043) = 28 remainder is $+40_{10}$
 (0044) = BD quotient is -67_{10}

That is, $-4,717 \div 71 = -67$ with a remainder of $+40$.

Hint: Determine the sign of the result, perform an unsigned division, and finally adjust the quotient and remainder to the proper forms.

8-5. SELF-CHECKING NUMBERS ALIGNED 1, 3, 7 MOD 10

Purpose: Calculate a checksum digit from a string of BCD digits. The length of the string of digits (number of bytes) is in memory location 0041, the string of digits (2 per byte) starts in memory location 0042. Calculate the checksum digit by the Aligned 1, 3, 7 Mod 10 method and store it in memory location 0040.

The Aligned 1, 3, 7 Mod 10 technique works as follows:

1. Clear the checksum to start.
2. Add the leading digit to the checksum.
3. Multiply the next digit by 3 and add the result to the checksum.
4. Multiply the next digit by 7 and add the result to the checksum.
5. Continue the process (Steps 2-4) until you have used all the digits.
6. The self-checking digit is the least significant digit of the checksum.

For example, if the string of digits is:

549321

the result will be:

$$\text{Checksum} = 5 + 3 \times 4 + 7 \times 9 + 3 + 3 \times 2 + 7 \times 1 = 96$$

$$\text{Self-checking digit} = 6$$

Sample Problems:

- a. (0041) = 03 Number of bytes
 (0042) = 36
 (0043) = 68
 (0044) = 51

Result: Checksum = $3 + 3 \times 6 + 7 \times 6 + 8 + 3 \times 5 + 7 \times 1 = 93$
 (0040) = 03

- b. (0041) = 04 Number of bytes
 (0042) = 50
 (0043) = 29
 (0044) = 16
 (0045) = 83

EL4 Result: Checksum = $5 + 3 \times 0 + 7 \times 2 + 9 + 3 \times 1 + 7 \times 6 + 8 + 3 \times 3 = 90$
 (0045) = 00

Hint: Note that $7 = 2 \times 3 + 1$ and $3 = 2 \times 1 + 1$, so the formula $M_i = 2 \times M_{i-1} + 1$ can be used to calculate the next multiplying factor.

REFERENCES

1. J. R. Herr. "Self-Checking Number Systems," *Computer Design*, June 1974, pp. 85-91.
2. Other methods for implementing multiplication, division, and other arithmetic tasks are discussed in:
 - Ali, Z. "Know the LSI Hardware Tradeoffs of Digital Signal Processors," *Electronic Design*, June 21, 1979, pp. 66-71.
 - Geist, D. J. "MOS Processor Picks up Speed with Bipolar Multipliers," *Electronics*, July 7, 1977, pp. 113-115.
 - Kolodzinski, A. and D. Wainland. "Multiplying with a Microcomputer," *Electronic Design*, January 18, 1978, pp. 78-83.
 - Mor, S. "An 8×8 Multiplier and 8-Bit Microprocessor Perform 16×16 -Bit Multiplication," *EDN*, November 5, 1979, pp. 147-152.
 - Tao, T. F. et al. "Applications of Microprocessors in Control Problems," Proceedings of the 1977 Joint Automatic Control Conference, San Francisco, Ca., June 22-24, 1977.
 - Waser, S. "State-of-the-Art in High-Speed Arithmetic Integrated Circuits," *Computer Design*, July 1978, pp. 67-75.
 - Waser, S. "Dedicated Multiplier ICs Speed up Processing in Fast Computer Systems," *Electronic Design*, September 13, 1978, pp. 98-103.
 - Waser, S. and A. Peterson. "Medium-Speed Multipliers Trim Cost, Shrink Bandwidth in Speech Transmission," *Electronic Design*, February 1, 1979, pp. 58-65.
 - Weissberger, A. J. and T. Toal. "Tough Mathematical Tasks are Child's Play for Number Cruncher," *Electronics*, February 17, 1977, pp. 102-107.

9

Tables and Lists

Tables and lists are two of the basic data structures used with all computers. We have already seen tables used to perform code conversions and arithmetic. Tables may also be used to identify or respond to commands and instructions, linearize data, provide access to files or records, define the meaning of keys or switches, and choose among alternate programs. Lists are usually less structured than tables. Lists may record tasks that the processor must perform, messages or data that the processor must record, or conditions that have changed or should be monitored. Tables are a simple way of making decisions or solving problems, since no computations or logical functions are necessary. The task, then, reduces to organizing the table so that the proper entry is easy to find. Lists allow the execution of sequences of tasks, the preparation of sets of results, and the construction of interrelated data (or data bases). Problems include how to add elements to a list and remove elements from it.

PROGRAM EXAMPLES

9-1. ADD ENTRY TO LIST

Purpose: Add the contents of memory location 0040 to a list if it is not already present in the list. The length of the list is in memory location 0041 and the list itself begins in memory location 0042.

9-2 6809 Assembly Language Programming

Sample Problems:

a. (0040) = 6B Entry to be added
 (0041) = 04 Length of list
 (0042) = 37 First element in list
 (0043) = 61
 (0044) = 38
 (0045) = 1D

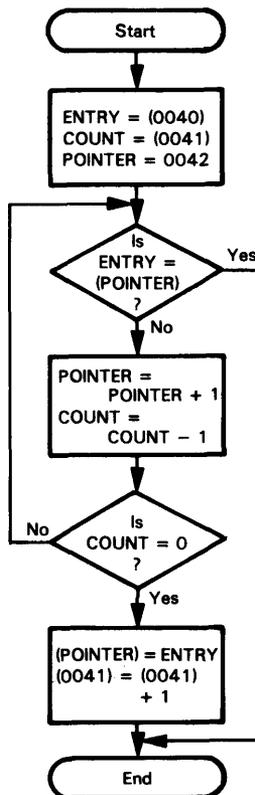
Result: (0041) = 05 New length
 (0046) = 6B

The entry 6B is added to the list, since it is not already there. The length of the list is increased by 1.

b. (0040) = 6B Entry to be added
 (0041) = 04 Length of list
 (0042) = 37 First element in list
 (0043) = 6B
 (0044) = 38
 (0045) = 1D

Result: No change, since the entry (6B) is already in the list (in memory location 0043)

Flowchart:



Program 9-1:

```

0000 8E 0042      LDX  #$42      POINT TO START OF LIST
0003 D6 41      LDB  $41      COUNT = LENGTH OF LIST
0005 96 40      LDA  $40      GET ENTRY
0007 A1 80      SRLST CMPA  ,X+    IS ENTRY = ELEMENT IN LIST?
0009 27 07      BEQ  DONE    YES, DONE
000B 5A         DECB         ALL ENTRIES EXAMINED?
000C 26 F9      BNE  SRLST   NO, KEEP LOOKING
000E A7 84      STA  ,X     YES, ADD ENTRY TO LIST
0010 0C 41      INC  $41    ADD 1 TO LIST LENGTH
0012 3F         DONE   SWI

```

Clearly, this method of adding elements is very inefficient if the list is long. We could improve the procedure by limiting the search to part of the list or by ordering the list. We could limit the search by using the entry to get a starting point in the list. This method is called "hashing," and is much like selecting a starting page in a dictionary or directory on the basis of the first letter in an entry.¹ We could order the list by numerical value. The search could then end when the list values went beyond the entry (larger or smaller, depending on the ordering technique used). A new entry would have to be inserted properly, and all the other entries would have to be moved down in the list.

The program could be restructured to use two tables. One table could provide a starting point in the other table; for example, the search point could be based on the most or least significant 4-bit digit in the entry.

The program does not work if the length of the list could be zero (what happens?). We could avoid this problem by checking the length initially. The initialization procedure would then be:

```

      LDB  $41      COUNT = LENGTH OF LIST
      BEQ  ADELM   ADD ENTRY TO LIST IF LENGTH IS ZERO
      .
      .
ADELM STA  ,X     YES, ADD ENTRY TO LIST

```

Unlike some other processors, the 6809's Zero flag is affected by simple data transfer instructions such as LD (load) and ST (store).

If each entry were more than one byte in length, a pattern-matching program would be necessary. The program would have to proceed to the next entry if a match failed; that is, skip over the last part of the current entry once a mismatch was found.

9-2. CHECK AN ORDERED LIST

Purpose: Check the contents of memory location 0041 to see if it is in an ordered list. The length of the list is in memory location 0042; the list itself begins in memory location 0043 and consists of unsigned binary numbers in increasing order. If the contents of location 0041 are in the list, clear memory location 0040; otherwise, set memory location 0040 to FF₁₆.

Sample Problems:

- a. (0041) = 6B Entry to be added
- (0042) = 04 Length of list
- (0043) = 37 First element in list
- (0044) = 55
- (0045) = 7D
- (0046) = A1

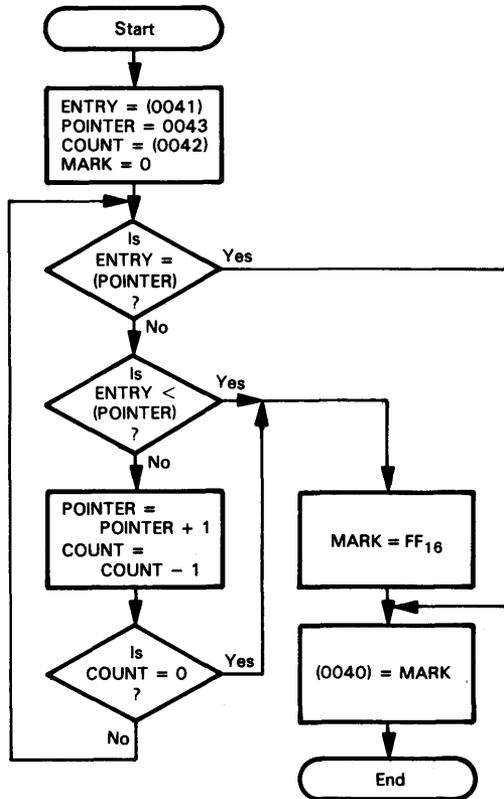
Result: (0040) = FF, since 6B is not in the list

9-4 6809 Assembly Language Programming

b. (0041) = 6B Entry to be added
 (0042) = 04 Length of list
 (0043) = 37 First element in list
 (0044) = 55
 (0045) = 6B
 (0046) = A1

 Result: (0040) = 00, since 6B is in the list

Flowchart:



The searching process is a bit different here since the elements are ordered. Once we find an element larger than the entry, the search is over, since subsequent elements will be even larger. You may want to try an example to convince yourself that the procedure works. Note that an element larger than the entry is indicated by a comparison that produces a borrow (that is, Carry = 1).

As in the previous problem, a table or other method that could choose a good starting point would speed up the search. One method would be to start in the middle and determine which half of the list the entry was in, then divide the half into halves, etc. This method is called a binary search, since it divides the remaining part of the list in half each time.^{2,3}

Program 9-2:

0000	0F	40	CLR	\$40	MARK ELEMENT AS IN LIST	
0002	8E	0043	LDX	#\$43	POINT TO START OF LIST	
0005	D6	42	LDB	\$42	COUNT = LENGTH OF LIST	
0007	96	41	LDA	\$41	GET ENTRY	
0009	A1	80	SRLST	CMPLA	,X+	IS ENTRY EQUAL TO ELEMENT?
000B	27	07	BEQ	DONE	YES, DONE	
000D	25	03	BCS	NOTIN	ENTRY NOT IN LIST IF ELEMENT IS LARGER	
			*			
000F	5A		DECB		ALL ELEMENTS EXAMINED?	
0010	26	F7	BNE	SRLST		
0012	03	40	NOTIN	COM	\$40	YES, MARK ELEMENT AS NOT IN LIST
0014	3F		DONE	SWI		

This algorithm is a bit slower than the one in Program 9-1 because of the extra conditional jump (BCS NOTIN). The average execution time for this simple search technique increases linearly with the length of the list, while the average execution time for a binary search increases logarithmically. For example, if the length of the list is doubled, the simple technique takes twice as long on the average, while the binary search method only requires one extra iteration.

9-3. REMOVE ELEMENT FROM QUEUE

Purpose: Memory locations 0042 and 0043 contain the address of the head of the queue (MSBs in 0042). Place the address of the first element (head) of a queue into memory locations 0040 and 0041 (MSBs in 0040) and update the queue to remove the element. Each element in the queue is two bytes long and contains the address of the next two-byte element in the queue. The last element in the queue contains zero to indicate that there is no next element.

Queues are used to store data in the order in which it will be used, or tasks in the order in which they will be executed. The queue is a first-in, first-out data structure; i.e., elements are removed from the queue in the same order in which they were entered. Operating systems place tasks in queues so that they will be executed in the proper order. I/O drivers transfer data to or from queues so that it will be transmitted or handled in the proper order. Buffers may be queued so that the next available one can easily be found and those that are released can easily be added to the available storage. Queues may also be used to link requests for storage, timing, or I/O so that they can be satisfied in the correct order.

In real applications, each element in the queue will typically contain a large amount of information or storage space besides the address required to link the element to the next one.

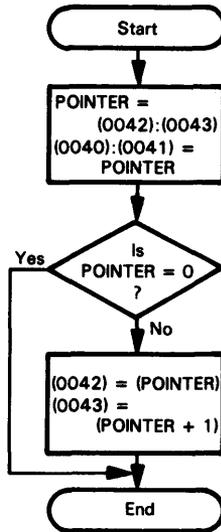
Sample Problems:

a.	(0042) = 00	} Address of first element in queue
	(0043) = 46	
	(0046) = 00	} Address of second element in queue
	(0047) = 4D	
	(004D) = 00	} End of queue
	(004E) = 00	
Result:	(0040) = 00	} Address of element removed from queue
	(0041) = 46	
	(0042) = 00	} Address of new first element in queue
	(0043) = 4D	

9-6 6809 Assembly Language Programming

b. (0042) = 00 } Empty queue
 (0043) = 00 }
 Result: (0040) = 00 } No element available from queue
 (0041) = 00 }

Flowchart:



Program 9-3:

0000 9E 42	LDX	\$42	GET ADDRESS OF HEAD OF QUEUE
0002 9F 40	STX	\$40	REMOVE HEAD OF QUEUE
0004 27 04	BEQ	DONE	DONE IF QUEUE WAS EMPTY
0006 AE 84	LDX	,X	GET ADDRESS FROM NEXT ELEMENT
0008 9F 42	STX	\$42	MOVE NEXT ELEMENT TO HEAD OF QUEUE
000A 3F	*	DONE	SWI

The 16-bit instructions LDX, LDY, LDU, STX, STY, and STU are very useful for moving addresses from one place to another. LDX, LDY, and LDU load the index register or stack pointer with the contents of the effective address and the next sequential address, thus allowing the loading of a 16-bit address with a single instruction. STX, STY, and STU similarly store a 16-bit address in memory. The addresses that are loaded or stored can later be used to fetch individual data items or addresses from a data structure.

Using Data Structures

The various indexed and indirect addressing modes allow us to use data structures in a very flexible way. If, for example, Index Register X contains the starting address of a block of information, we can refer to elements in the block with constant offsets. For example, the instruction

```
LDA $20,X
```

loads Accumulator A from the address that is 20_{16} bytes from the start of the block. The elements in the block may themselves be addresses; for example, the instruction

```
LDB [$14,X]
```

loads Accumulator B from the address that is stored 14_{16} and 15_{16} bytes from the start of the block.

How would we use such data structures? For example, we might want a piece of test equipment to execute a series of tests as specified by the operator. Using entries from a control panel, we will make up a queue of blocks of information, one for each test that the operator will eventually want to run. Each block of information contains:

1. The starting address of the next block (or 0 if there is no next block).
2. The starting address of the test program.
3. The address of the input device (e.g., keyboard, card reader, or communications line) from which data will be read during the test.
4. The address of the output device (e.g., printer, CRT terminal, or communications line) to which the results will be sent as the test is run.
5. The number of times the test will be repeated.
6. The starting address of the data area to be used for storing temporary data.
7. A flag that indicates whether failing a test should preclude continuing to the next test.

Clearly the block could contain even more information if there were more options for the operator to specify while setting up the test sequence. Note that some elements in the block contain data, others contain addresses, while still others may be 1-bit flags.

Note what we mean by flexibility in this example. Some of the procedures that the operator can easily implement are:

1. Run the same test with different sets of I/O devices. A trial run might use data from a local keyboard and send the results to the CRT, while a production run might use data from a remote communications line and produce a permanent record on a printer.
2. Execute tests in any order, just by changing the order in the queue.
3. Place temporary data in an area where it can easily be displayed or retrieved by a debugging program.
4. Make alternative decisions as to whether tests should be continued, errors should be reported, or procedures should be repeated. Here again, trial or debugging runs may use one option, while production runs use another.
5. Delete or insert tests merely by changing the links which connect a test to its successor. The operator can thus correct errors or make changes without reentering the entire list of tests.

For example, assume that the operator enters the sequence TEST 1, TEST 2, TEST 4, and TEST 5, accidentally omitting TEST 3. The blocks are linked as follows:

- Block 1 (for TEST 1) contains the starting address for block 2 (for TEST 2).
- Block 2 (for TEST 2) contains the starting address for block 3 (for TEST 4).
- Block 3 (for TEST 4) contains the starting address for block 4 (for TEST 5).
- Block 4 (for TEST 5) contains a link address of zero to indicate that it is the last block.

To insert TEST 3 between TEST 2 and TEST 4 merely involves the following changes.

Block 2 (for TEST 2) must now contain the starting address for block 5 (for TEST 3).

Block 5 (for TEST 3) must contain the starting address for block 3 (for TEST 4).

No other changes are necessary and no blocks have to be moved. Note how much simpler it is to insert or delete using linked lists, rather than lists that are stored in consecutive memory locations. There is no problem of moving elements up or down so as to remove or create empty spaces.

In our example, the blocks are organized as follows:

Byte Number	Contents
0	MSBs of starting address of next block
1	LSBs of starting address of next block
2	MSBs of starting address of test program
3	LSBs of starting address of test program
4	MSBs of input device address
5	LSBs of input device address
6	MSBs of output device address
7	LSBs of output device address
8	Number of test repetitions
9	MSBs of starting address of data area
10	LSBs of starting address of data area
11	Flag for continuation

If Index Register X contains the starting address of the block, some typical procedures are:

1. Get a byte of data from the input device and place it in byte 6 of the data area.

```
LDA [4,X]   GET INPUT DATA
LDY 9,X     GET ADDRESS OF DATA AREA
STA 6,Y     PLACE INPUT DATA IN DATA AREA
```

We need indirect addressing here since the block contains the address of the input device, not the actual input data.

2. Get a byte of data from byte 3 of the data area and send it to the output device.

```
LDY 9,X     GET ADDRESS OF DATA AREA
LDA 3,Y     GET A BYTE OF DATA
STA [6,X]   SEND DATA TO OUTPUT DEVICE
```

The indirect addressing allows us to use the address of the output device from the block. We could move that address to an index register or stack pointer if we needed it repeatedly.

3. Decrement the number of test repetitions by 1.

```
DEC 8,X     REDUCE NUMBER OF REPETITIONS BY 1
```

Queuing can handle lists that are not in sequential memory locations. Each element in the queue must contain the address of the next element. Such lists allow the programmer to handle data or tasks in the proper order, change variables or I/O devices, or fill in definitions in a program. Queuing requires extra storage as compared to sequential lists, but elements are far easier to add, delete, or insert.

Doubly Linked Lists

Sometimes you may want to maintain links in both directions. Then each element in the queue must contain the addresses of both the preceding and the following

elements.^{4,5} Such doubly linked lists allow you to easily retrace your steps (e.g., repeat the previous task if an error occurs in the current one) or access elements from either end (e.g., allowing you to remove or change the last two elements without having to go through the entire queue). **The data structure may then be used in either a first-in, first-out manner or in a last-in, first-out manner, depending on whether new elements are added to the head or to the tail.** How would you change the example program so that memory locations 0044 and 0045 contain the address of the last element (tail) of the queue?

Empty Queue

If there are no elements in the queue, the program clears memory locations 0040 and 0041. A program that requests an element from the queue must check those memory locations to see if its request has been satisfied (i.e., if there was anything in the queue). Can you suggest other ways to indicate whether the queue is empty?

9-4. 8-BIT SORT

Purpose: Sort an array of unsigned 8-bit binary numbers into descending order. The length of the array is in memory location 0041 and the array itself begins in memory location 0042.

Sample Problem:

	(0041) = 06	Length of array
	(0042) = 2A	First element of array
	(0043) = B5	
	(0044) = 60	
	(0045) = 3F	
	(0046) = D1	
	(0047) = 19	
Result:	(0042) = D1	Largest element of array
	(0043) = B5	
	(0044) = 60	
	(0045) = 3F	
	(0046) = 2A	
	(0047) = 19	Smallest element of array

Simple Sorting Algorithm

A simple sorting technique works as follows:

- Step 1. Set a flag INTER.**
- Step 2. Examine each consecutive pair of numbers in the array. If any are out of order, exchange them and clear INTER.**
- Step 3. If INTER = 0 after the entire array has been examined, return to Step 1.**

INTER will be cleared if any consecutive pair of numbers is out of order. Therefore, if INTER = 1 at the end of a pass through the entire array, the array is in proper order.

This sorting method is referred to as a "bubble sort." It is an easy algorithm to

implement. However, other sorting techniques should be considered when sorting long lists where speed is important.⁶⁻⁸

The technique operates as follows in a simple case. Let us assume that we want to sort an array into descending order; the array has four elements — 12, 03, 15, 08.

1st Iteration:

Step 1. INTER = 1

Step 2. Final order of the array is:

12
15
08
03

since the second pair (03, 15) is exchanged and so is the third pair (03, 08). INTER = 0.

2nd Iteration:

Step 1. INTER = 1

Step 2. Final order of the array is:

15
12
08
03

since the first pair (12, 15) is exchanged. INTER = 0.

3rd Iteration:

Step 1. INTER = 1

Step 2. The elements are already in order, so no exchanges are necessary and INTER remains 1.

This approach always requires one extra iteration to ensure that the elements are in the proper order. No exchanges are performed in the last iteration, so it does not really accomplish anything. Tracing through the examples shows that many of the comparisons are wasted and even repetitive. Thus the method could be improved greatly, particularly if the number of elements is in the thousands or millions, as it commonly is in large data processing applications. New sorting techniques are an important area of current research.⁹

Program 9-4:

```

0000 86 01      SORT  LDA    #1      INTERCHANGE FLAG = 1
0002 97 40      STA    $40
0004 96 41      LDA    $41      ADJUST ARRAY LENGTH TO NUMBER OF
0006 4A         DECA                   PAIRS
0007 8E 0042    PASS  LDX    #$42    POINT TO START OF ARRAY
000A E6 80      LDB    ,X+          IS PAIR OF ELEMENTS IN ORDER?
000C E1 84      CMPB   ,X
000E 24 0C      BCC    COUNT       YES, TRY NEXT PAIR
0010 0F 40      CLR    $40         NO, CLEAR INTERCHANGE FLAG
0012 34 02      PSHS   A           SAVE ARRAY COUNTER
0014 A6 84      LDA    ,X          INTERCHANGE ELEMENTS IF OUT OF
0016 E7 84      STB    ,X          ORDER
0018 A7 1F      STA    -1,X
001A 35 02      PULS   A           RESTORE ARRAY COUNTER
001C 4A         COUNT  DECA

```

```

001D 26 EB          BNE    PASS    CHECK FOR COMPLETED PASS
001F 0D 40          TST    $40     WERE ALL ELEMENTS IN ORDER?
0021 27 DD          BEQ    SORT    NO, GO THROUGH ARRAY AGAIN
0023 3F            SWI
    
```

The case where two elements in the array are equal is very important. The program should not perform an interchange in that case since that interchange would be performed in every pass. The result would be that every pass would set the interchange flag, thus producing an endless loop. The program compares the elements in the specified order so that the Carry flag is cleared if the elements are already arranged correctly. Remember that comparing two equal values always clears the Carry flag since the Carry is a borrow after subtractions or comparisons.

Since the 6809 has a complete set of unsigned conditional branches (BHI, BHS, BLO, BLS), we could perform the comparison in either direction. The sequence

```

LDB    1,X          IS PAIR OF ELEMENTS IN ORDER?
CMPB   ,X+
BLS    COUNT
    
```

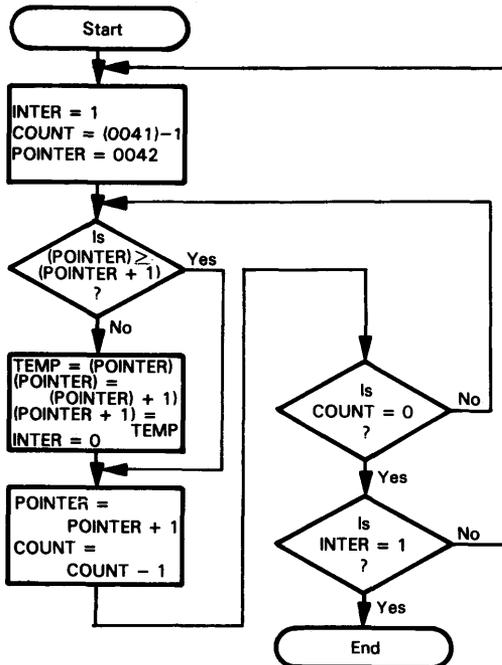
is equivalent to the one in the example program. We must use BLS rather than BLO (BCS) to force a branch if the elements are equal.

Before starting each sorting pass, we must be careful to reinitialize the index and the interchange flag.

The program must reduce the counter by 1 initially since the number of consecutive pairs is one less than the number of elements (the last element having no successor).

This program does not work properly if there are fewer than two elements in the array. How could you handle this degenerate case?

Flowchart:



Other Sorting Methods

There are many sorting algorithms that vary widely in efficiency. References 2, 7, and 8 describe some of these.

We have chosen to use the Hardware Stack for temporary storage in this problem; the advantage of this approach is that it does not tie up a specific memory address. Chapter 10 discusses the 6809's Hardware Stack in more detail. Of course, we could easily substitute a fixed memory location, such as 003F. Note the use of the special operation codes PSH for Store Registers in Stack and PUL for Load Registers from Stack, as opposed to the standard ST and LD.

9-5. USING AN ORDERED JUMP TABLE

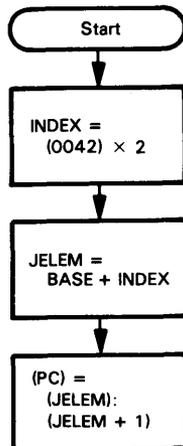
Purpose: Use the contents of memory location 0042 as an index to a jump table starting in memory location 0043. Each entry in the jump table contains a 16-bit address with the MSBs in the first byte. The program should transfer control to the address with the appropriate index; that is, if the index is 6, the program should jump to address entry #6 in the table. Assume that the table has fewer than 128 entries.

Sample Problem:

(0042)	=	02	Index for jump table
(0043)	=	00	} Zeroth element in jump table
(0044)	=	4C	
(0045)	=	00	} First element in jump table
(0046)	=	50	
(0047)	=	00	} Second element in jump table
(0048)	=	54	
(0049)	=	00	} Third element in jump table
(004A)	=	58	

Result: (PC) = 0054 since that is entry #2 (starting from zero) in the jump table. The next instruction to be executed will be the one located at that address.

Flowchart:



The last box in the flowchart results in a transfer of control to the address obtained

from the table. No ending block is necessary. Such transfers do not bother the processor at all, but you may want to add special notes to your flowchart and program documentation so that the sequence does not appear to be a "dead-end street" to the reader.

Program 9-5:

```

0000 96 42          LDA  $42          GET INDEX
0002 48             ASLA           DOUBLE INDEX FOR 2-BYTE ENTRIES
0003 8E 0043       LDX  #$43          GET BASE ADDRESS OF JUMP TABLE
0006 6E 96         JMP  [A,X]       TRANSFER CONTROL TO JUMP TABLE
                                     ENTRY

```

When you run this program, be sure to place some executable code (such as an SWI instruction) at each address to which control could be transferred. Otherwise the processor will never get back to the monitor program.

Jump Tables

Jump tables are very useful in situations where the processor must select one of several routines for execution. Such situations arise in decoding commands (entered, for example, from a control keyboard), selecting test programs, choosing alternative methods or units, or selecting an I/O configuration. For example, a 4-position switch on the front of an instrument or test system may select among the remote, self-test, automatic, or manual modes of operation. The processor reads the switch and selects the appropriate routine from a jump table as follows:

```

LDA  SWITCH          READ SWITCH POSITION
ASLA           DOUBLE INDEX FOR 2-BYTE ENTRIES
LDX  #MODES        GET BASE ADDRESS OF JUMP TABLE
JMP  [A,X]

```

The jump table is organized as follows:

Address	Contents
MODES	MSBs of starting address of REMOTE routine
MODES + 1	LSBs of starting address of REMOTE routine
MODES + 2	MSBs of starting address of SELF-TEST routine
MODES + 3	LSBs of starting address of SELF-TEST routine
MODES + 4	MSBs of starting address of AUTOMATIC routine
MODES + 5	LSBs of starting address of AUTOMATIC routine
MODES + 6	MSBs of starting address of MANUAL routine
MODES + 7	LSBs of starting address of MANUAL routine

The jump table replaces a series of conditional jump operations. The program that accesses the jump table could be used to access several different tables merely by changing the starting address.

The data must be multiplied by 2 to give the correct index since each entry in the jump table is a 16-bit address that occupies two bytes of memory. The instruction JMP [A,X] uses an indirect mode in which the destination is the address stored at the specified location rather than the location itself. The procedure is as follows:

1. Add the contents of Accumulator A and Index Register X.
2. Use that address to fetch the new value for the program counter.

JMP A,X would actually place the sum of Accumulator A and Index Register X in the program counter. JMP is an unconditional jump that allows direct (including base-page) or indexed addressing, as compared to BRA and LBRA which require relative addressing.

No terminating instruction such as SWI is necessary, since JMP A,X transfers control to the address obtained from the jump table. References 10 and 11 contain additional examples of the use of jump tables.

The program assumes that the jump table contains fewer than 128 entries (why?). How could you change the program to allow longer tables?

Jump and Branch Instructions

The terminology used in describing jump and branch instructions can be confusing. A jump instruction using direct addressing loads the specified address into the program counter; the result is more like the outcome of an LDX instruction using immediate addressing than it is like one using direct addressing. A jump instruction using one of the indirect modes works like other instructions (such as LDX or STX) using the corresponding non-indirect mode. For example,

1. JMP \$A000 transfers control to address $A000_{16}$. That is, $(PC) = A000_{16}$.

On the other hand, LDX \$A000 loads Index Register X from addresses $A000_{16}$ and $A001_{16}$. That is $(X) = (A000_{16}):(A001_{16})$.

2. JMP ,Y transfers control to the address in Index Register Y. That is, $(PC) = (Y)$.

On the other hand, LDX ,Y loads Index Register X starting at the address in Index Register Y. That is, $(X) = ((Y)):((Y) + 1)$.

However, the instruction JMP [,Y] transfers control to the address reached indirectly through Index Register Y. That is, $(PC) = ((Y)):((Y) + 1)$.

PROBLEMS

9-1. REMOVE ENTRY FROM LIST

Purpose: Remove the byte in memory location 0040 from a list if it is present. The length of the list is in memory location 0041 and the list itself begins in memory location 0042. Move the entries below the one removed up one position and reduce the length of the list by 1.

Sample Problems:

- a.
- | | |
|-------------|-------------------------------|
| (0040) = 6B | Entry to be removed from list |
| (0041) = 04 | Length of list |
| (0042) = 37 | First element in list |
| (0043) = 61 | |
| (0044) = 28 | |
| (0045) = 1D | |

Result: No change, since the entry is not in the list

- b.
- | | |
|-------------|-------------------------------|
| (0040) = 6B | Entry to be removed from list |
| (0041) = 04 | Length of list |
| (0042) = 37 | First element in list |
| (0043) = 6B | |
| (0044) = 28 | |
| (0045) = 1D | |

Result: (0041) = 03 Length of list reduced by 1
 (0043) = 28 Other elements in list moved up one position
 (0044) = 1D

The entry is removed from the list and the elements below it are moved up one position. The length of the list is reduced by 1.

9-2. ADD ENTRY TO ORDERED LIST

Purpose: Place the byte in memory location 0041 in an ordered list if it is not already there. The length of the list is in memory location 0042; the list itself begins in memory location 0043 and consists of unsigned binary numbers in increasing order. Place the new entry in the correct position in the list, adjust the elements below it down, and increase the length of the list by 1.

Sample Problems:

a. (0041) = 6B Entry to be added to list
 (0042) = 04 Length of list
 (0043) = 37 First element in list
 (0044) = 55
 (0045) = 7D
 (0046) = A1
 Result: (0042) = 05 Length of list increased by 1
 (0045) = 6B Entry placed in list
 (0046) = 7D Other elements in the list moved down one position
 (0047) = A1

b. (0041) = 6B Entry to be added to list
 (0042) = 04 Length of list
 (0043) = 37 First element in list
 (0044) = 55
 (0045) = 6B
 (0046) = A1
 Result: No change, since the entry is already in the list

9-3. ADD ELEMENT TO QUEUE

Purpose: Add the address in memory locations 0040 and 0041 (MSBs in 0040) to a queue. The address of the first element of the queue is in memory locations 0042 and 0043 (MSBs in 0042). Each element in the queue contains either the address of the next element in the queue or zero if there is no next element; all addresses are 16 bits long with the most significant bits in the first byte of the element. The new element goes at the end (tail) of the queue; its address will be in the element that was at the end of the queue and it will contain zero to indicate that it is now the end of the queue.

Sample Problem:

(0040) = 00 }
 (0041) = 4D } New element to be added to queue
 (0042) = 00 }
 (0043) = 46 } Pointer to head of queue
 (0046) = 00 }
 (0047) = 00 } Last element in queue

9-16 6809 Assembly Language Programming

```
Result: (0046) = 00 } Old last element points to new last element
        (0047) = 4D }
        (004D) = 00 } New last element in queue
        (004E) = 00 }
```

How would you add an element to the queue if memory locations 0044 and 0045 contain the address of the tail of the queue (or last element)?

9-4. 16-BIT SORT

Purpose: Sort an array of unsigned 16-bit binary numbers into descending order. The length of the array is in memory location 0040 and the array itself begins in memory location 0041. Each 16-bit number is stored with the most significant bits in the first byte.

Sample Problem:

```
(0040) = 03   Length of list
(0041) = 19 }
(0042) = D1 } 19D1   First element in list
(0043) = 3F }
(0044) = 60 } 3F60   Second element
(0045) = B5 }
(0046) = 2A } B52A   Third element
Result: (0041) = B5 } Largest element
        (0042) = 2A }
        (0043) = 3F
        (0044) = 60
        (0045) = 19 } Smallest element
        (0046) = D1 }
```

9-5. USING A JUMP TABLE WITH A KEY

Purpose: Use the contents of memory location 0042 as the key to a jump table starting in memory location 0043. Each entry in the jump table contains an 8-bit key value followed by a 16-bit address (MSBs in first byte) to which the program should transfer control if the key is equal to that key value.

Sample Problem:

```
(0042) = 38   Key value for search
(0043) = 32   Key value for first entry
(0044) = 00 }
(0045) = 4C } 004C   Jump address for first entry
(0046) = 35   Key value for second entry
(0047) = 00 }
(0048) = 50 } 0050   Jump address for second entry
(0049) = 38   Key value for third entry
(004A) = 00 }
(004B) = 54 } 0054   Jump address for third entry
Result: (PC) = 0054, since that address corresponds to key value 38
```

Note: Be sure to place some executable code (such as an SWI instruction) at each address to which the program could transfer control, so that the processor will get back to the monitor correctly.

REFERENCES

1. J. Hemenway and E. Teja. "EDN Software Tutorial: Hash Coding," *EDN*, September 20, 1979, pp. 108-10.
2. D. Knuth. *The Art of Computer Programming, Volume III: Searching and Sorting*, Addison-Wesley, Reading, Mass., 1978.
3. D. Knuth. "Algorithms," *Scientific American*, April 1977, pp. 63-80.
4. K. J. Thurber and P. C. Patton. *Data Structures and Computer Architecture*, Lexington Books, Lexington, Mass., 1977.
5. J. Hemenway and E. Teja. "Data Structures — Part 1," *EDN*, March 5, 1979, pp. 89-92. "Data Structures — Part 2," *EDN*, May 5, 1979, pp. 113-16.
6. See Reference 2.
7. B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*, McGraw-Hill, New York, 1978.
8. K. A. Schember and J. R. Rumsey "Minimal Storage Sorting and Searching Techniques for RAM Applications," *Computer*, June 1977, pp. 92-100.
9. "Sorting 30 Times Faster with DPS," *Datamation*, February 1978, pp. 200-03.
10. L. A. Leventhal. "Cut Your Processor's Computation Time," *Electronic Design*, August 16, 1977, pp. 82-89.
11. J. B. Peatman. *Microcomputer-Based Design*, McGraw-Hill, New York, 1977, Chapter 7.





Advanced Topics

The following chapters will discuss more advanced areas of assembly language programming. Chapters 10 and 11 deal with subroutines, an important aspect of all levels of programming. Chapter 10 defines and gives examples of subroutines, while Chapter 11 discusses 6809 implementations of important parameter passing techniques. The following three chapters cover input and output, a microprocessor's contact with the outside world. In Chapter 12 we discuss time delays and different types of peripherals. Chapter 13 deals with the 6820 Peripheral Interface Adapter, a popular parallel I/O device for Motorola processors, and gives examples of basic program tasks for that device. Chapter 14 illustrates basic routines for a serial interface device, the 6850 Asynchronous Communications Interface Adapter. Chapter 15 treats the important and often confusing topic of interrupts.



10

Subroutines

None of the examples that we have shown so far is typically a program all by itself. Most real programs perform a series of tasks, many of which may be the same or may be common to several different programs. We need a way to formulate these tasks once and make the formulations conveniently available both in different parts of the current program and in other programs.

Subroutine Library

The standard method is to write subroutines that perform particular tasks. The resulting sequences of instructions can be written once, tested once, and then used repeatedly. They can form a subroutine library that provides documented solutions to common problems.

Subroutine Instructions

Most microprocessors have special instructions for transferring control to subroutines and restoring control to the main program. We often refer to the special instruction that transfers control to a subroutine as Call, Jump-to-Subroutine, Jump and Mark Place, or Jump and Link. The special instruction that restores control to the main program is usually called Return.

On the 6809 microprocessor, the Jump-to-Subroutine (JSR) or Branch-to-Subroutine (BSR or LBSR) instructions save the old value of the Program Counter in the hardware stack before placing the starting address of the subroutine in the Program Counter; the Return from Subroutine (RTS) instruction gets the old value from

the Stack and puts it back in the Program Counter. The effect is to transfer program control, first to the subroutine and then back to the main program. Clearly the subroutine may itself transfer control to a subroutine, and so on.

Parameters

In order to be really useful, a subroutine must be general. A routine that can perform only a specialized task, such as looking for a particular letter in an input string of fixed length, will not be very useful. If, on the other hand, the subroutine can look for any letter in strings of any length, it will be far more helpful. **We call the data or addresses that the subroutine allows to vary “parameters.”** An important part of writing subroutines is deciding which variables should be parameters.

One problem is transferring the parameters to the subroutine; this process is called passing parameters. The simplest method is for the main program to place the parameters into registers. Then the subroutine can simply assume that the parameters are there. Of course, this technique is limited by the number of registers available. The parameters may, however, be addresses as well as data. For example, a sorting routine could begin with Index Register X containing the starting address of the array. Such 6809 features as indirect addressing, indexed addressing using the Stack Pointers, the ability to push and pop entire sets of registers with one instruction, the availability of both the user and the Hardware Stack Pointer, and the LEA instruction **provide far more powerful and more general ways of passing parameters.** The main program can place the parameters in the Stack and the subroutine can easily access them, utilize the Stack for temporary storage, and place the results back in the Stack. The only problems are keeping track of the return address (and not changing it) and cleaning the Stack of unwanted data. The two stack pointers and the LEA instruction are particularly helpful in stack management, as we shall show in Chapter 11. In that chapter we will also describe more general approaches to passing parameters.

Types of Subroutines

Sometimes a subroutine must have special characteristics. **A subroutine is relocatable if it can be placed anywhere in memory.** You can use such a subroutine easily, regardless of other programs or the arrangement of the memory. **A relocating loader is necessary to place the program in memory properly; the loader will start the program after other programs and will add the starting address or relocation constant to all addresses in the program. Position-independent code does not require a relocating loader — all addresses are expressed relative to the program counter’s current value.** We will discuss the writing of strictly relocatable or position-independent code later in this chapter.

A subroutine is reentrant if it can be interrupted and called by the interrupting program and still give the correct results for both the interrupting and interrupted programs. Reentrancy is important for standard subroutines in an interrupt-based system. Otherwise the interrupt service routines cannot use the standard subroutines without causing errors. Microprocessor subroutines are easy to make reentrant since the Call instruction uses the Stack and that procedure is automatically reentrant. The only remaining requirement is that the subroutine use the registers and Stack rather than fixed memory locations for temporary storage. This is a bit awkward, but usually can be done if necessary.

A subroutine is recursive if it calls itself. Such a subroutine clearly must also be reentrant. However, recursive subroutines are uncommon in microprocessor applications.

Subroutine Documentation

Most programs consist of a main program and several subroutines. This is advantageous because you can use proven routines and debug and test the other subroutines separately. You must, however, be careful to use the subroutines properly and remember their exact effects on registers and memory locations.

Subroutine listings must provide enough information so that users need not examine the subroutine's internal structure. Among the necessary specifications are:

- A description of the purpose of the subroutine
- A list of input and output parameters
- Registers and memory locations used
- A sample case, perhaps including a sample calling sequence.

The subroutine will be easy to use if you follow these guidelines.

Hardware Stack

The following examples all reserve an area of memory for the hardware stack. We have arbitrarily started the hardware stack at address 00FF by initializing the Stack Pointer to 0100₁₆. If your microcomputer establishes a Stack area, you may use it instead and you will not need an initial LDS instruction. If you wish to establish your own stack area, remember to save and restore the monitor's Stack Pointer (in two specified RAM locations) in order to produce a proper return at the end of your main program.

PROGRAM EXAMPLES

10-1. CONVERTING HEXADECIMAL TO ASCII

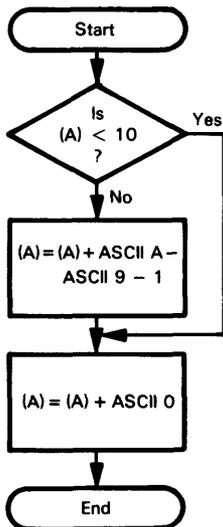
Purpose: Convert the contents of Accumulator A from a hexadecimal digit to an ASCII character. Assume that the original contents of Accumulator A are a valid hexadecimal digit.

Sample Problems:

- | | | | |
|----|---------|-----|----------|
| a. | (A) | = | 0C |
| | Result: | (A) | = 43 'C' |
| b. | (A) | = | 06 |
| | Result: | (A) | = 36 '6' |

10-4 6809 Assembly Language Programming

Flowchart:



Program 10-1:

The calling program starts the Stack at memory location 00FF, gets the data from memory location 0040, calls the conversion subroutine, and stores the result in memory location 0041.

0000			ORG	\$0000	
0000	10CE	0100	LDS	#\$100	START STACK AT MEMORY LOCATION
					00FF
0004	96	40	LDA	\$40	GET HEXADECIMAL DATA
0006	BD	0020	JSR	ASDEC	CONVERT DATA TO ASCII
0009	97	41	STA	\$41	STORE RESULT
000B	3F		SWI		

The subroutine converts the hexadecimal data to ASCII.

0020			ORG	\$0020		
0020	81	09	ASDEC	CMPA	#9	IS DATA A DECIMAL DIGIT?
0022	23	02		BLS	ASCZ	
0024	8B	07		ADDA	#\$'A-'9-1	NO, ADD EXTRA OFFSET FOR
						LETTERS
0026	8B	30	ASDC	ADDA	#\$'0	CONVERT DATA TO ASCII BY
0028	39		RTS			ADDING ZERO

Subroutine Documentation:

```

*
*SUBROUTINE ASDEC
*
*PURPOSE: ASDEC CONVERTS A HEXADECIMAL
* DIGIT IN ACCUMULATOR A TO AN
* ASCII DIGIT IN ACCUMULATOR A
*
*INITIAL CONDITIONS: HEXADECIMAL DIGIT IN A
*
*FINAL CONDITIONS: ASCII CHARACTER IN A
*
*REGISTERS AFFECTED: A, FLAGS
*
*SAMPLE CASE
* INITIAL CONDITIONS: 6 IN ACCUMULATOR A
* FINAL CONDITIONS: ASCII 6 (HEX 36)
* IN ACCUMULATOR A
  
```

The 6809 Stack grows downward (toward lower addresses); the Stack Pointer always contains the address of the last occupied location, rather than the next empty one as on some other microprocessors (including the 6800 and 6502). This means you must initialize the Stack Pointer to a value one higher than the largest address in the Stack area (e.g., initializing the Stack Pointer to 0100_{16} means that the largest address in the Stack area will be $00FF_{16}$).

JSR Instruction

The Jump to Subroutine instruction places the starting address of the subroutine (0020) in the Program Counter and saves the current value of the program counter (the address immediately following the JSR instruction) in the hardware stack. The procedure is:

- STEP 1 — Decrement Stack Pointer, save LSB's of current Program Counter in Stack.
- STEP 2 — Decrement Stack Pointer, save MSB's of current Program Counter in Stack.
- STEP 3 — Place starting address of subroutine in Program Counter.

The 6809 always decrements the Stack Pointer before storing a byte of data, so the procedure is the same as in the autodecrement addressing mode. Although the processor stores the LSB's of the current program counter first, the address ends up in the usual 6809 form (MSB's at the lower address) since the Stack is growing down (toward lower addresses).

The overall effect of JSR is:

```

((S)-1) — (PCL)
((S)-2) — (PCH)
(S) — (S) - 2
(PC) — EA

```

where PCH and PCL are the most and least significant bytes of the Program Counter, respectively, S is the Hardware Stack Pointer, and EA is the effective address for the JSR instruction. Since the processor has fetched the entire JSR instruction, the program counter contains the address of the following byte.

In our example, the effect of JSR ASDEC is:

```

(00FF) — 09 } Return address
(00FE) — 00 }
(S) — 00FE
(PC) — 0020

```

The only difference between JSR and JMP is that JSR "remembers" where it came from, thus providing for the resumption of the main program. The processor keeps a record in the hardware stack, much as one might jot down a starting point on a piece of paper. The advantages of using the stack are that it is ordered and expandable; subroutines can themselves call subroutines and so on without destroying any of the return addresses or restoring them in the wrong order. The latest return address is always at the top of the hardware stack, with the others under it in the order in which they will be used.

RTS Instruction

The **Return from Subroutine (RTS)** instruction retrieves the return address from the Stack (loading the top two bytes) and places that address back in the Program Counter. The procedure is:

- STEP 1 — Load top byte from the stack into the MSB's of the Program Counter, increment Stack Pointer.
- STEP 2 — Load top byte from the stack into the LSB's of the Program Counter, increment Stack Pointer.

The 6809 microprocessor always increments the Stack Pointer after loading a byte of data, so the procedure is the same as in the autoincrement addressing mode. RTS balances JSR, much as a right parenthesis balances a left parenthesis. The actions of RTS, however, are automatic; it simply takes the top two bytes in the hardware stack and places them in the Program Counter. The programmer must ensure that those top two bytes contain a legitimate return address; the processor does not examine them.

The overall effect of RTS is:

```
(PCH) ← ((S))
(PCL) ← ((S) + 1)
(S) ← (S) + 2
```

In our example, RTS has the following effects:

```
(PC) ← (00FE):(00FF) = 0009
(S) ← 0100
```

Parameters and Subroutine Characteristics

This subroutine has a single parameter and produces a single result. An accumulator is the obvious place to put both the parameter and the result.

The calling program consists of three steps: placing the data in the Accumulator, calling the subroutine, and storing the result. The overall initialization program must also load the Hardware Stack Pointer with the appropriate address.

This subroutine is reentrant since it uses no data memory; it is relocatable since the address ASCZ is relative. The use of BSR (Branch-to-Subroutine) rather than JSR would make the calling program relocatable as well.

The Jump-to-Subroutine instruction results in the execution of four or five instructions, taking 12 or 14 clock cycles. A subroutine call may take a long time even though it appears to be a single instruction in the program. Calling a subroutine always involves some overhead as well, since both the Jump-to-Subroutine and the Return-from-Subroutine instructions take time. In fact, a JSR takes 4 clock cycles longer than the corresponding JMP (with the same addressing mode) because JSR must save the current Program Counter in the RAM stack; RTS always takes 5 clock cycles.

If you use the stack for passing parameters, remember that Jump or Branch to Subroutine always saves the return address at the top of the stack. You can refer to the parameters using indexed addressing with offsets of 2 or more from the Hardware Stack Pointer (the return address occupies the addresses with offsets 0 and 1).

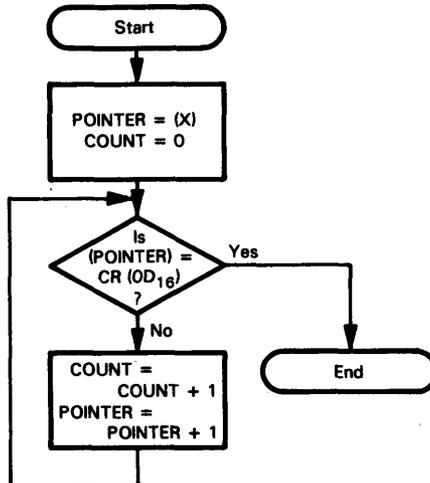
10-2. LENGTH OF A STRING OF CHARACTERS

Purpose: Determine the length of a string of ASCII characters. The starting address of the string is in Index Register X. The end of the string is marked by a carriage return character ('CR', $0D_{16}$). Place the length of the string (excluding the carriage return) in Accumulator B.

Sample Problems:

- a.
- | | | | |
|--------|---|------|----------------------------|
| (X) | = | 0043 | Starting address of string |
| (0043) | = | 52 | 'R' |
| (0044) | = | 41 | 'A' |
| (0045) | = | 54 | 'T' |
| (0046) | = | 48 | 'H' |
| (0047) | = | 45 | 'E' |
| (0048) | = | 52 | 'R' |
| (0049) | = | 0D | CR |
- Result: (B) = 06
- b.
- | | | | |
|--------|---|------|----------------------------|
| (X) | = | 0043 | Starting address of string |
| (0043) | = | 0D | CR |
- Result: (B) = 00

Flowchart:



Program 10-2:

The calling program starts the Stack at memory location 00FF, gets the starting address of the string from memory locations 0040 and 0041, calls the string length subroutine, and stores the result in memory location 0042.

0000			ORG	\$0000	
0000	10CE	0100	LDS	#\$100	START STACK AT MEMORY LOCATION
		*			00FF
0004	9E	40	LDX	\$40	GET STARTING ADDRESS OF STRING
0006	BD	0020	JSR	STLEN	DETERMINE LENGTH OF STRING
0009	D7	42	STB	\$42	STORE STRING LENGTH
000B	3F		SWI		

The subroutine determines the length of the string of ASCII characters and places the length in Accumulator B.

```

0020                ORG     $0020
0020 C6    FF      STLEN  LDB     #$FF      STRING LENGTH = -1
0022 86    0D                LDA     #$0D      GET ASCII CARRIAGE RETURN TO
                                *                               COMPARE
0024 5C                CHKCR INCB          ADD 1 TO STRING LENGTH
0025 A1    80                CMPA     ,X+    IS NEXT CHARACTER A CARRIAGE
                                *                               RETURN?
0027 26    FB                BNE     CHKCR   NO, KEEP LOOKING
0029 39                RTS

```

Subroutine Documentation:

```

*SUBROUTINE STLEN
*
*PURPOSE: STLEN DETERMINES THE LENGTH
* OF A STRING (NUMBER OF CHARACTERS
* BEFORE A CARRIAGE RETURN)
*
*INITIAL CONDITIONS: STARTING ADDRESS
* OF STRING IN INDEX REGISTER X
*
*FINAL CONDITIONS: NUMBER OF CHARACTERS IN B
*
*REGISTERS AFFECTED: A,B,X,FLAGS
*
*SAMPLE CASE
* INITIAL CONDITIONS: (X) = 0042
* (0042) = 4D, (0043) = 41, (0044) = 4E, (0045) = 0D
* FINAL CONDITIONS: (B) = 03

```

This subroutine has a single parameter which is an address; Index Register X is the obvious place to put it. The result is returned in Accumulator B.

The calling program consists of three steps: placing the starting address of the string in Index Register X, calling the subroutine, and storing the result in memory. The overall initialization must also load the Hardware Stack Pointer with the appropriate value.

The subroutine is reentrant, since it does not use any fixed memory addresses for storage.

The subroutine changes Accumulator A as well as Accumulator B and Index Register X. The programmer must be aware that calling this subroutine destroys the contents of Accumulator A, even though it does not contain a parameter. The subroutine documentation must specify which registers are affected in order to avoid unforeseen side effects.

An alternative approach would be for the subroutine to save and restore the original contents of Accumulator A. The instruction PSHS A would save those contents initially and the instruction PULS A would restore them before the return. This approach takes extra time and memory, but makes the subroutine easier to use since it does not produce as many incidental changes. We could save and restore the condition code register as well by using the instructions PSHS A,CC and PULS A,CC.

If the terminating character were not always an ASCII carriage return, we could make that character into another parameter. Then the calling program would have to place the terminating character in Accumulator A before calling the subroutine.

10-3. MAXIMUM VALUE

Purpose: Find the largest element in an array of unsigned binary numbers. The length of the array (number of bytes) is in Accumulator B and the starting address of the array is in Index Register X. The maximum value is returned in Accumulator A.

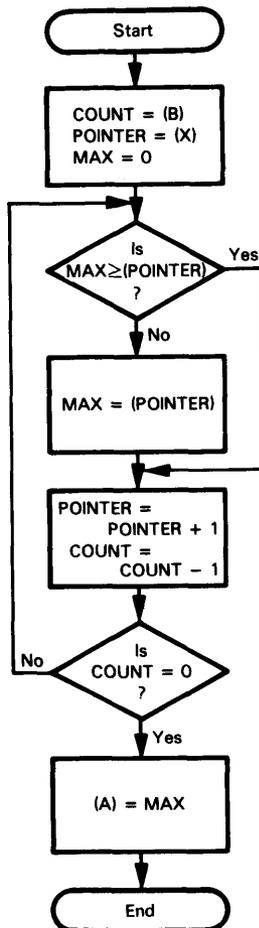
Sample Problem:

(B) = 05 Length of array (number of bytes)
 (X) = 0043 Starting address of array

(0043) = 67
 (0044) = 79
 (0045) = 15
 (0046) = E3
 (0047) = 72

Result: (A) = E3, since this is the largest of the five unsigned numbers in the array

Flowchart:



Program 10-3:

The calling program starts the Stack at memory location 00FF, sets the starting address of the array to 0043, gets the length of the array from memory location 0040, calls the maximum subroutine, and stores the maximum value in memory location 0041.

```

0000                                ORG    $0000
0000 10CE 0100                      LDS    #$0100    START STACK AT MEMORY LOCATION
                                *                                00FF
0004 8E    0043                      LDX    #$43    GET STARTING ADDRESS OF ARRAY
0007 D6    40                        LDB    $40    GET LENGTH OF ARRAY
0009 BD    0020                      JSR    MAXM   FIND MAXIMUM VALUE
000C 97    41                        STA    $41    SAVE MAXIMUM VALUE IN MEMORY
000E 3F                                SWI

```

The subroutine determines the maximum value in the array.

```

0020                                ORG    $0020
0020 4F                                *    CLRA    MAXIMUM = ZERO (MINIMUM POSSIBLE
                                *                                VALUE)
0021 A1    80                        *    CHKE   CMPA    ,X+    IS CURRENT ENTRY GREATER THAN
                                *                                MAXIMUM?
0023 24    02                        *    BCC   NOCHG
0025 A6    1F                        *    LDA   -1,X    YES, REPLACE MAXIMUM WITH
                                *                                CURRENT ENTRY
0027 5A                                *    NOCHG  DECB
0028 26    F7                        *    BNE   CHKE
002A 39                                *    RTS

```

Subroutine Documentation:

```

*SUBROUTINE MAXM
*
*PURPOSE: MAXM DETERMINES THE MAXIMUM VALUE IN AN ARRAY OF
* UNSIGNED BINARY NUMBERS
*
*INITIAL CONDITIONS: STARTING ADDRESS OF ARRAY IN INDEX REGISTER
* X, LENGTH OF ARRAY (NUMBER OF BYTES) IN ACCUMULATOR B
*
*FINAL CONDITIONS: MAXIMUM VALUE IN ACCUMULATOR A
*
*REGISTERS AFFECTED: A,B,X,FLAGS
*
*SAMPLE CASE:
* INITIAL CONDITIONS: 0043 IN INDEX REGISTER X, 03 IN
* ACCUMULATOR B, (0043) = 35, (0044) = 46, (0045) = 0D
* RESULT: (A) = 46

```

This subroutine has two parameters — an address and a number. Accumulator B is used to pass the number and Index Register X to pass the address. The result is a single number that is returned in Accumulator A.

The calling program must place the starting address of the array in Index Register X and the length of the array in Accumulator B before transferring control to the subroutine.

The subroutine is reentrant since it uses no fixed memory addresses and relocatable since it uses only relative branches.

We could retain the original contents of the condition code register by using the instructions PSHS CC and PULS CC.

This subroutine has some incidental effects: it changes the address in Index Register X (the final value is one beyond the last address in the array because of the autoincrementing) and it returns with zero in Accumulator B.

10-4. PATTERN MATCH

Purpose: Compare two strings of ASCII characters to see if they are the same. The length of the strings is in Accumulator B. The starting address of one string is in Index Register X and the starting address of the other string is in Index Register Y. If the two strings match, clear Accumulator B; otherwise, set Accumulator B to FF₁₆.

Sample Problems:

a.

(B)	=	03	Length of strings
(X)	=	0046	Starting address of string #1
(Y)	=	0050	Starting address of string #2
(0046)	=	43	'C'
(0047)	=	41	'A'
(0048)	=	54	'T'
(0050)	=	43	'C'
(0051)	=	41	'A'
(0052)	=	54	'T'

Result: (B) = 00 since the strings are the same

b.

(X)	=	0046	Starting address of #1
(Y)	=	0050	Starting address of string #2
(0046)	=	52	'R'
(0047)	=	41	'A'
(0048)	=	54	'T'
(0050)	=	43	'C'
(0051)	=	41	'A'
(0052)	=	54	'T'

Result: (B) = FF since the first characters differ

Program 10-4:

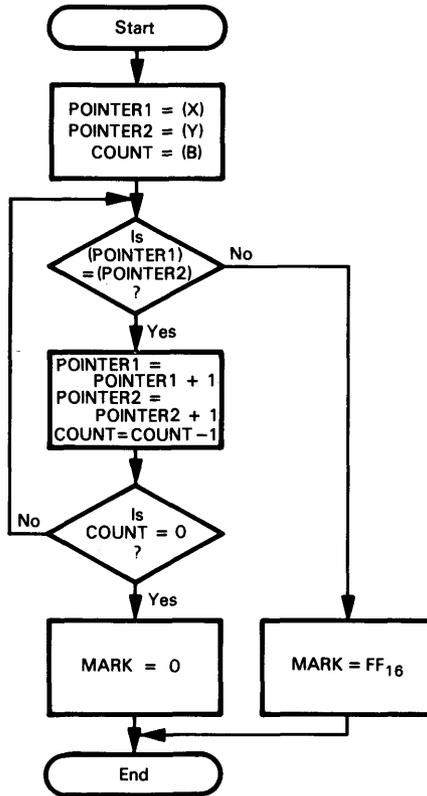
The calling program starts the Stack at memory location 00FF, sets the two starting addresses (Index Registers X and Y) to 0046 and 0050 respectively, gets the length of the string from memory location 0041, calls the pattern match subroutine, and places the result in memory location 0040.

0000			ORG	\$0000	
0000	10CE	0100	LDS	#\$0100	START STACK AT MEMORY LOCATION
		*			00FF
0004	8E	0046	LDX	#\$46	GET STARTING ADDRESS OF STRING 1
0007	108E	0050	LDY	#\$50	GET STARTING ADDRESS OF STRING 2
000B	D6	41	LDB	\$41	GET LENGTH OF STRINGS
000D	BD	0020	JSR	PMTCH	COMPARE STRINGS
0010	D7	40	STB	\$40	SAVE MATCH INDICATOR
0012	3F		SWI		

The subroutine determines if the two strings are the same.

0020			ORG	\$0020	
0020	A6	80	PMTCH LDA	,X+	GET A CHARACTER FROM STRING 1
0022	A1	A0	CMPLA	,Y+	IS THERE A MATCH WITH STRING 2?
0024	26	04	BNE	NOMCH	NO, DONE
0026	5A		DECB		ALL CHARACTERS CHECKED?
0027	26	F7	BNE	PMTCH	NO, CONTINUE
0029	39		RTS		YES, RETURN WITH INDICATOR =
		*			ZERO
002A	C6	FF	NOMCH LDB	#\$FF	NO MATCH, INDICATOR = FF HEX
			RTS		

Flowchart:



Subroutine Documentation:

```

*SUBROUTINE PMTCH
*
*PURPOSE: PMTCH DETERMINES IF TWO STRINGS ARE IDENTICAL
*
*INITIAL CONDITIONS: STARTING ADDRESSES OF STRINGS IN INDEX
* REGISTERS X AND Y, LENGTH OF STRINGS (IN BYTES) IN
* ACCUMULATOR B
*
*FINAL CONDITIONS: ZERO IN ACCUMULATOR B IF STRINGS MATCH,
* FF IN ACCUMULATOR B OTHERWISE
*
*REGISTERS AFFECTED: A,B,X,Y,FLAGS
*
*SAMPLE CASE:
* INITIAL CONDITIONS: (X) = 0046, (Y) = 0050, (B) = 02
* (0046) = 36, (0047) = 39
* (0050) = 35, (0051) = 39
* RESULT: (B) = 00 SINCE THE STRINGS ARE IDENTICAL
  
```

This subroutine, like the preceding examples, changes all the flags. You should generally assume that a subroutine call changes the flags unless it is specifically stated otherwise. If the main program needs the old flag values (for later checking), it must save them in the Stack (using PSHS CC) before calling the subroutine, and restore them afterward (using PULS CC).

This subroutine has three parameters — two starting addresses and the length of the strings. Two index registers (X and Y) are used for the starting addresses; Accumulator B is used for both the length of the strings and for the result. The subroutine changes Accumulator A incidentally.

The subroutine is reentrant, since it uses no fixed addresses.

Obviously, subroutines become far more complicated as soon as the number of parameters exceeds the number of registers. Using the registers is convenient, but it lacks generality; as soon as the number of parameters becomes large, you must use an entirely different approach.

Note that the subroutine has two exit points (i.e., two RTS instructions). This creates no problems, since either RTS terminates the subroutine and transfers control back to the main program.

10-5. MULTIPLE-PRECISION ADDITION

Purpose: Add two multi-byte binary numbers. The length of the numbers (in bytes) is in Accumulator B, the starting addresses of the numbers are in Index Registers X and Y, and the starting address of the result is in the User Stack Pointer U. All the numbers begin with the least significant bits.

Sample Problem:

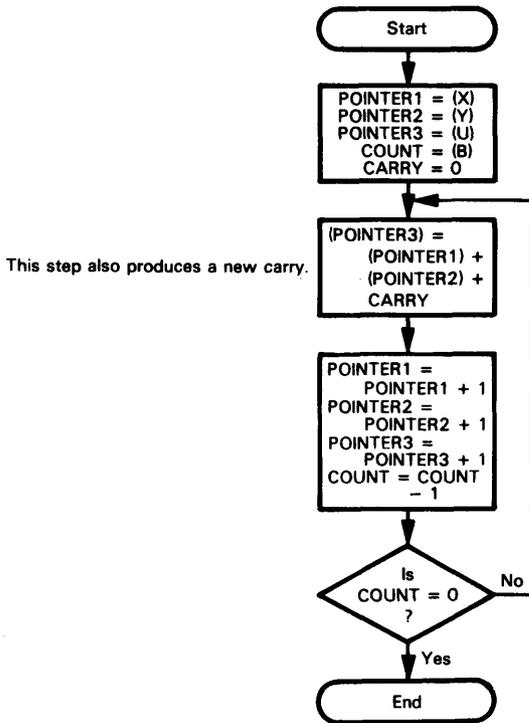
(B)	=	04	Length of numbers in bytes
(X)	=	0048	Starting address of first number
(Y)	=	004C	Starting address of second number
(U)	=	0050	Starting address of result
(0048)	=	C3	} 2F5BA7C3 ₁₆ is first number
(0049)	=	A7	
(004A)	=	5B	
(004B)	=	2F	
(004C)	=	B8	} 14DF35B8 ₁₆ is second number
(004D)	=	35	
(004E)	=	DF	
(004F)	=	14	
Result: (0050)	=	7B	} 443ADD7B ₁₆ is sum
(0051)	=	DD	
(0052)	=	3A	
(0053)	=	44	

Program 10-5:

The calling program starts the Stack at memory location 00FF, sets the starting addresses of the various numbers to 0048, 004C, and 0050, respectively, gets the length of the numbers (in bytes) from memory location 0040, and calls the multiple-precision addition subroutine.

0000		ORG	\$0000		
0000	10CE	0100	LDS	#\$0100	START STACK AT MEMORY LOCATION
					00FF
0004	8E	0048	LDX	#\$48	GET STARTING ADDRESS OF FIRST
					NUMBER
0007	108E	004C	LDY	#\$4C	GET STARTING ADDRESS OF SECOND
					NUMBER
000B	CE	0050	LDU	#\$50	GET STARTING ADDRESS OF SUM
000E	D6	40	LDB	\$40	GET LENGTH OF NUMBERS (IN BYTES)
0010	BD	0020	JSR	MPADD	PERFORM MULTIPLE-PRECISION
					ADDITION
0013	3F		SWI		

Flowchart:



The subroutine performs multiple-precision binary addition.

```

0020                ORG     $0020
0020 1C FE          MPADD  ANDCC  ##11111110  CLEAR CARRY TO START
0022 A6 80          ADDBYTE LDA   ,X+        GET BYTE FROM FIRST NUMBER
0024 A9 A0          ADCA   ,Y+        ADD BYTE FROM SECOND NUMBER
0026 A7 C0          STA   ,U+        STORE RESULT
0028 5A             DECB                ALL BYTES ADDED?
0029 26 F7          BNE   ADDBYTE NO, CONTINUE
002B 39             RTS
    
```

Subroutine Documentation:

```

*SUBROUTINE MPADD
*
*PURPOSE: MPADD ADDS TWO MULTI-BYTE BINARY NUMBERS
*
*INITIAL CONDITIONS: STARTING ADDRESSES OF NUMBERS (LSB'S) IN
* INDEX REGISTERS X AND Y, STARTING ADDRESS OF SUM IN USER
* STACK POINTER U, LENGTH OF NUMBERS (IN BYTES) IN ACCUMULATOR B
*
*REGISTERS AFFECTED: A,B,X,Y,U,FLAGS
*
*SAMPLE CASE:
* INITIAL CONDITIONS: (X) = 0048, (Y) = 004C, (U) = 0050,
* (B) = 02, (0048) = C3, (0049) = A7, (004C) = B8, (004D) = 35
* RESULT: (0050) = 7B, (0051) = DD
*
    
```

This subroutine has four parameters — three addresses and the length of the numbers. We use Index Register X, Index Register Y, User Stack Pointer U, and Accumulator B to pass them; no results are returned. User Stack Pointer U is really just

an extra index register. It is, in fact, somewhat more useful than Index Register Y since LDU and STU execute faster than LDY and STY. The reason for this difference is that LDU and STU require 1-byte operation codes, while LDY and STY require 2-byte operation codes. Note, however, that CMPU requires a 2-byte operation code, so U is slightly inferior to X. A further advantage of U which we will discuss shortly is the availability of the PSHU and PULU instructions, which can transfer an entire set of registers to or from the User Stack.

POSITION-INDEPENDENT CODE

Position-independent routines can be placed anywhere in memory without using a relocating loader and can be used with any combination of other programs. The keys to writing position-independent code are:

1. **Use relative branches (BSR, LBSR, BRA, LBRA), rather than JSR or JMP.**
2. **Refer to variables by means of the indexed addressing modes that use a constant offset from the Program Counter.** Remember that the assembler will calculate a relative offset for you if you specify the address as DEST, PCR. Thus the instruction

```
LDA    RDATA, PCR
```

will load Accumulator A from the relative address RDATA. You can use the indirect version to access data through addresses that are stored relatively.

3. **Use the Hardware Stack for temporary storage.** You can assign five Stack locations for temporary storage by subtracting five from the Hardware Stack Pointer with the instruction

```
LEAS  -5, S
```

You can then refer to these locations with indexed offsets and finally discard them with the instruction

```
LEAS  5, S
```

Note that such temporary storage locations are only allocated when the routine is actually executed (referred to as dynamic allocation); they need not be permanently assigned as fixed memory locations must be (referred to as static allocation). This use of the Hardware Stack for temporary storage also promotes reentrancy, since Stack locations are saved automatically when routines are interrupted or suspended.

If necessary, you can always determine the current value of the Program Counter by means of an instruction like

```
TFR    PC, X
```

which saves its absolute value (the address of the byte following the TFR instruction) in Index Register X. The program can thereby calculate its actual location in memory.

NESTED SUBROUTINES

The **BSR** and **JSR** instructions allow the nesting of subroutines, since subsequent subroutine calls will place their return addresses on top of the previous return addresses. No addresses are ever lost and an **RTS** instruction always returns control to the instruction just after the most recent **BSR** or **JSR**.

Jump and Link

We can use other methods to call one level of subroutine. For example, the instruction

```
EXG X, PC
```

loads the Program Counter with the previous contents of Index Register X and Index Register X with the previous contents of the Program Counter. This is equivalent to transferring control to the address in Index Register X, while saving the return address in that index register. However, this approach does not allow nesting, since Index Register X can only hold a single return address. Furthermore, it ties up Index Register X and makes the program rather difficult to follow. If you use this approach, remember that the instruction

```
EXG X, PC
```

at the end of the subroutine will transfer control back to the main program (as long as you have not disturbed Index Register X) and will save the address immediately following the **EXG** instruction in Index Register X. This approach is often referred to as **jump-and-link**, since it uses Index Register X as the link back to the main program.

PROBLEMS

Note that you are to write both a calling program for the sample problem and a properly documented subroutine.

10-1. CONVERT ASCII TO HEXADECIMAL

Purpose: Convert the contents of Accumulator A from the ASCII representation of a hexadecimal digit to the actual digit. Place the result in Accumulator A.

Sample Problems:

- | | | | | |
|----|---------|-----|----|-----|
| a. | (A) | = | 43 | 'C' |
| | Result: | (A) | = | 0C |
| b. | (A) | = | 36 | '6' |
| | Result: | (A) | = | 06 |

10-2. LENGTH OF A TELETYPEWRITER MESSAGE

Purpose: Determine the length of an ASCII-coded teletypewriter message. The starting address of the string of characters in which the message is embedded is in Index Register X. The message itself starts with an ASCII STX character (02_{16}) and ends with ASCII ETX (03_{16}). Place the length of the message (the number of characters between the STX and the ETX) in Accumulator B.

Sample Problem:

```
(X) = 0044 Starting address of string
(0044) = 49
(0045) = 02 STX
(0046) = 47 'G'
(0047) = 4F 'O'
(0048) = 03 ETX

Result: (B) = 02 since there are 2 characters between the ASCII
          STX and the ASCII ETX.
```

10-3. MINIMUM VALUE

Purpose: Find the smallest element in an array of 8-bit unsigned binary numbers. The length of the array (number of bytes) is in Accumulator B and the starting address of the array is in Index Register X. The minimum value is returned in Accumulator A.

Sample Problem:

```
(B) = 05 Length of array (number of bytes)
(X) = 0043 Starting address of array

(0043) = 67
(0044) = 79
(0045) = 15
(0046) = E3
(0047) = 73

Result: (A) = 15 since this is the smallest of the five
          unsigned numbers.
```

10-4. STRING COMPARISON

Purpose: Compare two strings of ASCII characters to see which is larger (i.e., which follows the other in 'alphabetical' ordering). The length of the strings is in Accumulator B. The starting address of string 1 is in Index Register X and the starting address of string 2 is in Index Register Y. If string 1 is larger than or equal to string 2, clear Accumulator B; otherwise, set Accumulator B to FF_{16} .

Sample Problems:

```
a. (B) = 03 Length of strings
    (X) = 0046 Starting address of string #1
    (Y) = 004A Starting address of string #2

    (0046) = 43 'C'
    (0047) = 41 'A'
    (0048) = 54 'T'

    (004A) = 42 'B'
    (004B) = 41 'A'
    (004C) = 54 'T'

Result: (B) = 00 since CAT is "larger" than BAT.
```

10-18 6809 Assembly Language Programming

- b.
- | | | | |
|--------|---|------|-------------------------------|
| (B) | = | 03 | Length of strings |
| (X) | = | 0046 | Starting address of string #1 |
| (Y) | = | 004A | Starting address of string #2 |
| (0046) | = | 44 | 'C' |
| (0047) | = | 41 | 'A' |
| (0048) | = | 54 | 'T' |
| (004A) | = | 44 | 'C' |
| (004B) | = | 41 | 'A' |
| (004C) | = | 54 | 'T' |
- Result: (B) = 00 since the two strings are the same
- c.
- | | | | |
|--------|---|------|-------------------------------|
| (B) | = | 03 | Length of strings |
| (X) | = | 0046 | Starting address of string #1 |
| (Y) | = | 004A | Starting address of string #2 |
| (0046) | = | 43 | 'C' |
| (0047) | = | 41 | 'A' |
| (0048) | = | 54 | 'T' |
| (004A) | = | 43 | 'C' |
| (004B) | = | 55 | 'U' |
| (004C) | = | 54 | 'T' |
- Result: (B) = FF since CUT is "larger" than CAT

10-5. DECIMAL SUBTRACTION

Purpose: Subtract one multi-digit decimal (BCD) number from another. The length of the numbers (in bytes) is in Accumulator B and the starting addresses of the numbers are in Index Registers X and Y. Subtract the number with the starting address in Index Register Y from the one with the starting address in Index Register X. The starting address of the result is in the user Stack Pointer U. All the numbers begin with the least significant digits. The sign of the result is returned in Accumulator B — zero if the result is positive, FF if it is negative.

Sample Problem:

- | | | | |
|-----|---|------|--------------------------------|
| (B) | = | 04 | Length of numbers in bytes |
| (X) | = | 0048 | Starting address of minuend |
| (Y) | = | 004C | Starting address of subtrahend |
| (U) | = | 0050 | Starting address of difference |
-
- | | | | |
|--------|---|----|-----------------------|
| (0048) | = | 85 | } 36701985 is minuend |
| (0049) | = | 19 | |
| (004A) | = | 70 | |
| (004B) | = | 36 | |
-
- | | | | |
|--------|---|----|--------------------------|
| (004C) | = | 59 | } 12663459 is subtrahend |
| (004D) | = | 34 | |
| (004E) | = | 66 | |
| (004F) | = | 12 | |
-
- Result: (B) = 00 Positive result
- | | | | |
|--------|---|----|----------------------------------|
| (0050) | = | 26 | } 24038526 is decimal difference |
| (0051) | = | 85 | |
| (0052) | = | 03 | |
| (0053) | = | 24 | |

11

Parameter Passing Techniques

In Chapter 10 we defined and briefly discussed parameters and the problem of transferring parameters to subroutines. The examples in Chapter 10 passed parameters through the 6809 registers; however, **in this chapter we will describe other, more general methods for passing parameters.** Since these parameter passing techniques make use of the 6809 stacks and stack pointers, **we will first discuss the important stack manipulation instructions PSH and PUL.**

THE PSH AND PUL INSTRUCTIONS

We have briefly mentioned the **PSH and PUL instructions** without fully exploring them. These instructions allow the programmer to **transfer sets of registers to and from the User Stack or the Hardware Stack.** Typical uses are to **transfer parameters to the Stack, transfer results from the Stack, and load or store a set of registers with one instruction.**

Each PSH or PUL instruction requires 2 bytes of program memory, one for the operation code and one to specify the list of registers that will be transferred to or from the Stack (either the User Stack or the Hardware Stack). The bits in the second byte of data determine whether particular registers will (if the assigned bit is 1) or will not (if the assigned bit is 0) be transferred to or from the Stack. Figure 11-1 shows how the bits are assigned and the order in which registers are pushed (stored on the stack) or pulled (loaded from the stack). Note that **neither Stack Pointer can be stored in or loaded from its own stack;** saving a Stack Pointer in its own stack would be like saving the key to a locked safe in the safe itself. The push order is, of course, the opposite of the pull order.

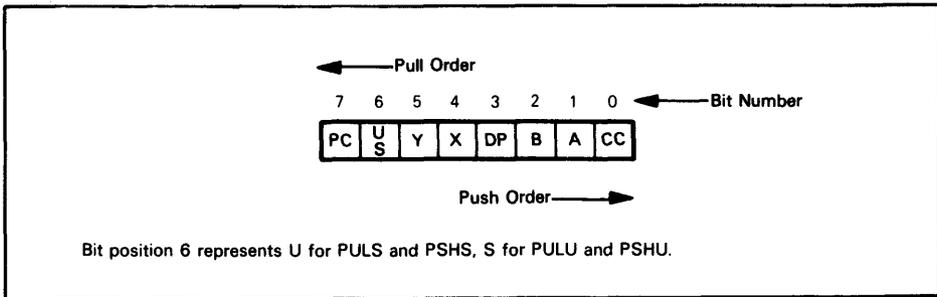


Figure 11-1. Assignment of Bits and Orders for PSH and PUL Instructions

The Stack grows downward, so the first registers pushed will end up at the highest addresses and the first registers pulled will come from the lowest addresses. 16-bit registers are pushed least significant byte first and pulled most significant byte first, thus maintaining compatibility with the standard 6809 method for storing 16-bit addresses or data. The 6809's Stack Pointers are decremented before each byte is stored and incremented after each byte is loaded.

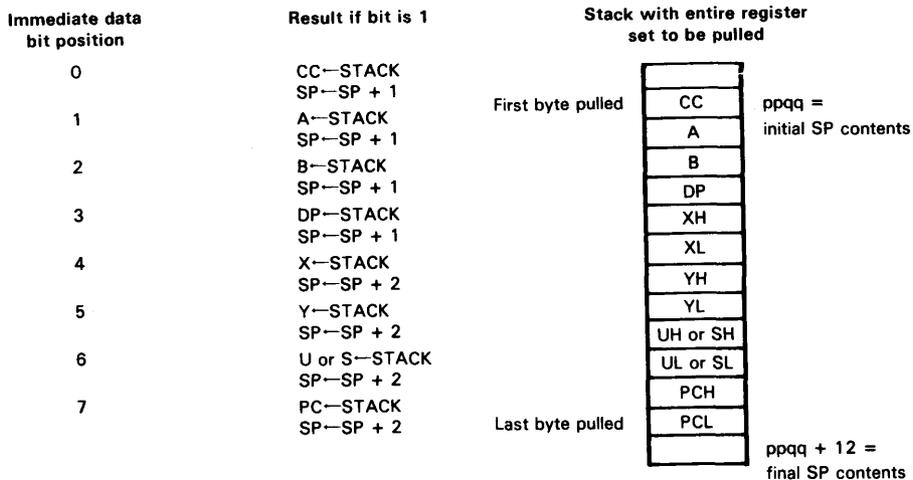
The result is that registers are pushed into either stack as follows:

Immediate data bit position	Result if bit is 1	Stack with entire register set pushed
7	SP ← SP - 2 STACK ← PC	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">Last byte pushed</div> <div style="border: 1px solid black; padding: 2px;">CC</div> <div style="margin-left: 10px;">ppqq - 12 = final SP contents</div> </div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">A</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">B</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">DP</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">XH</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">XL</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">YH</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">YL</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">UH or SH</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">UL or SL</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">PCH</div> <div style="border: 1px solid black; padding: 2px; margin: 2px;">PCL</div> <div style="margin-top: 10px;"> <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">First byte pushed</div> <div style="border: 1px solid black; padding: 2px; width: 40px; height: 20px;"></div> <div style="margin-left: 10px;">ppqq = initial SP contents</div> </div> </div>
6	SP ← SP - 2 STACK ← U or S	
5	SP ← SP - 2 STACK ← Y	
4	SP ← SP - 2 STACK ← X	
3	SP ← SP - 1 STACK ← DP	
2	SP ← SP - 1 STACK ← B	
1	SP ← SP - 1 STACK ← A	
0	SP ← SP - 1 STACK ← CC	

The description of PSH in Chapter 22 illustrates the result of stacking just two registers.

SP represents either the Hardware Stack Pointer (PSHS) or the User Stack Pointer (PSHU). Either PSH instruction can save any, all, any subset, or none of the user registers except its own pointer. The assembly language programmer simply provides a list of registers (in any order) in the operand field. The order in which registers are saved is a function of the hardware, not of the order in which the programmer specifies them.

The PULS or PULU instruction pulls the registers from the stack in the following order:



The description of PUL in Chapter 22 illustrates the result of unstacking just three registers.

PSH and PUL are particularly convenient when the entire state of a task must be saved or restored because the task has been suspended, preempted, or newly activated.

GENERAL PARAMETER PASSING TECHNIQUES^{1,2}

The registers often provide a fast, convenient way of passing parameters to subroutines and returning results. The limitations of this method are that it cannot be expanded beyond the number of registers, it often results in unforeseen side effects, and it lacks generality. **The tradeoff here is between fast execution time and a more general approach. Such a tradeoff is common in computer applications at all levels; general approaches are easy to learn, consistent, and can be automated through the use of compilers and other systems programs. On the other hand, approaches that take advantage of the specific features of a particular task require less time and memory. The choice of one approach or the other depends on your application, but you should take the general approach (saving programming time and simplifying documentation and maintenance) unless time or memory constraints force you to do otherwise.**

There are two general approaches to passing parameters:

1. Place the parameters (or arguments) immediately after the subroutine call.
2. Transfer the parameters and results on the Hardware Stack.

The first approach is convenient when the parameters are constants for a particular subroutine call, while the second approach is more general and is usually the choice made in writing interpreters, compilers, operating systems, and other systems programs.

USING ARGUMENT LISTS

In the first approach, the programmer follows each subroutine call with an appropriate list of parameters. The list itself must consist of constants if the program is to execute from ROM, although the constants may be the addresses of variable data or arrays. The programmer must **implement this approach as follows**:

1. **Use the DATA directives to store the parameters in program memory.** For the 6809 assembler, the directives are FCB for byte-length data, FDB for 16-bit data or addresses, and FCC for character data.
2. **Access the data by means of the return address** that the JSR or BSR instruction stores at the top of the Hardware Stack. The return address will actually be the starting address of the list of parameters. You can access the first element of the list indirectly with an instruction like

```
LDA    [ ,S]
```

or you can load the starting address into an Index Register (U, for example) with an instruction like

```
LEAU   [ ,S]
```

3. **Adjust the return address so that it points to the next executable instruction.** That is, add the length of the parameter list to the actual return address so that the processor does not accidentally try to execute the subroutine parameters. If the return address is in the User Stack Pointer U and the parameters occupy 5 bytes of program memory, the sequence

```
LEAU   S,U           MOVE RETURN ADDRESS PAST PARAMETERS
STU    ,S            SAVE ADJUSTED RETURN ADDRESS IN STACK
RTS
```

will return control to the next executable instruction.

EXAMPLES

11-1a. LENGTH OF A STRING OF CHARACTERS

Purpose: Determine the length of a string of ASCII characters. The terminating character and the starting address of the string follow the subroutine call. The length of the string (excluding the terminating character) is returned in Accumulator B. No other registers are affected.

Sample Problems:

a. The subroutine call is followed by:

FCB	\$0D	TERMINATING CHARACTER
FDB	\$43	STARTING ADDRESS OF STRING
	(0043)	= 52 'R'
	(0044)	= 41 'A'
	(0045)	= 54 'T'
	(0046)	= 48 'H'
	(0047)	= 45 'E'
	(0048)	= 52 'R'
	(0049)	= 0D CR

Result: (B) = 06

b. The subroutine call is followed by:

```
FCB  $0D      TERMINATING CHARACTER
FDB  $43      STARTING ADDRESS OF STRING
```

```
(0043) = 0D CR
```

```
Result: (B) = 00
```

Program 11-1a:

The calling program starts the Stack at memory location 00FF, calls the string length subroutine (specifying the terminator and starting address in the next three bytes), and stores the result in memory location 0042.

```
0000          ORG  $0000
0000 10CE 0100  LDS  #$100  START STACK AT MEMORY LOCATION
                    *      00FF
0004 BD  0020  JSR  STLEN  DETERMINE STRING LENGTH
0007 0D          FCB  $0D   STRING TERMINATOR
0008 0043       FDB  $43   STARTING ADDRESS OF STRING
000A D7  42     STB  $42   SAVE STRING LENGTH
000C 3F          SWI

                    *
                    *
0020          ORG  $0020
0020 34  53     STLEN PSHS  U,X,A,CC SAVE REGISTERS
0022 EE  66     LDU   6,S   ACCESS PARAMETER LIST
0024 37  12     PULU  A,X   GET STRING TERMINATOR,
0026 C6  FF     LDB  #$FF   STARTING ADDRESS
0028 5C          CHKTRM INCB  ADD 1 TO STRING LENGTH
0029 A1  80     CMPA  ,X+   IS NEXT CHARACTER A TERMINATOR?
002B 26  FB     BNE  CHKTRM NO, KEEP LOOKING
002D EF  66     STU   6,S   MOVE RETURN ADDRESS PAST
                    *      PARAMETER LIST
002F 35  D3     *      PULS  PC,U,X,A,CC RESTORE REGISTERS AND
                    *      RETURN
```

Subroutine Documentation:

```
*SUBROUTINE STLEN
*
*PURPOSE: STLEN DETERMINES THE LENGTH OF A STRING (NUMBER OF
* CHARACTERS PRECEDING A TERMINATOR)
*
*INITIAL CONDITIONS: TERMINATOR IN BYTE IMMEDIATELY FOLLOWING
* SUBROUTINE CALL, STARTING ADDRESS OF STRING IN NEXT TWO
* BYTES (MSB'S IN FIRST BYTE)
*
*FINAL CONDITIONS: NUMBER OF CHARACTERS IN B
*
*REGISTERS AFFECTED: B
*
*SAMPLE CASE:
* INITIAL CONDITIONS: TERMINATOR = 0D, STARTING ADDRESS = 0042
* (0042) = 4D, (0043) = 41, (0044) = 4E, (0045) = 0D
* FINAL CONDITIONS: (B) = 03
*
*TYPICAL CALL:
*      JSR  STLEN
*      FCB  TERM      TERMINATOR
*      FDB  START    STARTING ADDRESS OF STRING
*
```

The parameters follow the subroutine call in memory. We are mixing instructions and assembler directives, a practice that is acceptable as long as the processor never accidentally executes anything that is not an instruction. The result of the JSR instruction is:

11-6 6809 Assembly Language Programming

```
((S)-1) = (00FF)-(PCL) = 07
((S)-2) = (00FE)-(PCH) = 00
(S)-(S) - 2 = 00FE
```

The subroutine begins by storing all the incidental registers that it uses in the Stack with PSHS. The result is:

```
((S)-1) = (00FD)-(UL)
((S)-2) = (00FC)-(UH)
((S)-3) = (00FB)-(XL)
((S)-4) = (00FA)-(XH)
((S)-5) = (00F9)-(A)
((S)-6) = (00F8)-(CC)
(S)-(S) - 6 = 00FE - 6 = 00F8
```

Now the instruction LDU 6,S loads the return address from memory locations 00FE and 00FF into the User Stack Pointer.

```
(U) - ((S)+6):((S)+7) = (00F8+6):(00F8+7) = (00FE):(00FF) = 0007
```

The instruction PULU A,X loads the parameters into Accumulator A (the terminating character) and Index Register X. Note that the order of the parameters is critical — it must be the same as the pulling order of PULU.

```
(A)-(U) = (0007) = 0D
(XH)-(U) + 1 = (0008) = 00
(XL)-(U) + 2 = (0009) = 43
(U)-(U) + 3 = 000A
```

Not only does PULU load all the parameters into the registers, but it also adjusts the return address to the end of the parameter list.

After the length of the string has been determined in the same way as before, the instruction STU 6,S saves the adjusted return address in the Hardware Stack.

```
((S) + 6) = (00F8 + 6) = (00FE)-(UH) = 00
((S) + 7) = (00F8 + 7) = (00FF)-(UL) = 0A
```

Finally PULS PC,X,U,A,CC restores all the registers and transfers control back to the main program. No RTS instruction is necessary.

```
(CC)-(S) = (00F8)
(A)-(S)+1 = (00F9)
(XH)-(S)+2 = (00FA)
(XL)-(S)+3 = (00FB)
(UH)-(S)+4 = (00FC)
(UL)-(S)+5 = (00FD)
(PCH)-(S)+6 = (00FE) = 00
(PCL)-(S)+7 = (00FF) = 0A
(S)-(S) + 8 = 00F8 + 8 = 0100
```

Obviously the programming here is a great deal more complex and harder to understand than in the earlier version, Program 10-2. However, this version is reentrant, general, has no incidental side effects, and allows simple variation of the starting address and terminating character in different calls. Other parameters that we could add easily include a limiting number of characters (the maximum number that the routine will examine), an error exit (in the event that the processor does not find a terminating character), a starting character, and a memory address in which to store the result. You might try to expand the routine in a general way to include some or all of these parameters.

11-2a. MULTIPLE-PRECISION ADDITION

Purpose: Add two multi-byte binary numbers. The starting addresses of the numbers and the result, as well as the length of the numbers in bytes, follow the subroutine call. No registers or flags are affected.

Sample Problem:

The subroutine call is followed by:

	FCB	4	LENGTH OF STRINGS (IN BYTES)
	FDB	\$48	ADDRESS OF LSB'S OF 1ST NUMBER
	FDB	\$4C	ADDRESS OF LSB'S OF 2ND NUMBER
	FDB	50	ADDRESS OF LSB'S OF SUM
	(0048)	= C3	
	(0049)	= A7	
	(004A)	= 5B	2F5BA7C3 ₁₆ is first number
	(004B)	= 2F	
	(004C)	= B8	
	(004D)	= 35	14DF35B8 ₁₆ is second number
	(004E)	= DF	
	(004F)	= 14	
Result:	(0050)	= 7B	
	(0051)	= DD	
	(0052)	= 3A	443ADD7B ₁₆ is sum
	(0053)	= 44	

Program 11-2a:

The calling program starts the Stack at memory location 00FF and calls the multiple-precision addition subroutine, specifying the length (in bytes) and the starting addresses of the operands and sum in the next seven bytes.

0000		ORG	\$0000	
0000	10CE 0100	LDS	#\$100	START STACK AT MEMORY LOCATION
		*		00FF
0004	BD 0020	JSR	MPADD	PERFORM MULTIPLE-PRECISION
		*		ADDITION
0007	04	FCB	4	LENGTH OF STRINGS (IN BYTES)
0008	0048	FDB	\$48	ADDRESS OF LSB'S OF FIRST NUMBER
000A	004C	FDB	\$4C	ADDRESS OF LSB'S OF SECOND
		*		NUMBER
000C	0050	FDB	\$50	ADDRESS OF LSB'S OF SUM
000E	3F	SWI		
		*		
		*		
0020		ORG	\$0020	
0020	34 77	MPADD	PSHS	X,Y,U,A,B,CC SAVE ALL REGISTERS
0022	EE 69	LDU	9,S	ACCESS PARAMETER LIST
0024	37 34	PULU	X,Y,B	GET LENGTH, ADDRESSES OF
		*		OPERANDS
0026	EE C4	LDU	,U	GET ADDRESS OF SUM
0028	1C FE	ANDCC	##11111110	CLEAR CARRY TO START
002A	A6 80	ADDBYTE	LDA	,X+
002C	A9 A0	ADCA	,Y+	ADD BYTE FROM SECOND NUMBER
002E	A7 C0	STA	,U+	STORE RESULT
0030	5A	DECB		ALL BYTES ADDED?
0031	26 F7	BNE	ADDBYTE	NO, CONTINUE
0033	EE 69	LDU	9,S	ADJUST RETURN ADDRESS PAST
0035	33 47	LEAU	7,U	ARGUMENT LIST
0037	EF 69	STU	9,S	
0039	35 F7	PULS	PC,U,Y,X,B,A,CC	RESTORE REGISTERS AND
		*		RETURN

Subroutine Documentation:

```

*SUBROUTINE MPADD
*
*PURPOSE: MPADD ADDS TWO MULTI-BYTE BINARY NUMBERS
*
*INITIAL CONDITIONS: SUBROUTINE CALL IS FOLLOWED BY LENGTH OF
* STRINGS (IN BYTES), STARTING ADDRESSES OF LSB'S OF OPERANDS,
* AND STARTING ADDRESS OF LSB'S OF SUM
*
*REGISTERS AFFECTED: NONE
*
*SAMPLE CASE:
* INITIAL CONDITIONS: LENGTH = 02, OPERAND ADDRESSES = 0048 AND
* 004C,
* ADDRESS OF SUM = 0050
* (0048) = C3, (0049) = A7, (004C) = B8, (004D) = 35
* RESULT: (0050) = 7B, (0051) = DD (A7C3 + 35B8 = DD7B)
*
*TYPICAL CALL:
* JSR MPADD
* FCB LNTH LENGTH OF STRINGS (IN BYTES)
* FDB OPER1 STARTING ADDRESS (LSB'S) OF OPERAND 1
* FDB OPER2 STARTING ADDRESS (LSB'S) OF OPERAND 2
* FDB SUM STARTING ADDRESS (LSB'S) OF SUM
    
```

The only new problem here is that we cannot pull U from its own stack and we are very reluctant to change S (since it is used automatically in interrupts as we shall see in Chapter 15). So we must tiptoe around this limitation, retaining reentrancy as follows:

1. PULU X,Y,B loads the length of the numbers into Accumulator B and the starting addresses of the operands into Index Registers X and Y, respectively.
2. LDU ,U loads the starting address of the result into the User Stack Pointer U.
3. The ending sequence

```

LDU 9,S
LEAU 7,U
STU 9,S
    
```

adds 7 to the return address stored in the Stack, so that it now points to the address immediately following the list of arguments.

PASSING PARAMETERS ON THE STACK

In the second approach, all parameters and results are passed in the Hardware Stack. Here the parameters can be variables, since they are placed in RAM, not in ROM. The programmer must implement this approach as follows:

1. Use the LEAS instruction to decrement the Hardware Stack Pointer to leave room for results on the Hardware Stack.
2. Use the PSHS instruction to save all the parameters on the Hardware Stack.
3. Access the parameters by means of indexed offsets from the Hardware Stack Pointer, remembering that JSR or BSR places the return address at the top of the Stack. The User Stack Pointer can be used to remove many parameters at once.
4. Access the results by means of indexed offsets from the Hardware Stack Pointer. Again, the User Stack Pointer can be used to store many results at one time.
5. Clean up the stack after returning from the subroutine, so that the parameters are removed and the results are handled appropriately.

11-1b. LENGTH OF A STRING OF CHARACTERS

Purpose: Determine the length of a string of ASCII characters. The starting address of the string and the terminating character are placed in the Hardware Stack. The length of the string (excluding the terminating character) is returned at the top of the Hardware Stack. No registers are affected.

Sample Problems:

a. The subroutine call occurs with the top of the Hardware Stack containing:

	0D	String terminator
	00	MSBs of starting address of string
	43	LSBs of starting address of string
	empty byte	"Hole" for length of string
(0043)	= 52	'R'
(0044)	= 41	'A'
(0045)	= 54	'T'
(0046)	= 48	'H'
(0047)	= 45	'E'
(0048)	= 52	'R'
(0049)	= 0D	CR

Result: The top of the Hardware Stack contains:

	0D	String terminator
	00	MSBs of starting address of string
	43	LSBs of starting address of string
	06	Length of string (in bytes)

b. The subroutine call occurs with the top of the Hardware Stack containing:

	0D	String terminator
	00	MSBs of starting address of string
	43	LSBs of starting address of string
	empty byte	"Hole" for length of string
(0043)	= 0D	CR

Result: The top of the Hardware Stack contains:

	0D	String terminator
	00	MSBs of starting address of string
	43	LSBs of starting address of string
	00	Length of string (in bytes)

Program 11-1b:

The calling program starts the stack at memory location 00FF, leaves an empty byte on the stack for the string length, stores the terminator and starting address on the stack, calls the string length subroutine, removes the parameters from the stack (by incrementing the Hardware Stack Pointer), loads the string length from the stack, and stores the string length in memory location 0042.

0000		ORG	\$0000	
0000	10CE 0100	LDS	#\$100	START STACK AT MEMORY LOCATION
	*			00FF
0004	32 7F	LEAS	-1,S	LEAVE ROOM FOR LENGTH OF STRING
0006	86 0D	LDA	#\$0D	GET TERMINATOR
0008	8E 0043	LDX	#\$43	GET STARTING ADDRESS OF STRING
000B	34 12	PSHS	A,X	SAVE PARAMETERS IN HARDWARE
	*			STACK

11-10 6809 Assembly Language Programming

```

000D BD 0020      JSR  STLEN  DETERMINE STRING LENGTH
0010 32 63      LEAS 3,S    REMOVE PARAMETERS FROM STACK
0012 35 02      PULS A      GET STRING LENGTH FROM STACK
0014 97 42      STA $42   SAVE STRING LENGTH
0016 3F          SWI

*
*
0020            ORG  $0020
0020 34 57      STLEN PSHS  U,X,B,A,CC  SAVE REGISTERS
0022 33 69      LEAU 9,S    ACCESS PARAMETER LIST IN STACK
0024 37 12      PULU A,X    GET STRING TERMINATOR,
*                STARTING ADDRESS
0026 C6 FF      LDB  #$FF   STRING LENGTH = -1
0028 5C          CHKTRM INCB   ADD 1 TO STRING LENGTH
0029 A1 80      CMPA ,X+   IS NEXT CHARACTER A TERMINATOR?
002B 26 FB      BNE  CHKTRM NO, KEEP LOOKING
002D E7 C4      STB  ,U    SAVE STRING LENGTH IN STACK
002F 35 D7      PULS PC,X,U,B,A,CC RESTORE REGISTERS AND
*                RETURN

```

Subroutine Documentation:

```

*
*SUBROUTINE STLEN
*
*PURPOSE: STLEN DETERMINES THE LENGTH OF A STRING (NUMBER OF
* CHARACTERS PRECEDING A TERMINATOR)
*
*INITIAL CONDITIONS: TERMINATOR ON TOP OF STACK, FOLLOWED BY
* STARTING ADDRESS OF STRING AND AN EMPTY BYTE FOR THE STRING
* LENGTH
*
*FINAL CONDITIONS: STRING LENGTH ON STACK UNDER PARAMETERS
*
*REGISTERS AFFECTED: NONE
*
*SAMPLE CASE:
* INITIAL CONDITIONS: TERMINATOR = 0D, STARTING ADDRESS = 0042
* (0042) = 4D, (0043) = 41, (0044) = 4E, (0045) = 0D
* FINAL CONDITIONS: STRING LENGTH = 03
*
*TYPICAL CALL:
*
*      LEAS -1,S    LEAVE EMPTY BYTE FOR LENGTH OF STRING
*      LDA #TERM   STRING TERMINATOR
*      LDX #START  STARTING ADDRESS OF STRING
*      PSHS A,X    SAVE PARAMETERS IN STACK
*      JSR STLEN  DETERMINE STRING LENGTH
*

```

Here the idea is to leave space for the results on the stack, store the parameters on top of that space, call the subroutine, save the registers, use the parameters to calculate the results, save the results on the Stack, restore the registers, return to the main program, clear the parameters from the stack by increasing the Stack Pointer, and remove the results from the top of the stack.

LEAS -1,S leaves one location in the Stack for the length of the string. The result is:

$$((S)-(S) - 1 = 0100 - 1 = 00FF)$$

The processor does not store anything in the extra stack location. PSHS A,X stores the parameters in the Hardware Stack. The result is:

$$\begin{aligned}
 ((S)-1) &= (00FE)-(XL) = 43 \\
 ((S)-2) &= (00FD)-(XH) = 00 \\
 ((S)-3) &= (00FC)-(A) = 0D \\
 (S)-(S) - 3 &= 00FC
 \end{aligned}$$

JSR STLEN transfers control to the subroutine and saves the return address (0010) at the top of the Stack. The result is:

```
((S)-1) = (00FB)--(PCL) = 10
((S)-2) = (00FA)--(PCH) = 00
(S)--(S) - 2 = 00FC - 2 = 00FA
```

PSHS U,X,B,A,CC saves all the incidental registers in the Hardware Stack. The result is:

```
((S)-1) = (00F9)--(UL)
((S)-2) = (00F8)--(UH)
((S)-3) = (00F7)--(XL)
((S)-4) = (00F6)--(XH)
((S)-5) = (00F5)--(B)
((S)-6) = (00F4)--(A)
((S)-7) = (00F3)--(CC)
(S)--(S) - 7 = 00FA - 7 = 00F3
```

LEAU 9,S loads the User Stack Pointer with the starting address of the list of parameters.

```
(U)--(S) + 9 = 00F3 + 9 = 00FC
```

PULU A,X loads the parameters into Accumulator A (the terminating character) and Index Register X (the starting address of the string).

```
(A)--(U) = (00FC) = 0D
(XH)--(U)+1 = (00FD) = 00
(XL)--(U)+2 = (00FE) = 43
(U)--(U) + 3 = 00FC + 3 = 00FF
```

STB ,U stores the length of the string in the "hole" in the stack.

```
((U)) = (00FF)--(B)
```

PULS PC,X,U,B,A,CC restores all the incidental registers and transfers control back to the main program.

```
(CC)--(S) = (00F3)
(A)--((S)+1) = (00F4)
(B)--((S)+2) = (00F5)
(XH)--((S)+3) = (00F6)
(XL)--((S)+4) = (00F7)
(UH)--((S)+5) = (00F8)
(UL)--((S)+6) = (00F9)
(PCH)--((S)+7) = (00FA) = 00
(PCL)--((S)+8) = (00FB) = 10
(S)--(S)+9 = 00F3 + 9 = 00FC
```

Back in the main program, LEAS 3,S cleans the stack, essentially removing all the parameters.

```
(S)--(S)+3 = 00FC + 3 = 00FF
```

Finally PULS A removes the result (the length of the string) from the Hardware Stack.

```
(A)--(S) = (00FF)
(S)--(S)+1 = 00FF + 1 = 0100
```

Here again the programming is more complex and harder to understand than in our initial simple version, but this version is also reentrant, general, has no incidental side effects, and allows simple variation of parameters and generalization.

11-2b. MULTIPLE-PRECISION ADDITION

Purpose: Add two multi-byte binary numbers. The starting addresses of the numbers and the result, as well as the length of the numbers in bytes, are on the Hardware Stack. The starting address of the result ends up at the top of the Hardware Stack. No registers or flags are affected.

Sample Problem:

The subroutine call occurs with the top of the Hardware Stack containing:

04	Length of strings (in bytes)	
00 48	Starting address of operand 1	}
00 4C	Starting address of operand 2	}
00 50	Starting address of sum	}
(0048) = C3	} 2F5BA7C3 ₁₆ is first number	}
(0049) = A7		
(004A) = 5B		
(004B) = 2F		
(004C) = B8	} 14DF35B8 ₁₆ is second number	}
(004D) = 35		
(004E) = DF		
(004F) = 14		
Result: (0050) = 7B	} 443ADD7B ₁₆ is sum	}
(0051) = DD		
(0052) = 3A		
(0053) = 44		

The Hardware Stack is unchanged.

Program 11-2b:

The calling program starts the stack at memory location 00FF, stores the starting addresses of the strings and the length in the stack, calls the multiple-precision addition subroutine, removes the parameters from the stack (by increasing the Hardware Stack Pointer), loads the starting address of the sum from the stack, and stores the starting address in memory locations 0040 and 0041.

0000	ORG	\$0000	
0000 10CE 0100	LDS	#\$100	START STACK AT MEMORY LOCATION
	*		00FF
0004 CE 0050	LDU	#\$50	GET STARTING ADDRESS OF RESULT
0007 8E 0048	LDX	#\$48	GET STARTING ADDRESSES OF
	*		OPERANDS
000A 108E 004C	LDY	#\$4C	
000E 86 04	LDA	#4	GET LENGTH OF STRINGS
0010 34 72	PSHS	U, Y, X, A	SAVE PARAMETERS IN HARDWARE
	*		STACK
0012 BD 0020	JSR	MPADD	PERFORM MULTIPLE-PRECISION
	*		ADDITION

```

0015 32 65          LEAS  5,S    REMOVE PARAMETERS FROM STACK
0017 35 10          PULS  X      GET ADDRESS OF RESULT
0019 9F 40          STX   $40    SAVE ADDRESS OF RESULT IN MEMORY
001B 3F

*
*
0020              ORG   $0020
0020 34 77          MPADD PSHS  U,Y,X,B,A,CC  SAVE REGISTERS
0022 33 6B          LEAU  11,S    ACCESS PARAMETER LIST IN STACK
0024 37 34          PULU  X,Y,B   GET LENGTH, ADDRESSES OF
                                OPERANDS
0026 EE C4          LDU   ,U      GET STARTING ADDRESS OF RESULT
0028 1C FE          ANDCC #11111110 CLEAR CARRY TO START
002A A6 80          ADBYTE LDA  ,X+  GET BYTE FROM FIRST NUMBER
002C A9 A0          ADCA  ,Y+    ADD BYTE FROM SECOND NUMBER
002E A7 C0          STA   ,U+    STORE RESULT
0030 5A             DECB             ALL BYTES ADDED?
0031 26 F7          BNE  ADBYTE    NO, CONTINUE
0033 35 F7          PULS  PC,U,Y,X,B,A,CC  RESTORE REGISTERS AND
                                RETURN
*

```

Subroutine Documentation:

```

*
*SUBROUTINE MPADD
*
*PURPOSE: MPADD ADDS TWO MULTI-BYTE BINARY NUMBERS
*
*INITIAL CONDITIONS: LENGTH OF STRINGS (IN BYTES) ON TOP OF
* STACK, FOLLOWED BY STARTING ADDRESSES OF LSB'S OF OPERANDS
* AND STARTING ADDRESS OF LSB'S OF SUM
*
*REGISTERS AFFECTED: NONE
*
*SAMPLE CASE:
* INITIAL CONDITIONS: LENGTH = 02, OPERAND ADDRESSES = 0048
* AND 004C,
* ADDRESS OF SUM = 0050
* (0048) = C3, (0049) = A7, (004C) = B8, (004D) = 35
* RESULT: (0050) = 7B, (0051) = DD (A7C3 + 35B8 = DD7B)
*
*TYPICAL CALL:
* LDX #OPER1 STARTING ADDRESS (LSB'S) OF OPERAND 1
* LDY #OPER2 STARTING ADDRESS (LSB'S) OF OPERAND 2
* LDU #SUM STARTING ADDRESS (LSB'S) OF SUM
* LDA #LENGTH LENGTH OF STRINGS (IN BYTES)
* PSHS U,Y,X,A SAVE PARAMETERS IN HARDWARE STACK
* JSR MPADD PERFORM MULTIPLE-PRECISION ADDITION
*

```

TYPES OF PARAMETERS

Regardless of our approach to passing parameters, we can specify the parameters in a variety of ways. For example, we can:

1. Place the actual values in the parameter list. We can use immediate addressing or DATA directives and retrieve the data, if necessary, by using indexed offsets. This method is sometimes referred to as *call-by-value*, since only the values of the parameters are of concern.
2. Place the addresses of the parameters in the parameter list. We can use address-length registers or retrieve the data by using the indexed indirect modes. This method is sometimes referred to as *call-by-name*, since we are concerned with the locations of the parameters as well as their values.

REFERENCES

1. C. W. Gear. *Computer Organization and Programming*, 3rd ed., McGraw-Hill, New York, 1980, Chapter 4.
2. S. Mazor and C. Pitchford. "Develop Cooperative Microprocessor Subroutines," *Electronic Design*, June 7, 1978, pp. 116-118. Examples are for the 8080 microprocessor.

12

Input/Output

There are two problems in the design of input/output routines: one is how to interface peripherals to the computer and transfer data, status, and control signals; the other is how to address I/O devices so that the CPU can select a particular one for a data transfer. Clearly, the first problem is both more complex and more interesting. We will therefore discuss the interfacing of peripherals here and leave addressing to a more hardware-oriented book.

I/O AND MEMORY

In theory, the transfer of data to or from an I/O device is similar to the transfer of data to or from memory. In fact, we can consider the memory as just another I/O device. The memory is, however, special for the following reasons:

1. It operates at almost the same speed as the processor.
2. It uses the same type of signals as the CPU. The only circuits usually needed to interface the memory to the CPU are drivers, receivers, and level translators.
3. It requires no special formats or any control signals besides a Read/Write pulse.
4. It automatically latches data sent to it.
5. Its word length is the same as the computer's.

Most I/O devices do not have such convenient features. They may operate at speeds much slower than the processor; for example, a teletypewriter can transfer only 10 characters per second, while a slow processor can transfer 10,000 characters per sec-

ond. **The range of speeds is also very wide** — sensors may provide one reading per minute, while video displays or floppy disks may transfer 250,000 bits per second. Furthermore, **I/O devices may require continuous signals** (motors or thermometers), **currents rather than voltages** (teletypewriters), or **voltages at far different levels than the signals used by the processor** (gas-discharge displays). I/O devices may also require special formats, protocols, or control signals. Their word lengths may be much shorter or much longer than the word length of the computer. **These variations make the design of I/O routines difficult and mean that each peripheral presents its own special interfacing problem.**

I/O DEVICE CATEGORIES

We may, however, provide a general description of devices and interfacing methods. We may roughly separate devices into three categories, based on their data rates:

1. **Slow devices that change state no more than once per second.** Changing their states typically requires milliseconds or longer. Such devices include lighted displays, switches, relays, and many mechanical sensors and actuators.
2. **Medium-speed devices that transfer data at rates of 1 to 10,000 bits per second.** Such devices include keyboards, printers, card readers, paper tape readers and punches, cassettes, ordinary communications lines, and many analog data acquisition systems.
3. **High-speed devices that transfer data at rates of over 10,000 bits per second.** Such devices include magnetic tapes, magnetic disks, high-speed line printers, high-speed communications lines, and video displays.

INTERFACING SLOW DEVICES

The interfacing of slow devices is simple. Few control signals are necessary unless the devices are multiplexed, that is, several are handled from one port, as shown in Figures 12-1 to 12-4. Input data from slow devices need not be latched, since it remains stable for a long time interval. Output data must, of course, be latched. The only problems with input are transitions that occur while the computer is reading the data. One-shots, cross coupled latches, or software delay routines can smooth the transitions.

A single port can handle several slow devices. Figure 12-1 shows a demultiplexer that automatically directs the next output data to the next device by counting output operations. Figure 12-2 shows a control port that provides select inputs to a demultiplexer. The data outputs here can come in any order, but an additional output instruction is necessary to change the state of the control port. Output demultiplexers are commonly used to drive several displays from the same output port. Figures 12-3 and 12-4 show the same alternatives for an input multiplexer.

Note the differences between input and output with slow devices.

1. **Input data need not be latched** since the input device holds the data for an enormous length of time by computer standards. Output data must be latched since the output device will not respond to data that is present for only a few

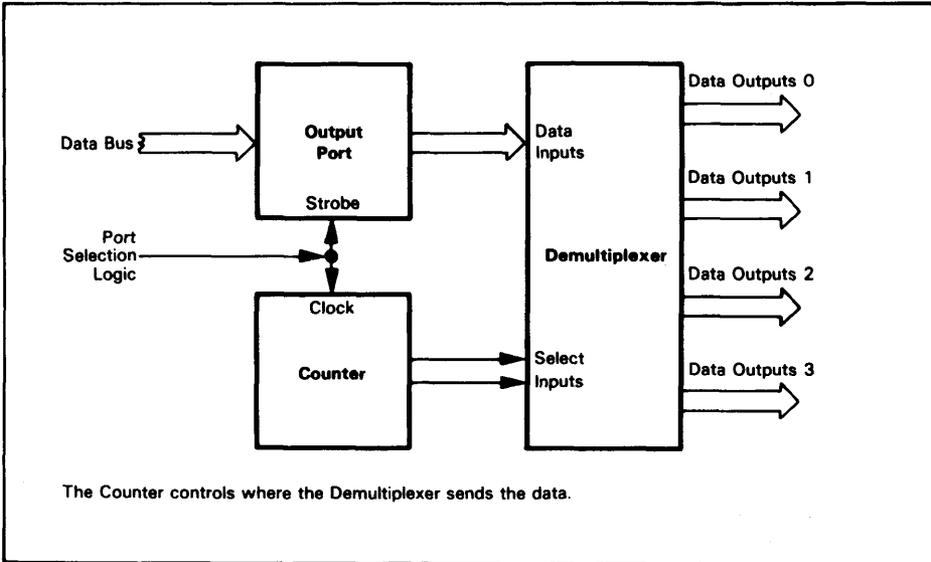


Figure 12-1. An Output Demultiplexer Controlled by a Counter

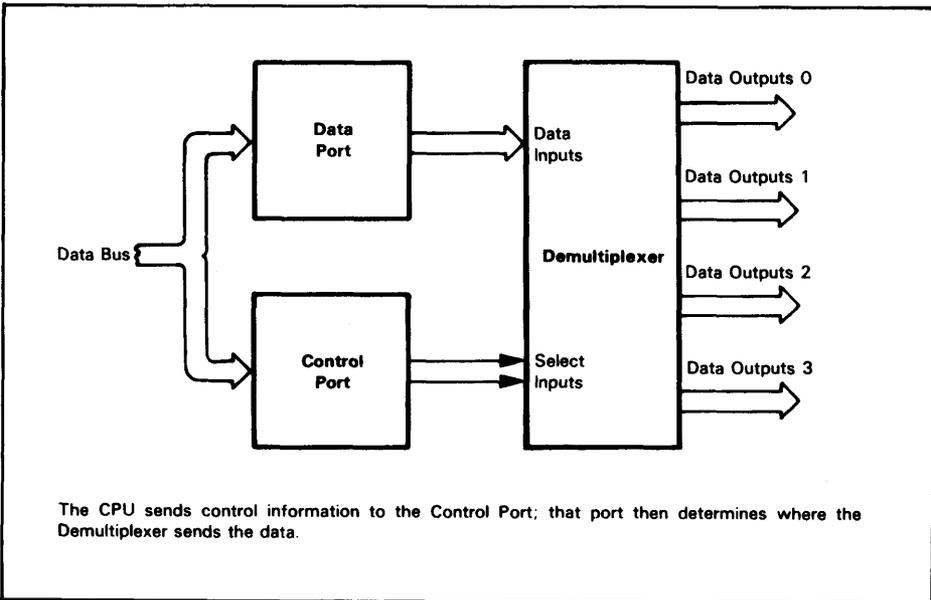


Figure 12-2. An Output Demultiplexer Controlled by a Port

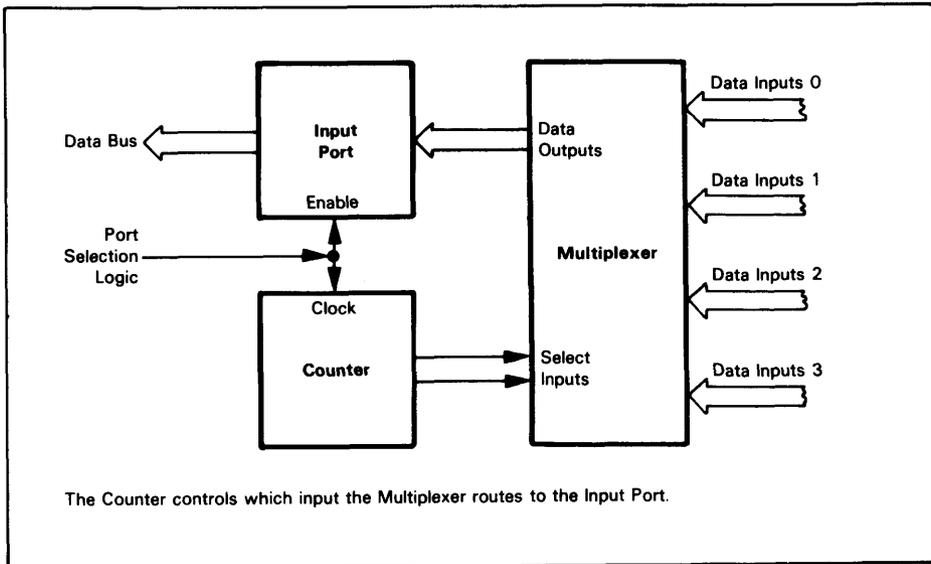


Figure 12-3. An Input Multiplexer Controlled by a Counter

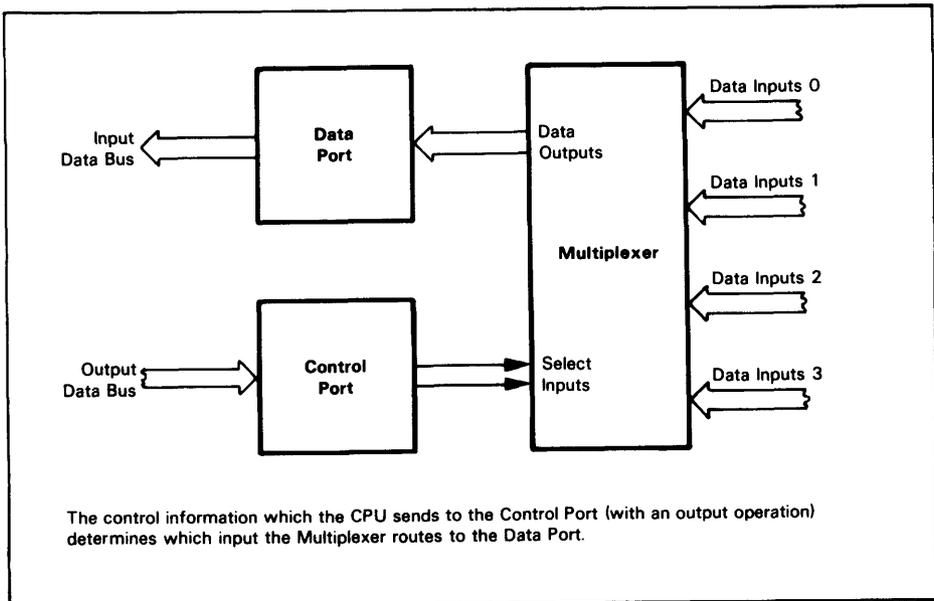


Figure 12-4. An Input Multiplexer Controlled by a Port

CPU clock cycles. Remember that the CPU is constantly using its data bus to perform ordinary memory transfers.

2. **Input transitions cause problems because of their duration; brief output transitions cause no problems** because the output devices (or the observers) react slowly.
3. **The major constraints on input are reaction time and responsiveness; the major constraints on output are response time and observability.**

INTERFACING MEDIUM-SPEED DEVICES

Medium-speed devices must be synchronized in some way to the processor clock. The CPU cannot simply treat these devices as if they held their data forever or could receive data at any time. Instead, the CPU must be able to determine when a device has new input data or is ready to receive output data. It must also have a way of telling a device that new output data is available or that the previous input data has been accepted. Note that **the peripheral may be or contain another processor.**

Handshake

The standard unlocked procedure is the handshake. Here the sender indicates the availability of data to the receiver and transfers the data; the receiver completes the handshake by acknowledging the receipt of the data. The receiver may control the situation by initially requesting the data or by indicating its readiness to accept data; the sender then sends the data and completes the handshake by indicating that data is available. In either case, the sender knows that the transfer has been completed successfully and the receiver knows when new data is available. The handshake procedure can operate at any speed, since the sender and receiver (not the clock) control the sequence of events.

Figures 12-5 and 12-6 show typical input and output operations using the handshake method. The procedure whereby the CPU checks the readiness of the peripheral before transferring data is called "polling." Clearly, polling can occupy a large amount of processor time if there are many I/O devices. **There are several ways of providing the handshake signals. Among these are:**

- **Separate dedicated I/O lines.** The processor may handle these as additional I/O ports or through special lines or interrupts. The 6809 microprocessor does not have special serial I/O lines, but the 6820 and 6821 Peripheral Interface Adapters (or programmable parallel interface chips) do.
- **Special patterns on the I/O lines.** These may be single start and stop bits or entire characters or groups of characters. The patterns must be easy to distinguish from background noise or inactive states.

Strobe

We often call a separate I/O line that indicates the availability of data or the occurrence of a transfer a "strobe." A strobe may, for example, clock data into a latch or fetch data from a buffer.

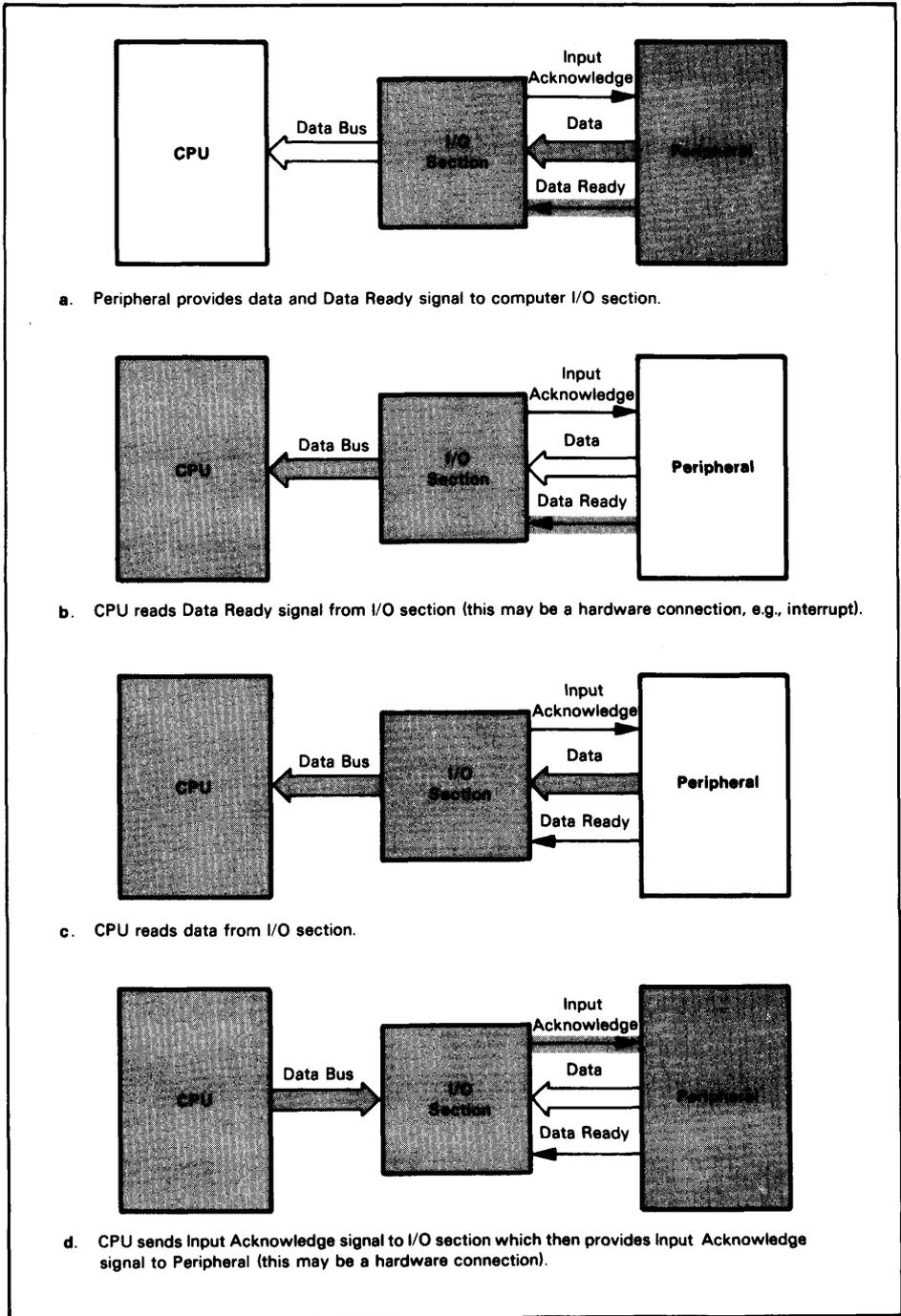


Figure 12-5. An Input Handshake

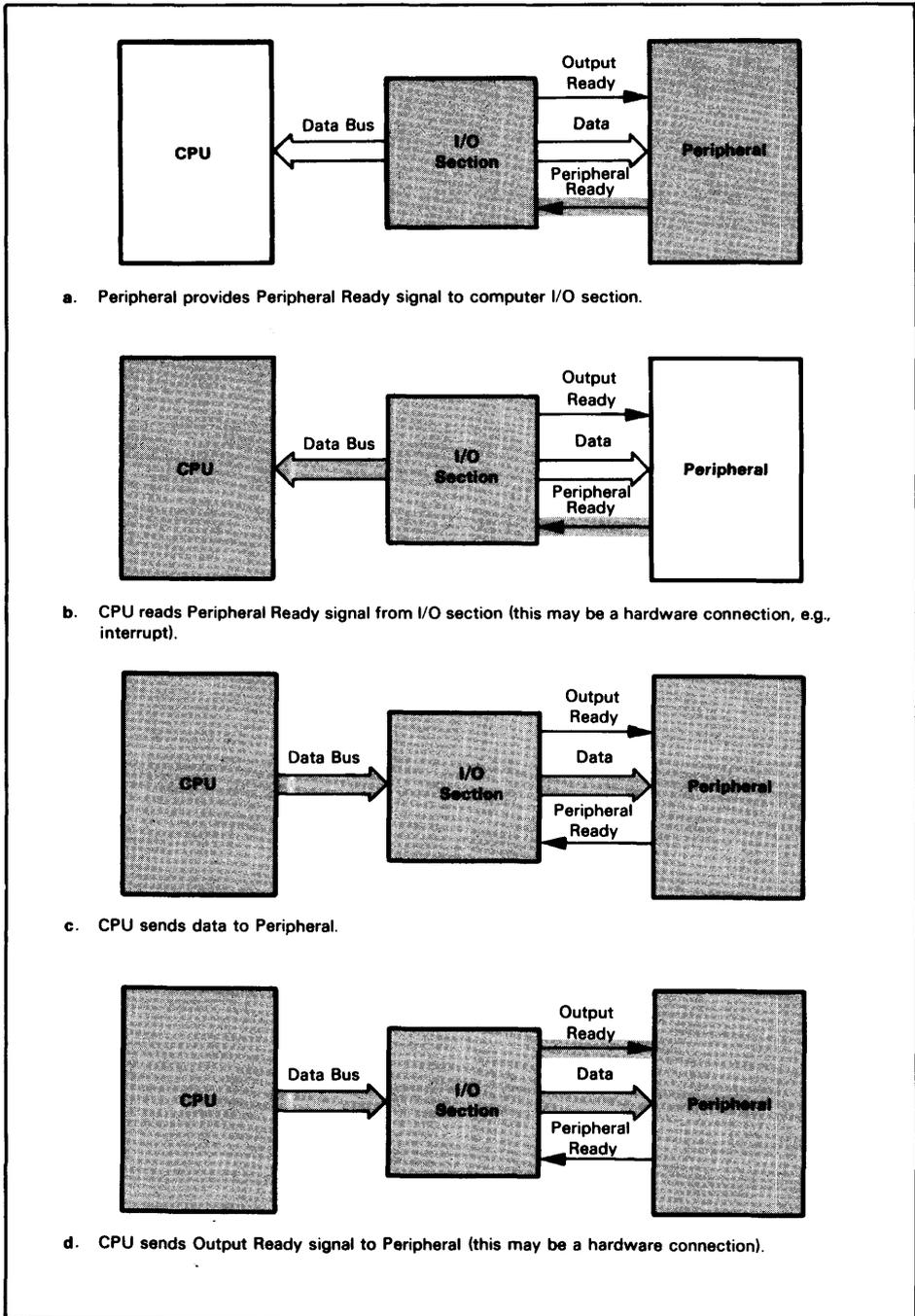


Figure 12-6. An Output Handshake

Many peripherals transfer data at regular intervals: i.e., synchronously. Here the only problem is starting the process by lining up to the first input or marking the first output. In some cases, the peripheral provides a clock input from which the processor can obtain timing information. **In synchronous I/O, the clock controls the speed of the transfers, rather than the sender and receiver.**

Reducing Transmission Errors

Transmission errors are a problem with medium-speed devices. Several methods can lessen the likelihood of such errors; they include:

- **Sampling input data at the center of the transmission interval** in order to avoid edge effects; that is, keep away from the edges where the data is changing.
- **Sampling each input several times and using majority logic.** For example, one could read each bit 5 times and choose the value that occurred most often.¹
- **Generating and checking parity;** an extra bit is used that makes the number of 1 bits in the correct data even or odd.
- **Using other error detecting and correcting codes** such as checksums, LRC (longitudinal redundancy check), and CRC (cyclic redundancy check).²

INTERFACING HIGH-SPEED DEVICES

High-speed devices that transfer more than 10,000 bits per second require special methods. The usual technique is to construct a special-purpose controller that transfers data directly between the memory and the I/O device. This process is called direct memory access (DMA). The DMA controller must force the CPU off the busses, provide addresses and control signals to the memory, and transfer the data. Such a controller will be fairly complex, typically consisting of 50 to 100 chips, although LSI devices such as the 6844 DMA controller³ for 6809-based microcomputers are now available. The CPU must initially load the Address and Data Counters in the controller so the controller will know where to start and how much data to transfer.

TIME INTERVALS

A common problem in I/O programming is how to provide time intervals of various lengths between operations. Such intervals are necessary to debounce mechanical switches (i.e., to smooth their irregular transitions), to provide pulses with specified lengths and frequencies for displays, and to time I/O operations for devices that transfer data regularly (e.g., a teletypewriter that sends or receives one bit every 9.1 ms).

METHODS FOR PRODUCING TIME INTERVALS

We can produce time intervals in several ways:

1. **In hardware** with one-shots or monostable multivibrators. These devices produce a single pulse of fixed duration in response to a pulse input. However, one-shots create reliability problems and they should be avoided whenever possible.
2. **In a combination of hardware and software** with a flexible device such as the 6840 Programmable Timer for 6809-based microcomputers.⁴ The 6840 device can provide time intervals of various lengths with a variety of starting and ending conditions.
3. **In software with delay routines.** A delay routine has no purpose other than to waste time; it is the computer equivalent of counting on your fingers. We can easily specify how much time the computer is to waste, since we know the clock speed of our particular microcomputer (this is system-dependent) and the number of clock cycles required to execute instructions (Appendices B and C). The problem with pure delay routines is that the processor cannot do other tasks while it is wasting time; however, delay routines require no hardware and may use processor time that would be wasted anyway.

The choice among these three methods depends on your application. The software method is inexpensive but may overburden the processor. The programmable timers are relatively expensive but are easy to interface and may be able to handle many complex timing tasks.

The timer in the 6846 Multifunction Support Device (ROM/IO/Timer)⁵ is available at no extra cost if this part is being used. The part is somewhat more expensive than simpler devices, but may be justifiable as a complete, one-chip package. 6846 devices are used in many board-level microcomputers.

DELAY ROUTINES

A simple delay routine works as follows:

- STEP 1 — Load a register with a specified value.
- STEP 2 — Decrement the register.
- STEP 3 — If the result is not zero, repeat STEP 2.

This routine does nothing except use time. The amount of time used depends on the execution time of the various instructions. The maximum length of the delay is

limited by the size of the register; however, the entire routine can be placed inside a similar routine that uses another register, etc.

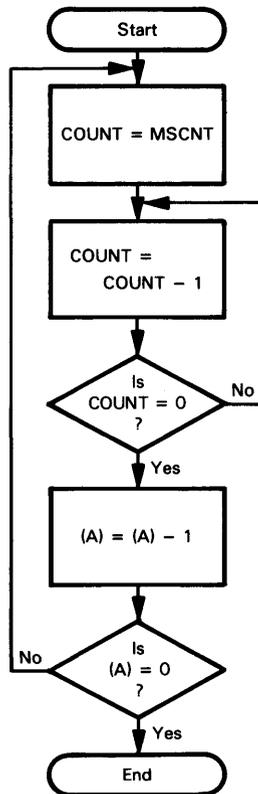
Be careful — the actual time used depends on the clock rate at which the processor is running, the speed of memory accesses, and operating conditions such as temperature, power supply voltage, and circuit loading which may affect the speed at which the processor executes instructions.

The following example subroutine (starting in memory address 0030) uses the two Accumulators to produce delays as long as 255 ms. The routine saves Accumulator B and the Condition Code Register in the Hardware Stack so they are not changed. We could use either of the general parameter passing techniques from Chapter 11 to write a completely “transparent” subroutine that would not affect any registers or flags. Of course, we would have to include the extra instructions that transfer parameters, save and restore registers, and adjust the return address in the time budget.

Program Example: A Delay Subroutine

Purpose: The subroutine produces a delay of 1 ms times the contents of Accumulator A.

Flowchart:



The value of MSCNT depends on the rate at which the CPU executes instructions.

Program a:

```

00C3      MSCNT EQU    $C3
          *
          ORG    $0030
0030 34   05      DELAY PSHS B,CC   SAVE INCIDENTAL REGISTERS
0032 C6   C3      DLY1  LDB  #MSCNT GET COUNT FOR 1 MS DELAY
0034 5A           DLY   DECB
0035 26   FD      BNE   DLY   COUNT WITH B FOR 1 MS
0037 4A           DECA      COUNT NUMBER OF MILLISECONDS
0038 26   F8      BNE   DLY1
003A 35   85      PULS  PC,B,CC RESTORE INCIDENTAL REGISTERS
          *                               AND RETURN

```

Time Budget:

Instruction	Number of Times Executed
PSHS B,CC	1
LDB #MSCNT	(A)
DECB	(A) × MSCNT
BNE DLY	(A) × MSCNT
DECA	(A)
BNE DLY1	(A)
PULS PC,B,CC	1

The total time used should be $(A) \times 1$ ms. If the memory is operating at full speed, the instructions require the following numbers of clock cycles (according to Appendix C).

Instruction	Number of Clock Cycles
PSHS B,CC	7
LDB #MSCNT	2
DECA or DECB	2
BNE	3
PULS PC,B,CC	9

Remember that PSHS and PULS require 5 clock cycles plus 1 clock cycle for each *byte* pushed or pulled.

Ignoring the Jump or Branch-to-Subroutine instruction (its execution time depends on the addressing mode used), the program takes

$$(A) \times (7 + 5 \times \text{MSCNT}) + 16 \text{ clock cycles}$$

The 7 is the number of cycles required by LDB #MSCNT, DECA, and BNE DLY1; the 5 is the number of cycles required by DECB and BNE DLY; the 16 is the number of cycles required by PSHS B,CC and PULS PC,B,CC.

So, to make the delay 1 ms,

$$23 + 5 \times \text{MSCNT} = N_c$$

where N_c is the number of clock cycles per millisecond. At a 1 MHz 6809 clock rate, $N_c = 1000$ so

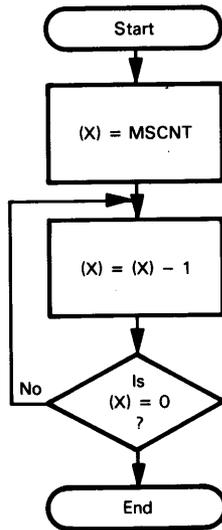
$$5 \times \text{MSCNT} = 977$$

$$\text{MSCNT} = 195 \text{ (C3}_{16}\text{) at a 6809 clock rate of 1 MHz}$$

The next version is a subroutine using Index Register X to produce a delay of 1 millisecond without affecting any registers.

12-12 6809 Assembly Language Programming

Flowchart:



The value of MSCNT depends on the execution time of the instructions in the program.

Program b:

```

007A MSCNT EQU $007A
*
ORG $0030
0030 34 10 DELAY PSHS X
0032 8E 007A LDX #MSCNT GET COUNT FOR 1 MS DELAY
0035 30 1F DLY LEAX -1,X COUNT X DOWN FOR 1 MS
0037 26 FC BNE DLY
0039 35 90 PULS PC,X
  
```

Remember that MSCNT is a 16-bit number.

Time Budget:

Instruction	Number of Times Executed	Number of Clock Cycles
PSHS X	1	7
LDX #MSCNT	1	3
LEAX -1,X	MSCNT	5
BNE DLY	MSCNT	3
PULS PC,X	1	9

Ignoring the JSR or BSR instruction, the program takes

$$19 + 8 \times \text{MSCNT clock cycles}$$

For this program to take 1 ms to execute at a 1 MHz clock rate, we need

$$19 + 8 \times \text{MSCNT} = 1000$$

$$\text{MSCNT} = 122 (007A_{16})$$

At a 2 MHz clock rate, we need

$$19 + 8 \times \text{MSCNT} = 2000$$

$$\text{MSCNT} = 247 (00F7_{16})$$

LOGICAL AND PHYSICAL DEVICES⁶

An important goal in writing I/O routines is to make them independent of particular physical hardware. The routines can then transfer data to or from I/O devices, with the actual addresses being supplied as parameters. The I/O device that can actually be accessed through a particular interface is referred to as a *physical device*. The I/O device to which the program transfers data is referred to as a *logical device*. The operating system or supervisor program must provide a mapping of logical devices on to physical devices, that is, assign actual physical I/O addresses and characteristics to be used by the I/O routines.

Note the advantages of this approach:

1. **The operating system can vary the assignments under user control.** Now the user can easily substitute a test panel or a development system interface for the actual I/O devices. This is useful in field maintenance as well as in debugging and testing. Furthermore, the user can change the I/O devices for different situations; typical examples are directing intermediate output to a video display and final output to a printer or obtaining some input from a remote communications line rather than from a local keyboard.
2. **The same I/O routines can handle several identical or similar devices.** The operating system or user only has to supply the address of a particular teletypewriter, RS-232 terminal, or printer, for example.
3. **Changes, corrections, or additions to the I/O configuration are easy to make** since only the assignments (or mapping) must be changed. On the 6809 microprocessor, the I/O routines can use the indexed addressing modes to provide independence of specific physical addresses. Indirect addressing allows one to access a physical device through a table. You can also use the LEA instruction to load the actual device address into an Index Register or Stack Pointer.

I/O DEVICE TABLE

If the system has a table of I/O addresses in memory (for example, starting at address IODEV) all an I/O routine needs is an index into the table. It can then access the I/O device using the indirect accumulator indexed mode. If, for example, the device address is table entry DEV, the following program calculates the index and loads the base address of the table into Index Register X:

```

LDA    DEV      GET DEVICE NUMBER
ASLA   #IODEV   MULTIPLY DEVICE NUMBER BY 2 FOR 2-BYTE ADDRESS TABLE
LDX    #IODEV   GET BASE ADDRESS OF I/O TABLE

```

The program can now transfer data to or from the I/O device using the instructions

```

LDB    DATA    GET DATA
STB    [A,X]    SEND DATA TO LOGICAL I/O DEVICE

```

or

```

LDB    [A,X]    GET DATA FROM LOGICAL I/O DEVICE
STB    DATA    SAVE DATA IN MEMORY

```

If the program uses an I/O device address repeatedly, it can load it into Index Register X with the instruction LDX [A,X] or LEAX [A,X]. Later instructions can then use the non-indirect indexed addressing mode with no offset.

Using this approach, a single I/O routine can transfer data to or from many different I/O devices. The main program simply supplies the I/O routine with the index for the device table. Compare the flexibility of this approach with the inflexibility of I/O routines that use direct or extended addressing to transfer data to or from I/O devices and are therefore tied to specific physical addresses.

STANDARD INTERFACES

You can use other standard interfaces besides the TTY current-loop and RS-232 to connect peripherals to a microcomputer. Popular ones include:^{7,8}

1. The serial RS-449, RS-422, and RS-423 interfaces.⁹
2. The 8-bit parallel General Purpose Interface Bus, also known as IEEE 488 or Hewlett-Packard Interface Bus (HPIB).¹⁰
3. The S-100 or IEEE 696 bus.¹¹ This 8-bit bus can also be used as a 16-bit bus.
4. The Intel Multibus.¹² This is another 8-bit bus that can be expanded to handle 16 bits in parallel.
5. Limited busses such as the Mostek/Pro-Log STD bus¹³ and the Intel iSBX bus.¹⁴ These are 8-bit busses that are intended to handle small additions to standard boards.

The S-100 and Multibus differ from the others listed in that they are “mother-board” busses which connect circuit boards within a single chassis. Such a bus connects peripheral interface control logic to the central processor and memory; a different interface (either a custom job or one of the standards we have mentioned) connects the peripheral device itself to the interface card in the microcomputer chassis.

6809 INPUT/OUTPUT CHIPS

Most 6809 input/output routines are based on LSI interface chips. These devices combine latches, buffers, flip-flops, and other logic circuits needed for handshaking and other simple interfacing techniques. They contain many logic connections, certain sets of which can be selected according to the contents of programmable registers. Thus the designer has the equivalent of a Circuit Designer's Casebook under his or her control. The initialization phase of the program places the appropriate values in registers to select the required logic connections. Input or output routines based on programmable LSI interface chips can handle many different applications, and changes or corrections can be made in software rather than by rewiring.

Designers often use the following LSI interface chips with the 6809 microprocessor:

1. **The 6820 or 6821 Peripheral Interface Adapter.** We will discuss this device in the next chapter. It contains two 8-bit I/O ports and four serial control lines. There are minor hardware differences between the 6820 and 6821 devices, but we will treat them as identical since they are the same from the programmer's point of view.
2. **The 6850 Asynchronous Communications Interface Adapter.** This device transforms data between the 8-bit parallel form and the serial form required in most communications applications. We will discuss the 6850 ACIA in Chapter 14.
3. **The 6551 Asynchronous Communications Interface Adapter.** This device is similar to the 6850 ACIA but includes an on-chip baud rate generator.
4. **The 6522 Versatile Interface Adapter,^{15,16} which includes two 8-bit I/O ports, four serial control lines, two 16-bit counter/timers, and an 8-bit shift register.**

REFERENCES

1. J. Barnes and V. Gregory. "Use Microprocessors to Enhance Performance with Noisy Data," *EDN*, August 20, 1976, pp. 71-72.
2. S. V. Alekar. "M6800 Program Performs Cyclic Redundancy Checks," *Electronics*, December 6, 1979, p. 167.
J. E. McNamara. *Technical Aspects of Data Communications*, Digital Equipment Corporation, Maynard, Mass., 1977, Chapter 13.
R. Swanson. "Understanding Cyclic Redundancy Codes," *Computer Design*, November 1975, pp. 93-99.
J. Wong et al. "Software Error Checking Procedures for Data Communications Protocols," *Computer Design*, February 1979, pp. 122-125.
3. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, Osborne/McGraw-Hill, 1978, pp. 9-106 through 9-123.
4. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, pp. 9-78 through 9-106.
5. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, pp. 9-124 through 9-130.
6. C. W. Gear. *Computer Organization and Programming*, 3rd ed., McGraw-Hill, New York, 1980, Chapter 6.
7. J. Kane et al. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices*, Chapter J, Osborne/McGraw-Hill, Berkeley, Calif.
C. A. Ogdin. "Microcomputer Buses," *Mini-Micro Systems*, June 1978, pp. 97-104 (Part 1); July 1978, pp. 76-80 (Part 2).

8. E. Teja and R. Peterson. "Selecting the Proper Bus," *EDN*, December 15, 1979, pp. 231-236.
9. D. Morris. "Revised Data Interface Standards," *Electronic Design*, September 1, 1977, pp. 138-141.
10. Institute of Electrical and Electronic Engineers. "IEEE Standard Digital Interface for Programmable Instrumentation," IEEE Std488-1978, IEEE, 445 Hoes Lane, Piscataway, N.J. 08854.
J. B. Peatman. *Microcomputer-Based Design*, McGraw-Hill, New York, 1977, pp. 299-311.
S. C. Baunach. "An Example of an M6800-Based GPIB Interface," *EDN*, September 20, 1977, pp. 125-128. A more detailed version of this article, complete with program listing and schematics, is available from Tektronix, Inc., Box 500, Beaverton, OR 97077.
S. M. Babb et al. "A General-Purpose IEEE-488 Bus Interface," Proceedings of the 1979 Conference on Industrial Applications of Microprocessors, Philadelphia, Pa., March 1979, pp. 121-125. This article also includes basic talker code for an M6800.
11. G. Morrow and H. Fullmer. "Proposed Standard for the S-100 Bus," *Computer*, May 1978, pp. 84-89.
K. A. Elmquist et al. "Standard Specification for S-100 Bus Interface Devices," *Computer*, July 1979, pp. 28-51.
12. T. Rolander. "Intel Multibus Interfacing," Intel Application Note AP-28, Intel Corporation, Santa Clara, CA, 1977.
An Introduction to Microcomputers: Volume 3 — Some Real Support Devices, Section J.
13. M. Biewer. "This Bus Handles Different Microprocessors," *Electronic Design*, October 11, 1978, pp. 220-224.
14. G. Sawyer et al. "Special-Function Modules Ride on Computer Board," *Electronics*, April 10, 1980, pp. 135-140.
15. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, pp. 10-34 through 10-53.
16. L. A. Leventhal. *6502 Assembly Language Programming*, Osborne/McGraw-Hill, pp. 11-23 through 11-38.

13

Using the 6820 Peripheral Interface Adapter (PIA)

The 6820 PIA^{1,2} is a device which supports many modes of parallel I/O. In this chapter we will discuss the programming of this device in some detail, and give several examples of fundamental I/O routines. The discussion in this chapter applies to the 6821 PIA as well; it and the 6820 appear equivalent to the programmer.

REGISTERS AND CONTROL LINES

Figure 13-1 is the block diagram of a PIA. The device contains two nearly identical 8-bit ports — A, which is usually an input port, and B, which is usually an output port. Each port contains:

- **A Data or Peripheral register** that holds either input or output data. This register is latched when used for output but unlatched when used for input.
- **A Data Direction register.** The bits in this register determine whether the corresponding data register bits (and pins) are inputs (0) or outputs (1).
- **A Control register** that holds the status signals required for handshaking, and other bits that select logic connections within the PIA.
- **Two control lines** that are configured by the control registers. These lines can be used for the handshaking signals shown in Figures 12-5 and 12-6.

The meanings of the bits in the Data Direction and Control Registers are related to the underlying hardware and are entirely arbitrary as far as the assembly language programmer is concerned. You must either memorize them or look them up in the appropriate tables (Tables 13-2 through 13-6).

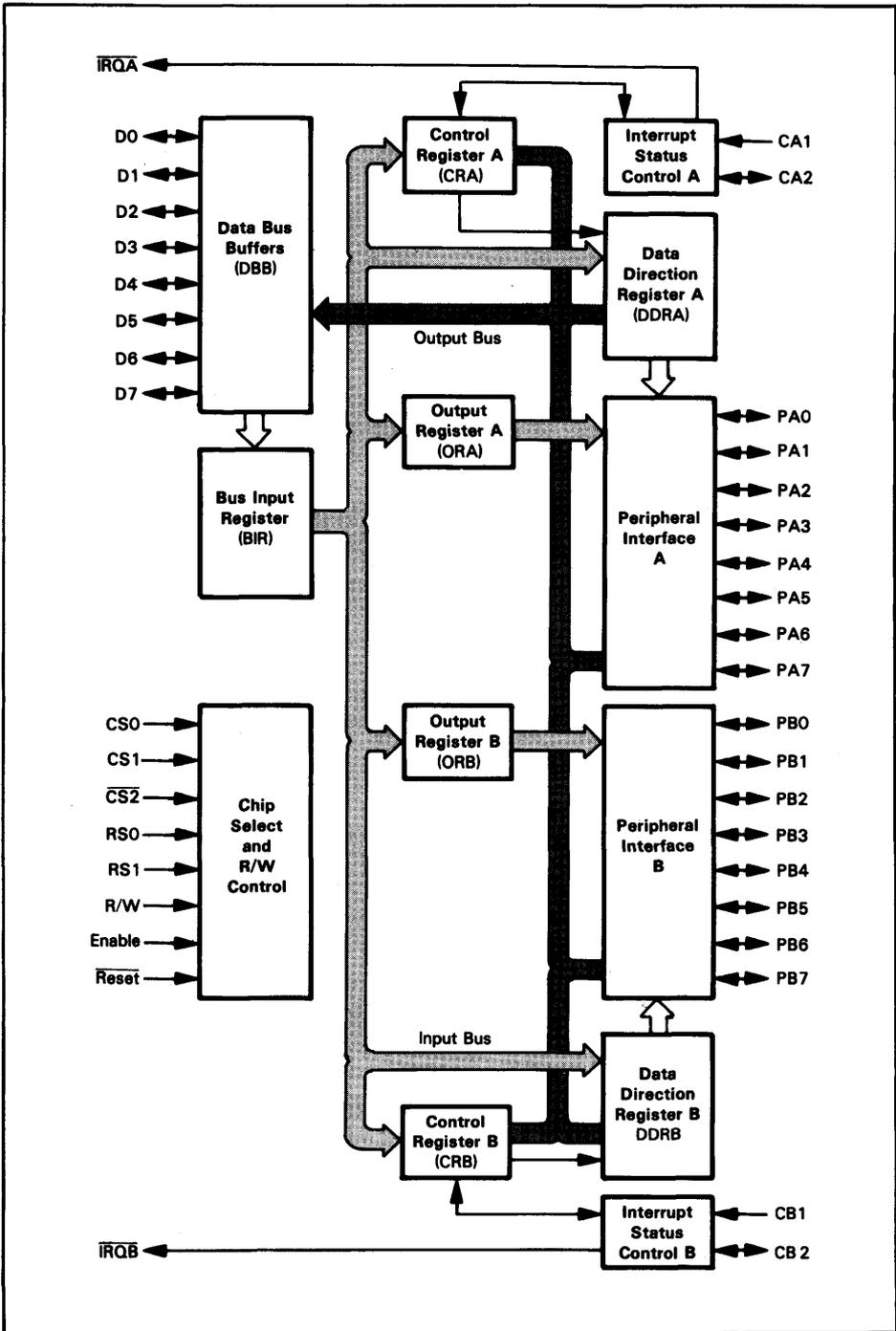


Figure 13-1. Block Diagram of the 6820 Peripheral Interface Adapter

Table 13-1. Addressing 6820 PIA Internal Registers

Address Lines		Control Register Bit		Register Selected	Offset Address (Index Register or Stack Pointer) = Address of Peripheral (Data) Register A
RS1	RS0	CRA-2	CRB-2		
0	0	1	X	Peripheral Register A	0
0	0	0	X	Data Direction Register A	0
0	1	X	X	Control Register A	1
1	0	X	1	Peripheral Register B	2
1	0	X	0	Data Direction Register B	2
1	1	X	X	Control Register B	3

X = Either 0 or 1

Addresses

Each PIA occupies four memory addresses. The RS (register select) lines choose one of the four registers, as described in Table 13-1. Since there are six registers (two peripheral, two data direction, and two control) in each PIA, one further bit is needed for addressing. Bit 2 of each Control Register determines whether the other address on that side refers to the Data Direction Register (0) or to the Peripheral Register (1). This sharing of an external address means that

1. A program must change the bit in the Control Register in order to use the register that is not currently being addressed.
2. The programmer must know the contents of the Control Register to determine which register is being addressed. RESET clears the Control Register and thus addresses the Data Direction register.

Table 13-1 also shows a convenient way to address the registers in a PIA. If, as is usually the case, the register select lines are tied to the least significant address lines (RS0 to A0 and RS1 to A1), the programmer can load an Index Register or Stack Pointer with the address of Data (Peripheral) Register A and refer to the other register by means of the constant offsets in the last column of Table 13-1.

PIA Control Registers

Table 13-2 shows the organization of the PIA Control Registers. We may describe the general purpose of each bit as follows:

- Bit 7: status bit set by transitions on control line 1 and cleared by reading the Peripheral (Data) register
- Bit 6: same as bit 7 except set by transitions on control line 2
- Bit 5: determines whether control line 2 is an input (0) or output (1)
- Bit 4: Control line 2 input: determines whether bit 6 is set by high-to-low transitions (0) or low-to-high transitions (1) on control line 2
Control line 2 output: determines whether control line 2 is a pulse (0) or a level (1)
- Bit 3: Control line 2 input: if 1, enables interrupt output from bit 6

13-4 6809 Assembly Language Programming

Control line 2 output: determines ending condition for pulse (0 = handshake acknowledgment lasting until next transition on control line 1, 1 = brief strobe lasting one clock cycle) or value of level

Bit 2: selects Data Direction Register (0) or Data Register (1)

Bit 1: determines whether bit 7 is set by high-to-low transitions (0) or low-to-high transitions (1) on control line 1

Bit 0: if 1, enables interrupt output from bit 7 of Control Register

Tables 13-3 through 13-6 describe the bits in more detail. Since E is normally tied to the $\Phi 2$ clock, you can interpret "E" pulse as "clock pulse."

Table 13-2. Organization of the PIA Control Registers

CRA	7	6	5	4	3	2	1	0
	IRQA1	IRQA2	CA2 Control			DDRA Access	CA1 Control	
CRB	7	6	5	4	3	2	1	0
	IRQB1	IRQB2	CB2 Control			DDRB Access	CB1 Control	

Table 13-3. Control of 6820 PIA Interrupt Inputs CA1 and CB1

CRA 1 (CRB-1)	CRA-0 (CRB-0)	Interrupt Input CA1 (CB1)	Interrupt Flag CRA-7 (CRB-7)	MPU Interrupt Request \overline{IRQA} (\overline{IRQB})
0	0	↓ Active	Set high on ↓ of CA1 (CB1)	Disabled — \overline{IRQ} remains high
0	1	↓ Active	Set high on ↓ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high
1	0	↑ Active	Set high on ↑ of CA1 (CB1)	Disabled — \overline{IRQ} remains high
1	1	↑ Active	Set high on ↑ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high

Notes:

- ↑ indicates positive transition (low to high)
- ↓ indicates negative transition (high to low)
- The interrupt flag bit CRA-7 is cleared by an MPU Read of the A Data Register, and CRB-7 is cleared by an MPU Read of the B Data Register
- If CRA-0 (CRB-0) is low when an interrupt occurs (Interrupt disabled) and is later brought high, \overline{IRQA} (\overline{IRQB}) occurs after CRA-0 (CRB-0) is written to a "one."

Table 13-4. Control of 6820 PIA Interrupt Inputs CA2 and CB2 (CRA5 (CRB5) is Low)

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	Interrupt Input CA2 (CB2)	Interrupt Flag CRA-6 (CRB-6)	MPU Interrupt Request \overline{IRQA} (\overline{IRQB})
0	0	0	↓ Active	Set high on ↓ of CA2 (CB2)	Disabled — \overline{IRQ} remains high
0	0	1	↓ Active	Set high on ↓ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high
0	1	0	↑ Active	Set high on ↑ of CA2 (CB2)	Disabled — \overline{IRQ} remains high
0	1	1	↑ Active	Set high or ↑ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high

Notes

- ↑ indicates positive transition (low to high)
- ↓ indicates negative transition (high to low)
- The Interrupt flag bit CRA-6 is cleared by an MPU Read of the A Data Register and CRB-6 is cleared by an MPU Read of the B Data Register
- If CRA-3 (CRB-3) is low when an interrupt occurs (Interrupt disabled) and is later brought high, \overline{IRQA} (\overline{IRQB}) occurs after CRA-3 (CRB-3) is written to a "one"

Table 13-5. Control of 6820 PIA CB2 Output Line (CRB5 is High)

CRB-5	CRB-4	CRB-3	CB2		Mode
			Cleared	Set	
1	0	0	Low on the positive transition of the first E pulse following an MPU Write "B" Data Register operation.	High when the interrupt flag bit CRB-7 is set by an active transition of the CB1 signal.	Automatic Output Acknowledge
1	0	0	Low on the positive transition of the first E pulse after an MPU Write "B" Data Register operation.	High on the positive edge of the first "E" pulse following an "E" pulse which occurred while the part was deselected.	Automatic Output (Write) Strobe
1	1	0	Low when CRB-3 goes low as a result of an MPU Write in Control Register "B".	Always low as long as CRB-3 is low. Will go high on an MPU Write in Control Register "B" that changes CRB-3 to "one".	Manual Output (Low)
1	1	1	Always high as long as CRB-3 is high. Will be cleared when an MPU Write Control Register "B" results in clearing CRB-3 to "zero."	High when CRB-3 goes high as a result of an MPU Write into Control Register "B".	Manual Output (High)

Table 13-6. Control of 6820 PIA CA2 Output Line (CRA5 is High)

CRA-5	CRA-4	CRA-3	CA2		Mode
			Cleared	Set	
1	0	0	Low on negative transition of E after an MPU Read "A" Data operation.	High when the interrupt flag bit CRA-7 is set by an active transition of the CA1 signal.	Automatic Input Acknowledge
1	0	1	Low on negative transition of E after an MPU Read "A" Data operation.	High on the negative edge of the first "E" pulse which occurs during a deselect.	Automatic Input (Read) Strobe
1	1	0	Low when CRA-3 goes low as a result of an MPU Write to Control Register "A".	Always low as long as CRA-3 is low. Will go high on an MPU Write to Control Register "A" that changes CRA-3 to "one".	Manual Output (Low)
1	1	1	Always high as long as CRA-3 is high. Will be cleared on an MPU Write to Control Register "A" that clears CRA-3 to a "zero".	High when CRA-3 goes high as a result of an MPU Write to Control Register "A".	Manual Output (High)

INITIALIZING A PIA

As part of the general system initialization, the program must determine how each PIA will operate. Remember that a PIA contains a large number of logic connections, much as the processor itself does. The data stored in the control and data direction registers activates certain connections within the PIA, much as the data loaded into the instruction register of the CPU activates certain connections. The differences are that the PIA contains far fewer connections than the CPU and the program rarely, if ever, changes the active connections in a PIA.

The steps in determining how the PIA will operate are:

1. **Address the Data Direction Registers** by clearing bit 2 of each Control Register. This allows the program to determine which I/O pins will be inputs and which outputs. Since RESET clears the entire control register, this step is unnecessary in the overall system startup routine.
2. **Determine which I/O pins will be inputs and which outputs** by loading the appropriate combinations of 0's (for inputs) and 1's (for outputs) into the Data Direction Registers.
3. **Determine how the status and control lines will operate** by loading the appropriate values into bit positions 0, 1, 3, 4, and 5 of the Control Registers. Address the Data Registers by setting bit 2 of each Control Register.

The program can address a Data Direction Register as follows:

```

CLR   PIACR          CLEAR PIA CONTROL REGISTER

OR

LDA   PIACR
ANDA  #%11111011    ADDRESS DATA DIRECTION REGISTER
STA   PIACR

```

The second version is more general, since it does not change any of the other bits in the Control Register.

After the program has addressed the Data Direction Register, it can select the appropriate combination of inputs and outputs by storing the corresponding pattern of 0's and 1's in that register. Some simple examples are:

1. CLR PIADDR MAKE ALL DATA LINES INPUTS
2. LDA #\$FF MAKE ALL DATA LINES OUTPUTS
STA PIADDR
3. LDA #\$F0 MAKE DATA LINES 4-7 OUTPUTS, 0-3 INPUTS
STA PIADDR

The third step is clearly the most difficult, since it involves selecting the active logic connections in the PIA and thus determining how the device will operate.

Some factors to remember are:

1. **You cannot change bits 6 and 7 of the Control Register by writing data into them.** Only transitions on the control lines set these bits and only the reading of the corresponding Data Registers clears them.
2. **You must set bit 2 of each Control Register to address the Data Register and allow the transfer of data to or from the outside world.** As long as bit 2 of a Control Register is zero, the CPU can only access the corresponding Data Direction Register; it cannot transfer data to or from the I/O pins through the Data Register.
3. **Bit 1 of the Control Register determines which edge of a pulse on control line 1 will set bit 7.** If bit 1 is 0, a high-to-low transition (rising edge) on control line 1 will set bit 7; if bit 1 is 1, a low-to-high transition (falling edge) on control line 1 will perform that function. If control line 2 is an input, bit 4 provides the same choice for it.
4. **Bit 0 of the Control Register is an interrupt enable for control line 1.** Remember that this bit must be *set* to enable interrupts, unlike the 6809 Interrupt Mask bit, which must be *cleared* to enable interrupts. Chapter 15 describes interrupts in more detail. If control line 2 is an input, bit 3 performs the same function for it.
5. **Bit 5 determines whether control line 2 is an output (1) or an input (0).** Bits 3 and 4 determine how control line 2 will operate. In the pulse or automatic strobe mode, ports A and B differ; port A produces a pulse on CA2 only after the processor reads Data Register A, while port B produces a pulse on CB2 only after the processor writes into Data Register B.
6. **You must determine the operating mode of each port of each PIA in your system.** Each port has a separate Control Register, Data Direction Register, and Data Register.

PIA OPERATING MODES

We can refer to the operating modes in which CA2 or CB2 are output control signals as follows:

The modes in which the PIA automatically produces a pulse on CA2 after an

input operation or on CB2 after an output operation are called *automatic modes*, since the PIA produces the entire pulse without any explicit CPU intervention. The programmer has no control over the length or polarity of the pulse.

The mode in which bit 3 of the PIA Control Register determines the level of control line 2 is called a *manual mode*, since the CPU must produce changes by explicitly setting or clearing control register bit 3. The PIA does nothing automatically. This mode requires extra instructions, but gives the programmer complete control over the length and polarity of pulses.

Control line 2 has the following functions in the two automatic modes:

In the mode in which the automatic pulse lasts until the next active transition on control line 1, control line 2 is an *acknowledgment*. The active part of the pulse (the low period) signifies that the CPU has completed its part of the most recent I/O operation; the I/O device may start the next operation by sending data (input) or indicating its readiness (output).

In the mode in which the automatic pulse lasts one clock cycle, control line 2 is a *strobe*. The pulse indicates that the CPU has performed an I/O operation.

Examples of Selecting a PIA Operating Mode

1. A simple input port with no control lines (for example, as needed for a set of switches):

```
CLR  PIACR      ADDRESS DATA DIRECTION REGISTER
CLR  PIADDR     MAKE ALL DATA LINES INPUTS
LDA  #00000100 ADDRESS DATA REGISTER
STA  PIACR
```

The program first clears bit 2 of the Control Register to gain access to the Data Direction Register. It then makes all the data lines inputs by storing 0's in all the bits of the Data Direction Register and sets bit 2 of the Control Register to gain access to the Data Register (and the input port itself). The same sequence of instructions will handle the case in which a high-to-low transition (falling edge) on control line 1 indicates DATA READY or PERIPHERAL READY.

2. A simple output port with no control lines (for example, as needed for a set of single LED displays):

```
CLR  PIACR      ADDRESS DATA DIRECTION REGISTER
LDA  #FFF       MAKE ALL DATA LINE OUTPUTS
STA  PIADDR
LDA  #00000100 ADDRESS DATA REGISTER
STA  PIACR
```

The only difference from the previous example is that the program makes all the data lines outputs by storing 1's in all the bits of the Data Direction Register.

3. An input port with a status input that indicates DATA READY with a low-to-high transition (positive transition on control line 1).

```
CLR  PIACR      ADDRESS DATA DIRECTION REGISTER
CLR  PIADDR     MAKE ALL DATA LINES INPUTS
LDA  #00000110 MAKE DATA READY ACTIVE LOW-TO-HIGH
STA  PIACR
```

The only difference from Example 1 is that the program sets bit 1 of the Control Register. The result is that low-to-high transitions on control line 1 will

set bit 7 of the Control Register. This operating mode is suitable for most encoded keyboards.

4. **An output port that produces a brief strobe to indicate DATA READY or OUTPUT READY.** This strobe could be used to multiplex displays or to provide a DATA AVAILABLE signal to a printer.

```
CLR  PIACR          ADDRESS DATA DIRECTION REGISTER
LDA  #$FF          MAKE ALL DATA LINES OUTPUTS
STA  PIADDR
LDA  #%0101100    MAKE CONTROL LINE 2 A BRIEF STROBE
STA  PIACR
```

This program selects an operating mode for control line 2 as follows:

- Bit 5 = 1 to make control line 2 an output.
- Bit 4 = 0 to make control line 2 a pulse, rather than a level.
- Bit 3 = 1 to make the pulse one clock period long.

After each instruction that writes data into PIA Data Register B, control line 2 will go low for one clock cycle. For example, the instruction

```
STA  PIADB
```

will both send data to the Data Register (and hence to the output port) and cause a strobe on control line 2. However, the A port of a PIA will produce a strobe only after a read operation. The sequence

```
STA  PIADA          WRITE DATA
LDA  PIADA          PRODUCE AN OUTPUT STROBE
```

will both send the data to the output port and cause a strobe. The LDA instruction is a "dummy read;" it has no effect except causing the strobe (and wasting a few clock cycles). Other instructions besides LDA could serve the same purpose; you should try to name some of them.

5. **An input port with a handshake INPUT ACKNOWLEDGE strobe.** The strobe goes low when the CPU has read the data in the port and can accept more.

```
CLR  PIACR          ADDRESS DATA DIRECTION REGISTER
CLR  PIADDR        MAKE ALL DATA LINES INPUTS
LDA  #%0100100    CONTROL LINE 2 - HANDSHAKE ACKNOWLEDGE
STA  PIACR
```

Control register bit 5 = 1 to make control line 2 an output, bit 4 = 0 to make it a pulse, and bit 3 = 0 to make it an active-low acknowledgment that remains low until the next active transition on control line 1. The port operates as follows:

- a. A high-to-low transition on control line 1 indicates that the input peripheral has sent the computer new data. Bit 7 of the PIA Control Register is set and control line 2 goes high.
- b. The CPU determines that new data is available by examining bit 7 of the PIA Control Register. It therefore loads the data from the Data Register, thus clearing bit 7 of the Control Register and sending control line 2 low.
- c. The input peripheral can determine that the CPU has accepted the most recent data by examining control line 2. It can then repeat step a with complete assurance that no data will be lost.

The acknowledgment automatically follows any instruction that reads PIA Data Register A; for example, the instruction

```
LDA PIADA
```

will both read the data and cause the acknowledgment. However, the B port will produce an acknowledgment only after an instruction that writes into the Data Register. The sequence

```
LDA PIADB      READ DATA
STA PIADB      PRODUCE ACKNOWLEDGEMENT
```

will both read data and produce an acknowledgment. The STA instruction is a "dummy write;" it has no effect other than to cause an acknowledgment (that is, to send control line 2 low) and use a few clock cycles. Note that the instructions here are in the opposite order from those in Example 4. This operating mode is suitable for many CRT terminals that require a complete handshake.

6. **An output port with a latched zero control bit** (latched serial output or level output with value 0). The serial output can be used to turn a peripheral on or off or to determine its mode of operation.

```
CLR PIACR      ADDRESS DATA DIRECTION REGISTER
LDA #$FF      MAKE ALL DATA LINES OUTPUTS
STA PIADDR
LDA #%00110100 CONTROL LINE 2 - LATCHED OUTPUT, VALUE 0
STA PIACR
```

Bit 5 = 1 to make control line 2 an output, bit 4 = 1 to make it a level or latched bit, and bit 3 = 0 to make the value of the level zero. Operations on the Data Register do not affect control line 2 in this operating mode, so it will not automatically change value. The only way to change its value is for the program to change the value of bit 3 of the PIA control register; i.e.,

```
LDA PIACR
ORA #%00001000 MAKE SERIAL OUTPUT ONE
STA PIACR
```

OR

```
LDA PIACR
ANDA #%11110111 MAKE SERIAL OUTPUT ZERO
STA PIACR
```

You can use this operating mode to produce active-high strobes or to provide pulses with lengths determined by the program, rather than by the hardware.

USING THE PIA TO TRANSFER DATA

Once the program has determined the operating mode of the PIA, you may use its data registers like any other memory locations. **The most straightforward instructions for transferring data from an input device or to an output device are as follows:**

Load Accumulator transfers 8 bits of data from the specified input pins to an Accumulator.

Store Accumulator transfers 8 bits of data from an Accumulator to the specified output pins.

You must be cautious in situations in which input and output ports do not behave like memory locations. For example, it often makes no sense to write data into input ports or read data from output ports. Be particularly careful if the input port is not latched or if the output port is not buffered.

Other instructions that transfer data to or from memory can also serve as I/O instructions. Typical examples are:

Clear places zeros on a set of output pins.

Test sets the flags according to the values of a set of input pins.

Compare sets the flags as if the values of a set of input pins had been subtracted from the contents of an Accumulator.

Here also you must be aware of the physical limitations of the I/O ports. Be particularly careful of instructions like Test, Shift, Complement, Increment, and Decrement, which involve both read and write cycles.

We cannot overemphasize the importance of careful documentation. **Often, complex I/O transfers can be concealed in instructions with no obvious functions. You must describe the purposes of such instructions carefully.** For example, one could easily be tempted to remove the dummy read and write operations mentioned earlier since they do not appear to accomplish anything.

PIA Status Bits

Bit 7 of the PIA Control Register often serves as a status bit, such as Data Ready or Peripheral Ready. You can check its value with either of the following sequences:

LDA	PIACR	IS READY FLAG 1?
BMI	DEVVDY	YES, DEVICE READY
TST	PIACR	IS READY FLAG 1?
BMI	DEVVDY	YES, DEVICE READY

Note that you should not use the shift instructions, since they will change the contents of the Control Register (why?). The following program will wait for the Ready flag to go high:

WAITR	LDA	PIACR	IS READY FLAG 1?
	BPL	WAITR	NO, WAIT

How would you change these programs to examine bit 6 instead of bit 7?

The only way to clear bit 7 (or bit 6) is to read the Data Register. A dummy read will be necessary if a read operation is not normally part of the response to the bit being set. If the port is used for output, the sequence

STA	PIADR	SEND DATA
LDA	PIADR	CLEAR READY FLAG

will do the job. Note that here the dummy read is necessary on either port of the PIA. The Test instruction can also clear the strobe without changing anything except the flags. Be particularly careful of situations in which the CPU is not ready for input data or has no output data to send.

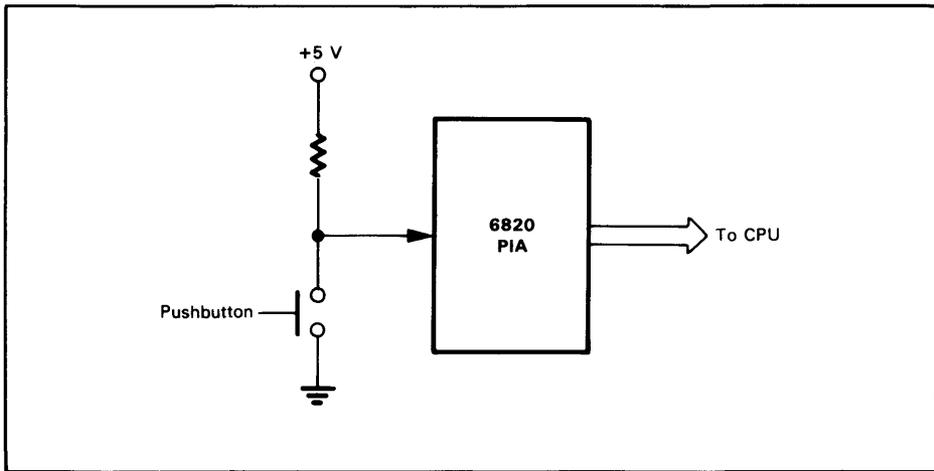


Figure 13-2. A Pushbutton Circuit

EXAMPLES

13-1. A PUSHBUTTON

We will interface a pushbutton to a 6809 microprocessor by means of a 6820 Peripheral Interface Adapter. The pushbutton is a mechanical switch; pressing the button closes the switch and connects the input bit to ground (see Fig. 13-2).

The 6820 PIA acts as a buffer; no latch is needed since the pushbutton remains closed for many CPU clock cycles. Pressing the button grounds one bit of the PIA. The pullup resistor ensures that the input bit is one if the button is not being pressed.

We will perform two tasks with this circuit. They are:

- a. Set a memory location based on the state of the button.
- b. Count the number of times the button is pressed.

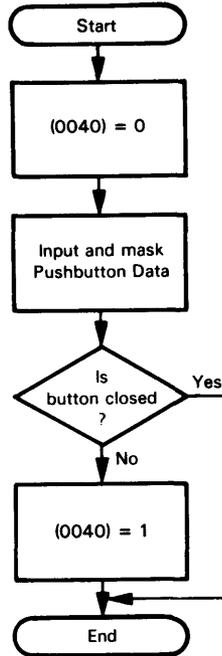
Task 13-1a. Determine Switch Closure

Purpose: Set memory location 0040 to one if the button is not being pressed, and to zero if it is.

Sample Cases:

1. Button open (not pressed)
Result = (0040) = 01
2. Button closed (pressed)
Result = (0040) = 00

Flowchart:



Program 13-1a:

```

      8001  PIACA  EQU   $8001
      8000  PIADDA EQU   $8000
      8000  PIADA  EQU   $8000
      0001  MASK  EQU   %00000001
      *
0000      ORG   $0000
0000 7F 8001      CLR   PIACA   ADDRESS DATA DIRECTION REGISTER
0003 7F 8000      CLR   PIADDA  MAKE ALL DATA LINES INPUTS
0006 86 04        LDA   #%00000100 ADDRESS DATA REGISTER
0008 B7 8001      STA   PIACA
000B 0F 40        CLR   $40     CLEAR BUTTON MARKER
000D B6 8000      LDA   PIADA  READ BUTTON POSITION
0010 84 01        ANDA  #MASK   IS BUTTON CLOSED (LOGICAL ZERO)?
0012 27 02        BEQ   DONE    YES, DONE
0014 0C 40        INC   $40     NO, SET BUTTON MARKER
0016 3F          DONE  SWI

```

The addresses PIACA (Control Register A), PIADDA (Data Direction Register A), and PIADA (Data Register A) depend on how the PIA is connected in your microcomputer. This example does not use the PIA control lines.

MASK depends on the bit to which the pushbutton is connected. It has a one in the button position and zeros elsewhere.

Button Position (Bit Number)	Mask	
	Binary	Hexadecimal
0	00000001	01
1	00000010	02
2	00000100	04
3	00001000	08
4	00010000	10
5	00100000	20
6	01000000	40
7	10000000	80

If the button is attached to bit 7 of the PIA input port, the program can use a LDA or TST instruction to set the Sign (Negative) flag and thereby determine the button's state. For example,

```

LDA  PIADA      IS BUTTON CLOSED (LOGIC ZERO)?
BPL  DONE      YES, DONE

TST  PIADA      IS BUTTON CLOSED (LOGIC ZERO)?
BPL  DONE      YES, DONE
    
```

We could also use shift instructions if the button is attached to bit 0, 6, or 7. The sequence for bit 0 is:

```

LSR  PIADA      IS BUTTON CLOSED (LOGIC ZERO)?
BCC  DONE      YES, DONE
    
```

The instructions ASL or ROL can be used with bit 6 or 7. Do the contents of the PIA data register actually change? Explain your answer.

Task 13-1b. Count Switch Closures

Purpose: Count the number of button closures by incrementing memory location 0040 after each closure.

Sample Case:

Pressing the button ten times after the start of the program should result in

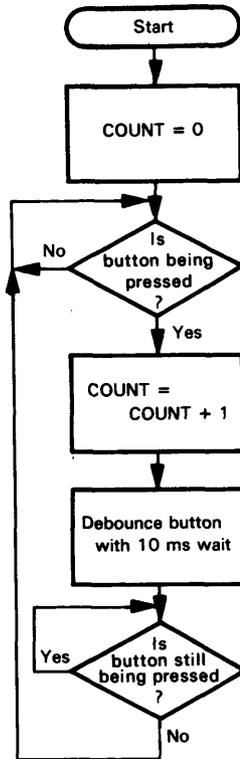
$$(0040) = 0A$$

In order to count the number of times the button has been pressed, we must be sure that each closure causes a single transition. However, a mechanical pushbutton does not produce a single transition for each closure, because the mechanical contacts bounce back and forth before settling into their final positions. We can use a one-shot to eliminate the bounce or we can handle it in software.

The program can debounce the pushbutton by waiting after it finds a closure. The required delay is called the debouncing time and is part of the specifications of the pushbutton. It is typically a few milliseconds long. The program should not examine the pushbutton during this period because it might mistake the bounces for new closures. The program may either enter a delay routine like the one described previously or may simply perform other tasks for the specified amount of time.

Even after debouncing, the program must still wait for the present closure to end before looking for a new closure. This procedure avoids double counting. The following program uses a software delay of 10 ms to debounce the pushbutton. You may want to try varying the delay or eliminating it entirely to see what happens. To run this program, you must also enter the delay subroutine into memory starting at location 0030.

Flowchart:



Program 13-1b:

```

0030 DELAY EQU $0030
8001 PIACA EQU $8001
8000 PIADDA EQU $8000
8000 PIADA EQU $8000
0001 MASK EQU %00000001
*
0000 ORG $0000
0000 7F 8001 CLR PIACA ADDRESS DATA DIRECTION REGISTER
0003 7F 8000 CLR PIADDA MAKE PORT A LINES INPUTS
0006 86 04 LDA #%00000100 ADDRESS DATA REGISTER
0008 B7 8001 STA PIACA
000B 0F 40 CLR $40 COUNT = ZERO INITIALLY
000D B6 8000 CHKCLO LDA PIADA
0010 84 01 ANDA #MASK IS BUTTON BEING PRESSED?
0012 26 F9 BNE CHKCLO NO, WAIT UNTIL IT IS
0014 0C 40 INC $40 YES, ADD 1 TO CLOSURE COUNT
0016 86 0A LDA #10 WAIT 10 MS TO DEBOUNCE BUTTON
0018 9D 30 JSR DELAY
001A B6 8000 CHKOPN LDA PIADA IS BUTTON STILL BEING PRESSED?
001D 84 01 ANDA #MASK
001F 27 F9 BEQ CHKOPN YES, WAIT FOR RELEASE
0021 20 EA BRA CHKCLO NO, LOOK FOR NEXT CLOSURE
    
```

13-16 6809 Assembly Language Programming

The three instructions beginning with the label CHKOPN determine when the switch reopens (i.e., when the button is released). If the PIA is addressed as shown in the last column of Table 13-1, we can load Index Register X with the address of Data Register A, and we can then use indexed offsets to address the PIA as follows:

Original		Replacement	
		(X) = PIADRA	
CLR	PIACRA	CLR	1,X
CLR	PIADDR	CLR	,X
STA	PIACRA	STA	1,X
LDA	PIADRA	LDA	,X

Clearly we do not need a PIA for this simple interface. An addressable tristate buffer would do the job at far lower cost.

13-2. A MULTIPLE-POSITION (ROTARY, SELECTOR, OR THUMBWHEEL) SWITCH

We will interface a multiple-position switch to a 6809 microprocessor. The lead corresponding to the switch position is grounded, while the other leads are high (logic ones).

Figure 13-3 shows the circuitry required to interface an 8-position switch. The switch uses all eight data bits of one port of a PIA. Typical tasks are to determine the position of the switch and to check if that position has changed. The program must handle two special conditions:

1. The switch is temporarily between positions so no leads are grounded.
2. The switch has not reached its final position.

The first condition can be handled by waiting until the input is not all ones, i.e., until a switch lead is grounded. We can handle the second condition by examining the switch again after a delay (such as 1 or 2 seconds) and only accepting the input when it

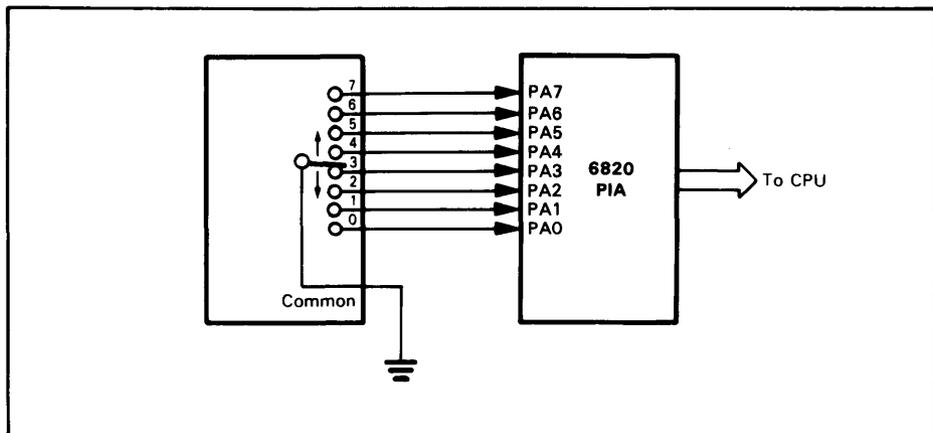


Figure 13-3. A Multiple-Position Switch

Table 13-7. Data Input vs Switch Position

Switch Position	Data Input	
	Binary	Hex
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

This scheme is inefficient, since it requires eight bits to distinguish among eight different positions

remains the same. This delay will not affect the responsiveness of the system to the switch. Alternatively, we could use another switch (i.e., a Load switch) to tell the processor when to read the selector switch.

We will perform two tasks involving the circuit of Figure 13-3. These are:

- a. Monitor the switch until it is in a definite position, then determine the position and store its binary value in a memory location.
- b. Wait for the position of the switch to change, then store the new position in a memory location.

If the switch is in a position, the lead from that position is grounded through the common line. Pullup resistors on the input lines avoid problems caused by noise.

Task 13-2a. Determine Switch Position

Purpose: The program waits for the switch to be in a specific position and then stores the position number in memory location 0040.

Table 13-7 contains the data inputs corresponding to the various switch positions. This scheme is inefficient, since it requires eight bits to distinguish among eight different positions.

A TTL or MOS encoder could reduce the number of input bits needed. Figure 13-4 shows a circuit using the 74LS148 TTL 8-to-3 encoder.³ We attach the switch outputs in inverse order, since the 74LS148 device has active-low inputs and outputs. The output of the encoder circuit is a 3-bit representation of the switch position. Many switches include encoders so their outputs are coded, usually as a BCD digit (in negative logic).

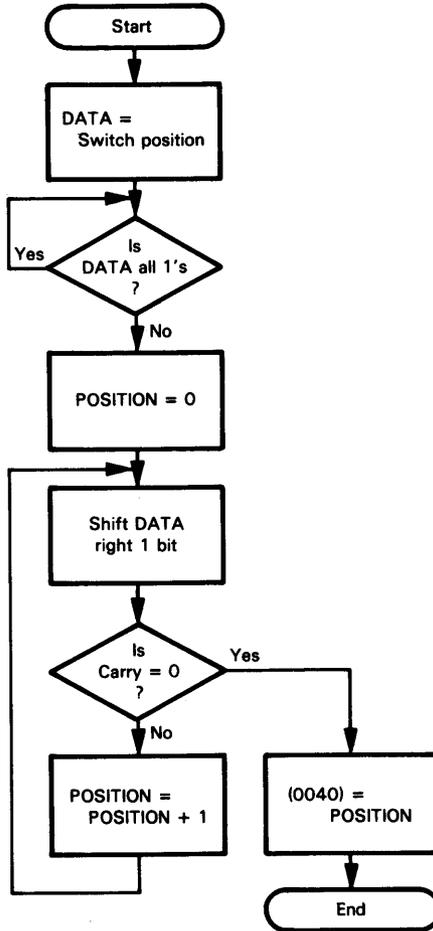
The encoder produces active-low outputs, so, for example, switch position 5, which is attached to input 2, produces an output of 2 in negative logic (or 5 in positive logic). You may want to verify the double negative for yourself.

Suppose that a faulty switch or defective PIA results in the input always being FF₁₆. How could you change the program so it would detect this situation?

We have arranged the loop that identifies the switch position for somewhat increased efficiency. The program initializes the position to -1 and then increments the position (with INCB) before shifting the input (with LSRA). What happens if you initialize the switch position to zero and shift and check the input before incrementing the position? The approach in which you start the program one step backward often increases execution speed because it lets you handle the first iteration in the same way as the subsequent ones.

13-18 6809 Assembly Language Programming

Flowchart:



Program 13-2a:

8001	PIACA	EQU	\$8001	
8000	PIADDA	EQU	\$8000	
8000	PIADA	EQU	\$8000	
	*			
0000		ORG	\$0000	
0000 7F	8001	CLR	PIACA	ADDRESS DATA DIRECTION REGISTER
0003 7F	8000	CLR	PIADDA	MAKE ALL DATA LINES INPUTS
0006 86	04	LDA	##00000100	ADDRESS DATA REGISTER
0008 B7	8001	STA	PIACA	
000B B6	8000	CHKSW	PIADA	
000E 81	FF	CMPA	#\$FF	IS SWITCH IN A POSITION?
0010 27	F9	BEQ	CHKSW	NO, WAIT UNTIL IT IS
0012 C6	FF	LDB	#\$FF	SWITCH POSITION = -1
0014 5C		CHKPOS	INCB	ADD 1 TO SWITCH POSITION
0015 44		LSRA		IS NEXT BIT GROUNDED POSITION?
0016 25	FC	BCS	CHKPOS	NO, KEEP LOOKING
0018 D7	40	STB	\$40	SAVE SWITCH POSITION
001A 3F		SWI		

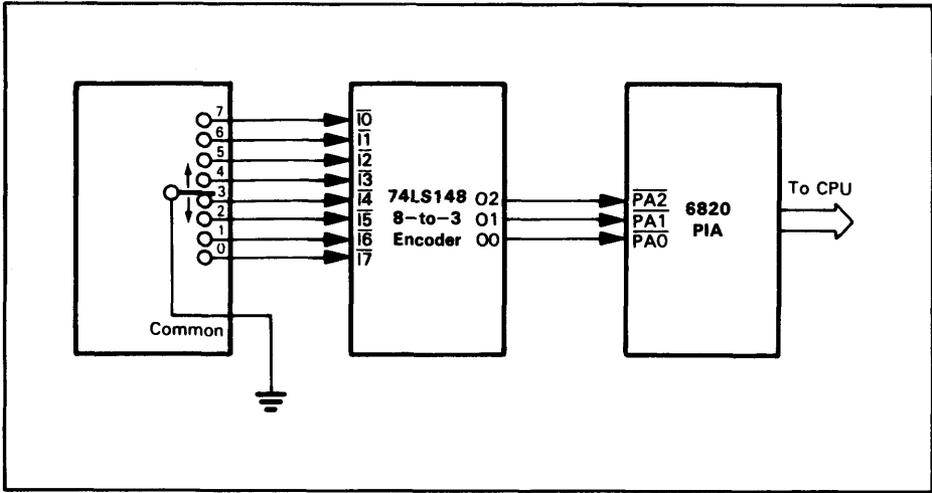


Figure 13-4. A Multiple-Position Switch with an Encoder

A short, quick way to determine if the switch is in a position is:

```

CHKSW  INC  PIADA  IS SWITCH IN A POSITION?
        BEQ  CHKSW  NO, WAIT UNTIL IT IS
    
```

Why does this approach work? Do the contents of the PIA Data Register actually change? Could you use the CARRY flag instead of the ZERO flag? Explain your answers.

This example assumes that the switch is debounced in hardware. How would you change the program to debounce the switch in software?

Task 13-2b. Wait for Switch Position to Change

Purpose: The program waits for the switch position to change and places the new position (decoded) into memory location 0040. The program waits until the switch reaches its new position.

Program 13-2b:

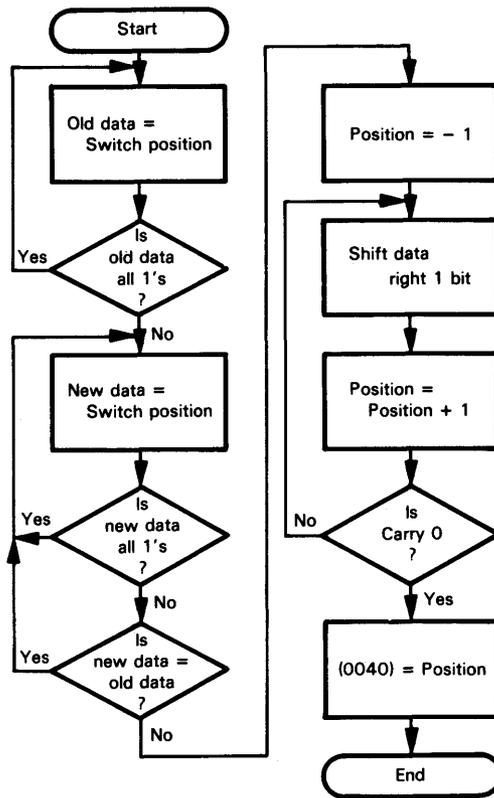
```

            8001  PIACA  EQU  $8001
            8000  PIADDA EQU  $8000
            8000  PIADA  EQU  $8000
            *
0000        ORG  $0000
0000 7F 8001    CLR  PIACA  ADDRESS DATA DIRECTION REGISTER
0003 7F 8000    CLR  PIADDA  MAKE ALL DATA LINES INPUTS
0006 86 04     LDA  #$0000100 ADDRESS DATA REGISTER
0008 B7 8001    STA  PIACA
000B B6 8000    CHKFS  LDA  PIADA  GET SWITCH DATA
000E 81 FF     CMPA  #$FF  IS THE SWITCH IN A POSITION?
0010 27 F9     BEQ  CHKFS  NO, WAIT UNTIL IT IS
0012 97 40     STA  $40  YES, SAVE SWITCH POSITION
0014 B6 8000    CHKSEC  LDA  PIADA  GET NEW SWITCH DATA
0017 91 40     CMPA  $40  IS POSITION SAME AS BEFORE?
0019 27 F9     BEQ  CHKSEC  YES, WAIT FOR POSITION TO
            *                                CHANGE
    
```

13-20 6809 Assembly Language Programming

001B C6 FF	LDB #SFF	NO, START POSITION AT -1
001D 5C	CHKPOS	ADD 1 TO SWITCH POSITION
001E 44	LSRA	IS NEXT BIT GROUNDED POSITION?
001F 25 FC	BCS CHKPOS	NO, KEEP LOOKING
0021 D7 40	STB \$40	YES, STORE SWITCH POSITION
0023 3F	SWI	

Flowchart:



The last two programs both need one byte of temporary storage. How would you rewrite each example to use the Hardware Stack for that storage? What are the advantages and disadvantages of using the Stack? Note that it makes the programs reentrant and more general, as well as easier to use since they do not incidentally change specific memory locations.

13-3. A SINGLE LED

We will interface a single light-emitting diode to a 6809 microprocessor, providing separate interfaces and programs to handle positive logic (a '1' lights the LED and a '0' turns it off) and negative logic (a '0' lights the LED and a '1' turns it off).

Figure 13-5 shows the circuitry required to interface an LED. The LED lights when its anode is positive with respect to its cathode (Figure 13-5a). Therefore, you

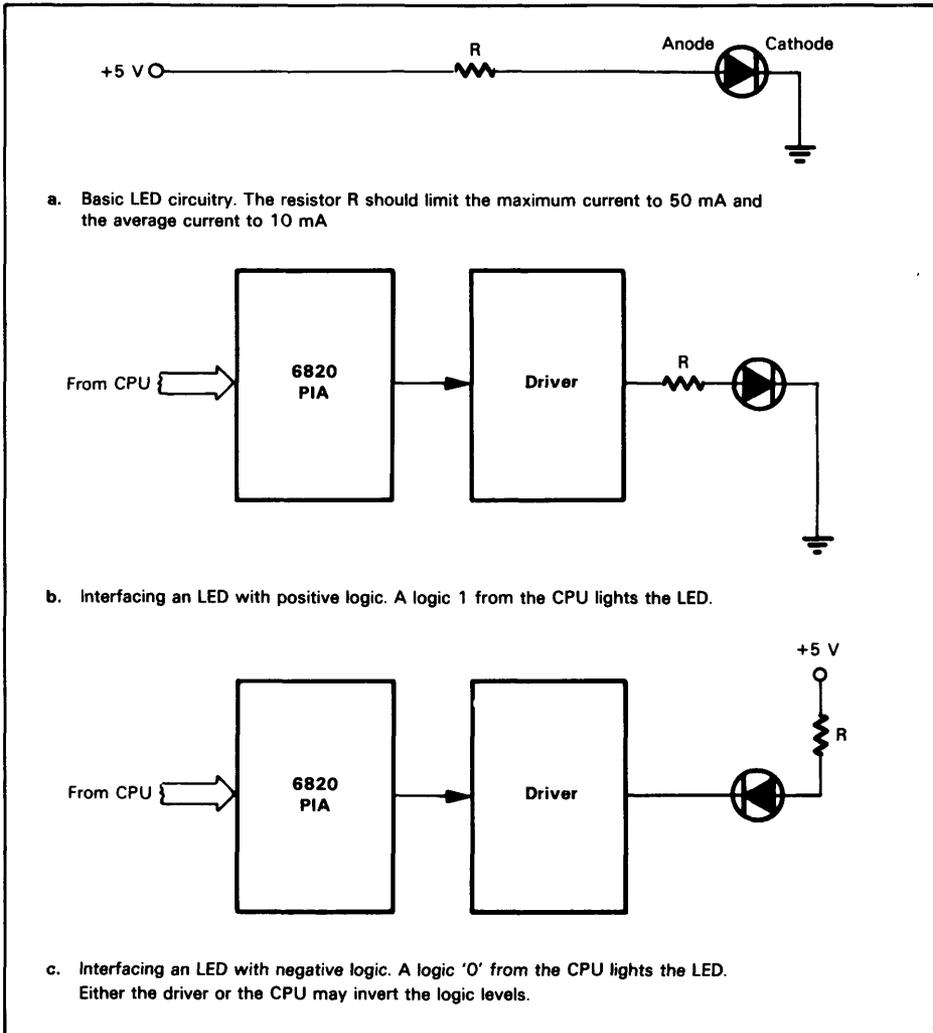


Figure 13-5. Interfacing an LED

can light the LED either by grounding the cathode and having the computer supply a one to the anode (Figure 13-5b) or by connecting the anode to +5 volts and having the computer supply a zero to the cathode (Figure 13-5c). Controlling the cathode is the more common approach since most MOS or TTL I/O ports perform better in this mode. The LED is brightest when it operates from pulsed currents of about 10 or 50 mA applied a few hundred times per second. LEDs have a very short turn-on time (in the microsecond range) so they are well suited to multiplexing (operating several from a single port). LED circuits usually need peripheral or transistor drivers and current-limiting resistors. MOS devices normally cannot drive LEDs directly and make them bright enough for easy viewing.

The PIA has an output latch on each port. However, the B port is normally used

13-22 6809 Assembly Language Programming

for output, since it has somewhat more drive capability. In particular, the B port outputs are capable of driving Darlington transistors (providing 3.2 mA minimum at 1.5V). Darlington transistors are high-gain transistors capable of switching large amounts of current at high speed; they are useful in driving solenoids, relays, and other devices.

Task 13-3. Turn the Light On or Off

Purpose: The program turns a single LED either on or off.

- A. Send a Logic One to the LED** (light a display that operates in positive logic or turn off a display that operates in negative logic).

Program 13-3:

(form data initially)

```

                8003  PIACB  EQU   $8003
                8002  PIADDB EQU   $8002
                8002  PIADB  EQU   $8002
                0080  MASKP  EQU   %10000000
                *
0000                ORG   $0000
0000 7F 8003        CLR   PIACB    ADDRESS DATA DIRECTION REGISTER
0003 86 FF         LDA   #$FF     MAKE ALL DATA LINES OUTPUTS
0005 B7 8002       STA   PIADDB
0008 86 04        LDA   %%00000100 ADDRESS DATA REGISTER
000A B7 8003       STA   PIACB
000D 86 80        LDA   #MASKP    GET DATA FOR LED
000F B7 8002       STA   PIADB     SEND DATA TO LED
0012 3F           SWI
```

We use the B side of the PIA because of the buffering. This allows the CPU to read the data back (if necessary) without any difficulty.

(update data)

```

0013 B6 8002       LDA   PIADB     GET OLD DATA
0016 8A 80         ORA   #MASKP    SET LED OUTPUT BIT TO 1
0018 B7 8002       STA   PIADB
001B 3F           SWI
```

MASKP has a '1' bit in the LED position and zeros elsewhere. Logically ORing the contents of Accumulator A with MASKP leaves the other bit positions unchanged; those positions may control unrelated LEDs. Note that the CPU can read the PIA data register even when some or all the pins have been assigned as outputs.

- B. Send a Logic Zero to the LED** (turn off a display that operates in positive logic or light a display that operates in negative logic).

The differences are that MASKP must be replaced by its logical complement MASKN and ORA.#MASKP must be replaced by ANDA #MASKN. MASKN has a zero bit in the LED position and ones elsewhere. Logically ANDing with MASKN does not affect the other bit positions.

13-4. SEVEN-SEGMENT LED DISPLAY

We will interface a seven-segment LED display to a 6809 microprocessor. The display may be either common-anode (negative logic) or common-cathode (positive logic).

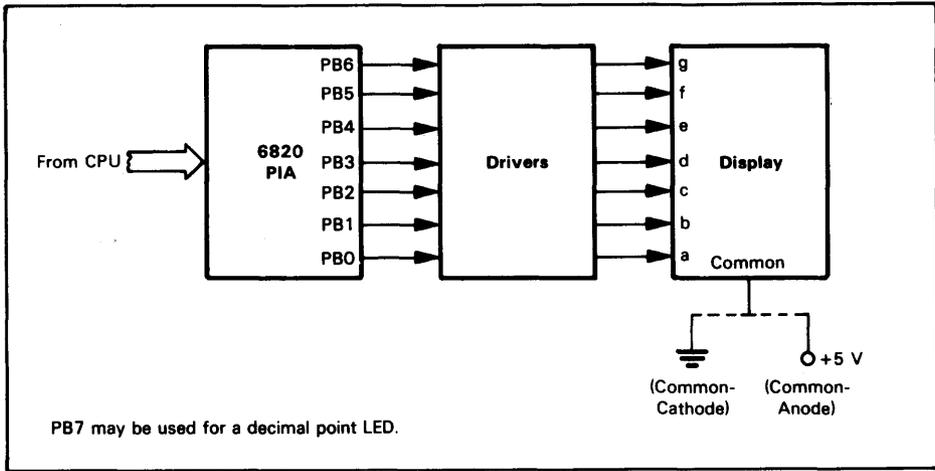
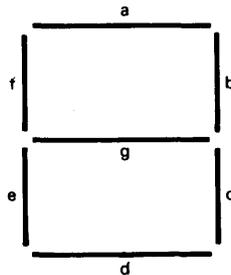


Figure 13-6. Interfacing a Seven-Segment Display

Figure 13-6 shows the circuitry required to interface a seven-segment display. Each segment may have one, two, or more LEDs attached in the same way. There are two ways of connecting the displays. One is tying all the cathodes together to ground (see Figure 13-7a); this is a “common-cathode display,” and a logic one at an anode lights a segment. The other is tying all the anodes together to a positive voltage supply (see Figure 13-7b); this is a “common-anode display,” and a logic zero at a cathode lights a segment. So the common-cathode display uses positive logic and the common-anode display negative logic. Either display requires appropriate drivers and resistors.

The Common line from the display is tied either to ground or to +5 volts. The display segments are customarily labelled:



The seven-segment display is widely used because it contains the smallest number of separately controlled segments that can provide recognizable representations of all the decimal digits (see Figure 13-8 and Table 13-8). Seven-segment displays can also produce some letters and other characters (see Table 13-9). Better representations require a substantially larger number of segments and more circuitry.⁴ Since seven-segment displays are so popular, low-cost seven-segment decoder/drivers have become widely available. The most popular devices are the 7447 common-anode driver and the 7448 common-cathode driver;⁵ these devices have Lamp Test inputs (that turn

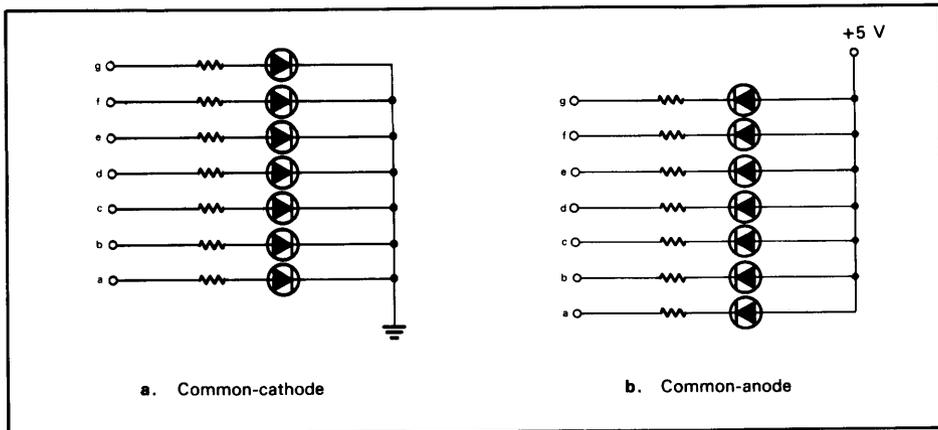


Figure 13-7. Seven-Segment Display Organization

Table 13-8. Seven-Segment Representations of Decimal Numbers

Number	Hexadecimal Representation	
	Common-cathode	Common-anode
0	3F	40
1	06	79
2	5B	24
3	4F	30
4	66	19
5	6D	12
6	7D	02
7	07	78
8	7F	00
9	67	18

Bit 7 is always zero and the others are g, f, e, d, c, b, and a in decreasing order of significance.

all the segments on) and blanking inputs and outputs (for blanking leading or trailing zeros).

Task 13-4a. Display a Decimal Digit

Purpose: Display the contents of memory location 0040 on a seven-segment display if it contains a decimal digit. Otherwise, blank the display.

Sample Problems:

- a. (0040) = 05
Result is 5 on display
- b. (0040) = 66
Result is a blank display

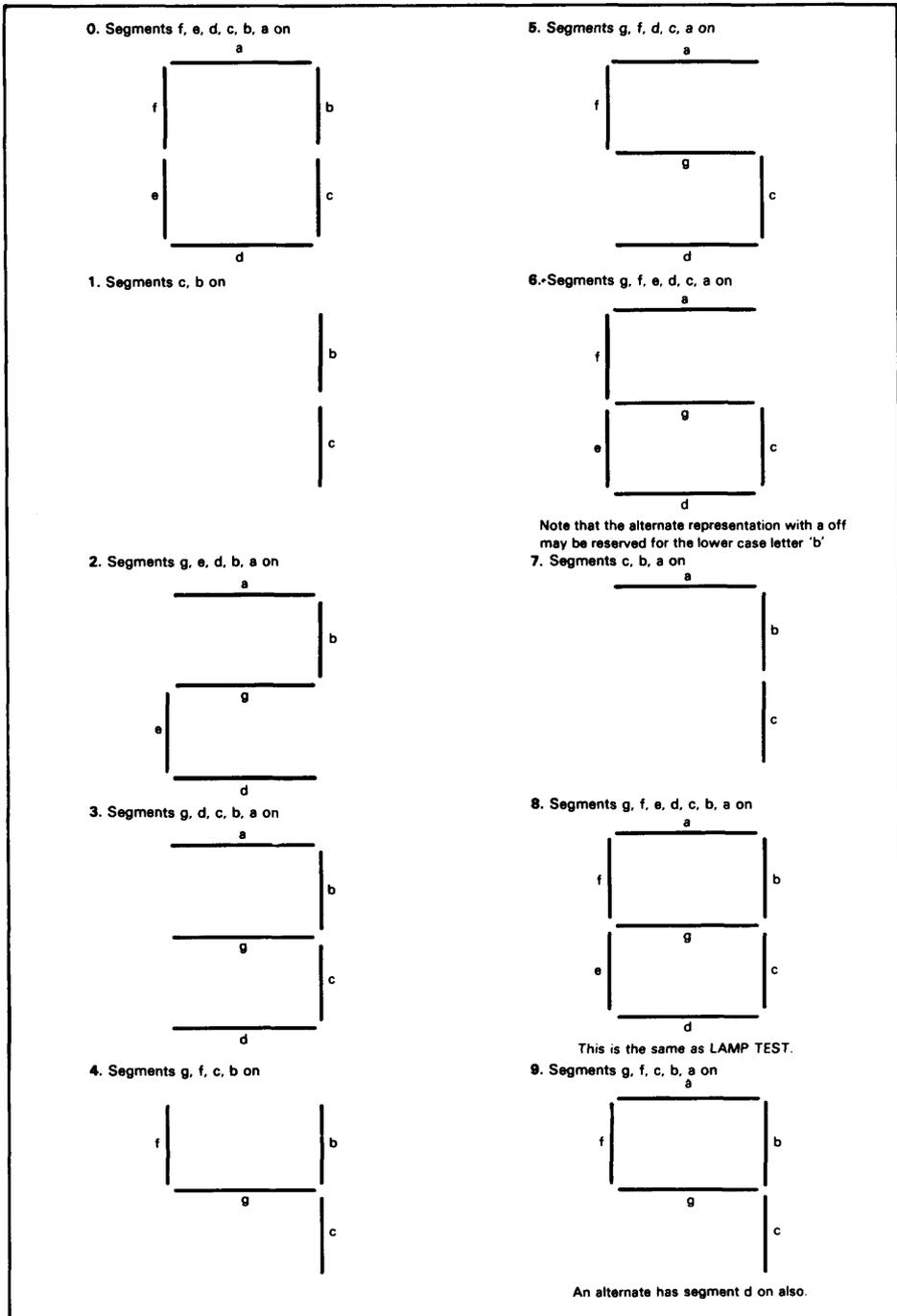


Figure 13-8. Seven-Segment Representation of Decimal Digits

Table 13-9. Seven-Segment Representations of Letters and Symbols

Upper-case Letters			Lower-case Letters and Special Characters		
Letter	Hexadecimal Representation		Character	Hexadecimal Representation	
	Common-cathode	Common-anode		Common-cathode	Common-anode
A	77	08	b	7C	03
C	39	46	c	58	27
E	79	06	d	5E	21
F	71	0E	h	74	0B
H	76	09	n	54	2B
I	06	79	o	5C	23
J	1E	61	r	50	2F
L	38	47	u	1C	63
O	3F	40	-	4C	3F
P	73	0C	?	53	2C
U	3E	41			
Y	66	19			

Source Program:

```

                8003  PIACB EQU  $8003
                8002  PIADB EQU  $8002
                8002  PIADDB EQU $8002
                00FF  BLANK EQU  $FF

                *
0000                ORG  $0000
0000 7F 8003        CLR  PIACB    ADDRESS DATA DIRECTION REGISTER
0003 86 FF          LDA  #$FF     MAKE ALL DATA LINES OUTPUTS
0005 B7 8002        STA  PIADDB
0008 86 04          LDA  #0000100 ADDRESS DATA REGISTER
000A B7 8003        STA  PIACB
000D C6 FF          LDB  #BLANK   GET BLANK CODE
000F 96 40          LDA  $40      GET DATA
0011 81 09          CMPA #9       IS DATA A DECIMAL DIGIT (9 OR
                                LESS)?
0013 22 05          *          BHI  DSPLY   NO, DISPLAY BLANK CODE
0015 8E 011E        LDX  #SSEG   YES, CONVERT DIGIT TO SEVEN-
0018 E6 86          LDB  A,X     SEGMENT CODE
001A F7 8002        DSPLY STB  PIADB  SEND CODE TO DISPLAY
001D 3F             SWI
    
```

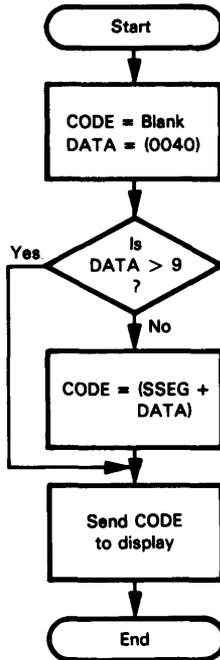
BLANK is 00 for a common-cathode display, FF for a common-anode display. An alternative procedure would be to put the blank code at the end of the table and replace all improper data values with 10; i.e., the instructions after STA PIACRB are:

```

LDA  $40      GET DATA
CMPA #9      IS DATA A DECIMAL DIGIT (9 OR LESS)?
BLS  CNVRT
LDA  #10     NO, REPLACE DATA WITH INDEX FOR BLANK
                                CODE
CNVRT LDX #SSEG  CONVERT DATA TO SEVEN-SEGMENT CODE
LDB  A,X
STB  PIADRB  SEND CODE TO DISPLAY
SWI
    
```

Table SSEG is either the common-cathode or the common-anode representation of the decimal digits from Table 13-8 with the appropriate blank code in the tenth position.

Figure 13-9 shows how to multiplex displays (i.e., drive several displays from the same port).⁶ A brief pulse on control line CB2 automatically clocks the decade counter

Flowchart:

after each output operation, thus directing the data to the next display. RESET initializes the decade counter to 9 so that the first output operation clears the counter and directs data to the first (actually, the zeroth) display.

The next program uses a transparent 1 ms delay routine (described earlier in this chapter) to pulse each of ten common-cathode displays for 1 ms. An observer will see a continuous ten-digit display much like the ones typical of electronic calculators, watches, and point-of-sale terminals.

Task 13-4b. Display Ten Decimal Digits

Purpose: Continuously display the contents of memory locations 0040 through 0049 on ten 7-segment displays that are multiplexed with a counter and a decoder. The most significant (leftmost) digit is in memory location 0040.

Sample Problem:

```

(0040) = 66
(0041) = 3F
(0042) = 7F
(0043) = 7F
(0044) = 06
(0045) = 5B
(0046) = 07
(0047) = 4F
(0048) = 6D
(0049) = 7D
  
```

The number on the displays is 4088127356

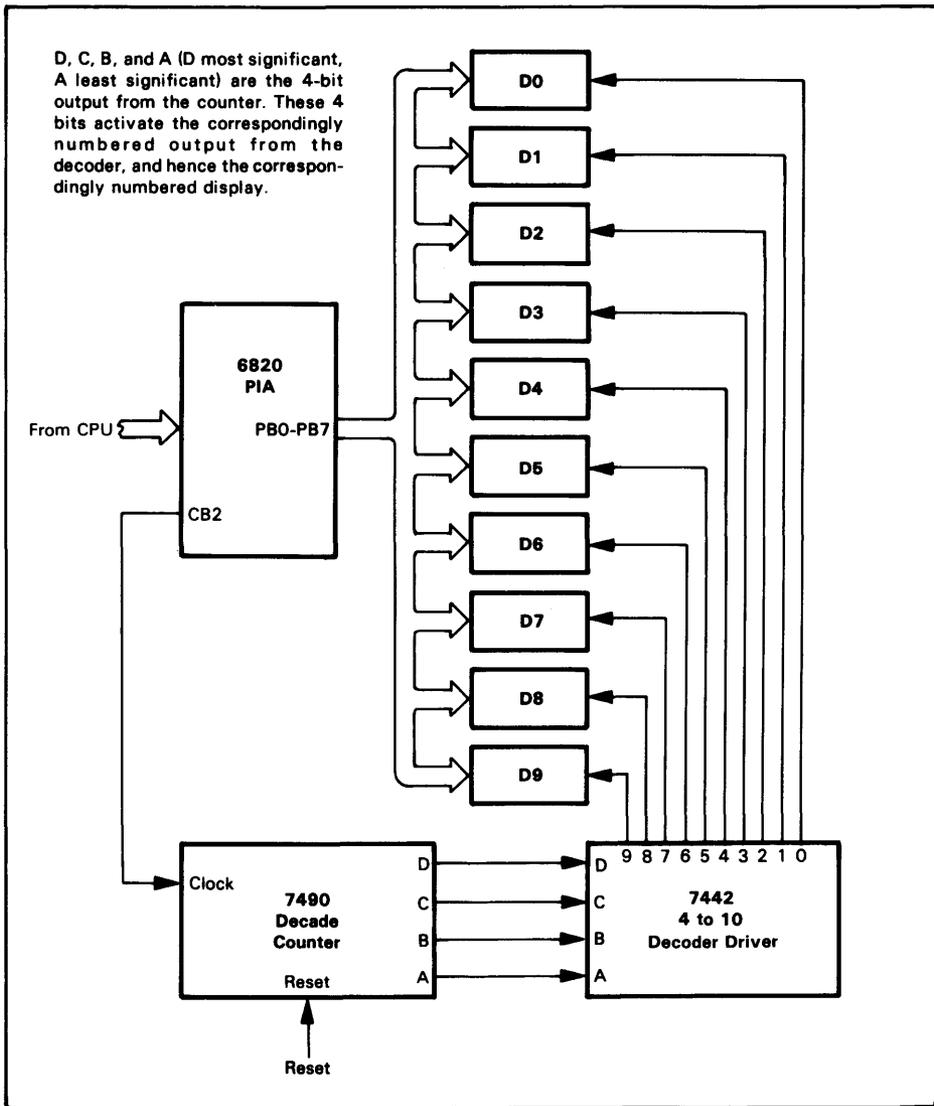


Figure 13-9. Multiplexed Seven-Segment Displays

Program 13-4b:

```

0030    DELAY EQU    $0030
8003    PIACB EQU    $8003
8002    PIADDB EQU   $8002
8002    PIADB EQU    $8002
      *
0000    ORG     $0000
0000 7F 8003    CLR     PIACB    ADDRESS DATA DIRECTION REGISTER
0003 86 FF      LDA     #$FF    MAKE ALL DATA LINES OUTPUTS
0005 B7 8002    STA     PIADDB
0008 86 2C      LDA     #$00101100 MAKE CB2 A BRIEF PULSE,
      *                               ADDRESS I/O
000A B7 8003    STA     PIACB
000D 8E 0040    SCAN    LDX     #$40    POINT TO START OF DATA
0010 C6 0A      LDB     #10    NUMBER OF DISPLAYS = 10
0012 A6 80      DSPLY   LDA     ,X+    GET DATA FOR A DISPLAY
0014 B7 8002    STA     PIADB    SEND DATA TO DISPLAY
0017 9D 30      JSR     DELAY    WAIT 1 MS
0019 5A        DECB    COUNT DISPLAYS
001A 26 F6      BNE     DSPLY
001C 20 EF      BRA     SCAN    START ANOTHER SCAN

```

Control register bit 5 = 1 to make CB2 an output, bit 4 = 0 to make it a pulse, and bit 3 = 1 to make it a brief pulse lasting one clock cycle. We have assumed here that subroutine DELAY (starting at address 0030) provides a transparent 1 ms wait (i.e., it does not affect any registers).

MORE COMPLEX I/O DEVICES

More complex I/O devices differ from simple keyboards, switches, and displays in that:

1. They transfer data at higher rates.
2. They may have their own internal clocks and timing.
3. They produce status information and require control information, as well as transferring data.

Because of their high data rates, you cannot handle these I/O devices casually. If the processor does not provide the appropriate service, the system may miss input data or produce erroneous output data. You are therefore working under much more exacting constraints than in dealing with simpler devices. Interrupts are a convenient method for handling complex I/O devices, as we shall see in Chapter 15.

SYNCHRONIZATION

Peripherals such as keyboards, teletypewriters, cassettes, and floppy disks produce their own internal timing. These devices provide streams of data separated by specific timing intervals. **The computer must synchronize the initial input or output operation with the peripheral clock and then provide the proper interval between subsequent operations. A simple delay loop like the one shown previously can produce the time interval. The synchronization may require one or more of the following procedures:**

1. **Looking for a transition on a clock or strobe line provided by the peripheral for timing purposes.** The simplest method is to tie the strobe to a PIA control

line and wait until the appropriate bit of the PIA control register is set.

2. **Finding the center of the time interval during which the data is stable.** We would prefer to determine the value of the data at the center of the pulse rather than at the edges, where the data may be changing. Finding the center requires a delay of one-half of a transmission interval (bit time) after the edge. Sampling the data at the center also means that small timing errors have little effect on the accuracy of the reception.
3. **Recognizing a special starting code.** This is easy if the code is a single bit or if we have some timing information. The procedure is more complex if the code is long and could start at any time. Shifting will be necessary to determine where the transmitter is starting its bits, characters, or messages (this is often called a search for the correct "framing").
4. **Sampling the data several times.** This reduces the probability of receiving data incorrectly from noisy lines. Majority logic (such as best 3 out of 5 or 5 out of 8) can be used to decide on the actual data value.

Reception is, of course, much more difficult than transmission, since the peripheral controls the reception and the computer must interpret timing information generated by the peripheral. In transmission, the computer provides the proper timing and formatting for a specific peripheral.

CONTROL AND STATUS INFORMATION

Peripherals may require or provide other information besides data and timing. We refer to other information transmitted by the computer as "control information;" it may select modes of operation, start or stop processes, clock registers, enable buffers, choose formats or protocols, provide operator displays, count operations, or identify the type and priority of the operation. **We refer to other information transmitted by the peripheral as "status information;"** it may indicate the mode of operation, the readiness of devices, the presence of error conditions, the format of protocol in use, and other states or conditions.

The computer handles control and status information just like data. This information seldom changes, even though actual data may be transferred at a high rate. The control or status information may be single bits, digits, bytes, or multiple bytes. Often single bits or short fields are combined and handled by a single input or output port.

Separating Status Information

Combining status and control information into bytes reduces the total number of I/O port addresses required by the peripherals. However, the combination does mean that individual status input bits must be separately interpreted and control output bits must be separately determined. **The procedure for isolating status bits is as follows:**

- Step 1. Read status data from the peripheral.**
- Step 2. Logical AND with a mask** (the mask has ones in bit positions that must be examined, and zeros elsewhere).
- Step 3. Shift the separated bits to the least significant bit positions.**

Step 3 is unnecessary if the field is a single bit, since the Zero flag will contain the

complement of that bit after Step 2 (try it!). A Shift or Load instruction can replace Step 2 if the field is a single bit and occupies the least significant, most significant, or next to most significant bit position (positions 0, 7, or 6). These positions are often reserved for the most frequently used status information. You should try to write the required instruction sequences for the 6809 processor. Note, in particular, the use of the Bit Test instruction. This instruction performs a logical AND between the contents of the Accumulator and the contents of a memory location but does not save the result; the flags are set as follows:

Zero flag = 1 if the logical AND produces a zero result, 0 if it does not.

Sign flag = bit 7 of the result of the logical AND.

Combining Control Information

This is the procedure for setting and clearing control bits:

Step 1. Read prior control information.

Step 2. Logical AND with mask to clear bits (mask has zeros in bit positions to be cleared, ones elsewhere).

Step 3. Logically OR with mask to set bits (mask has ones in bit positions to be set, zeros elsewhere).

Step 4. Send new control information to peripheral.

Here again the procedure is simpler if the field is a single bit and occupies a position at either end of a data byte.

Some examples of separating and combining status bits are:

1. A 3-bit field in bit positions 2 through 4 of a PIA Data Register is a scaling factor. Place that factor in Accumulator A.

```
*
*READ STATUS DATA FROM INPUT PORT
*
    LDA    PIADR        READ STATUS DATA
*
*MASK OFF UNWANTED BITS AND SHIFT RESULT
*
    ANDA  #%00011100  SAVE SCALING FACTOR
    LSRA  LSRA        SHIFT TWICE TO NORMALIZE
    LSRA  LSRA
```

2. Accumulator A contains a 2-bit field that must be placed in bit positions 3 and 4 of a PIA Data Register.

```
*
*MOVE DATA TO FIELD POSITIONS
*
    ASLA  ASLA        SHIFT DATA TO BIT POSITIONS 3 AND 4
    ASLA  ASLA
    ANDA  #%00011000  CLEAR OUT OTHER BITS
*
*COMBINE NEW FIELD POSITIONS WITH OTHER DATA
*
    PSHS  A           SAVE NEW FIELD VALUE AT TOP OF STACK
    LDA  PIADR        CLEAR OLD FIELD VALUE
    ANDA  %%11100111
    ORA  ,S+         INSERT NEW FIELD VALUE AND CLEAR STACK
    STA  PIADR
```

The instruction ORA ,S+ not only logically ORs the accumulator with the data we

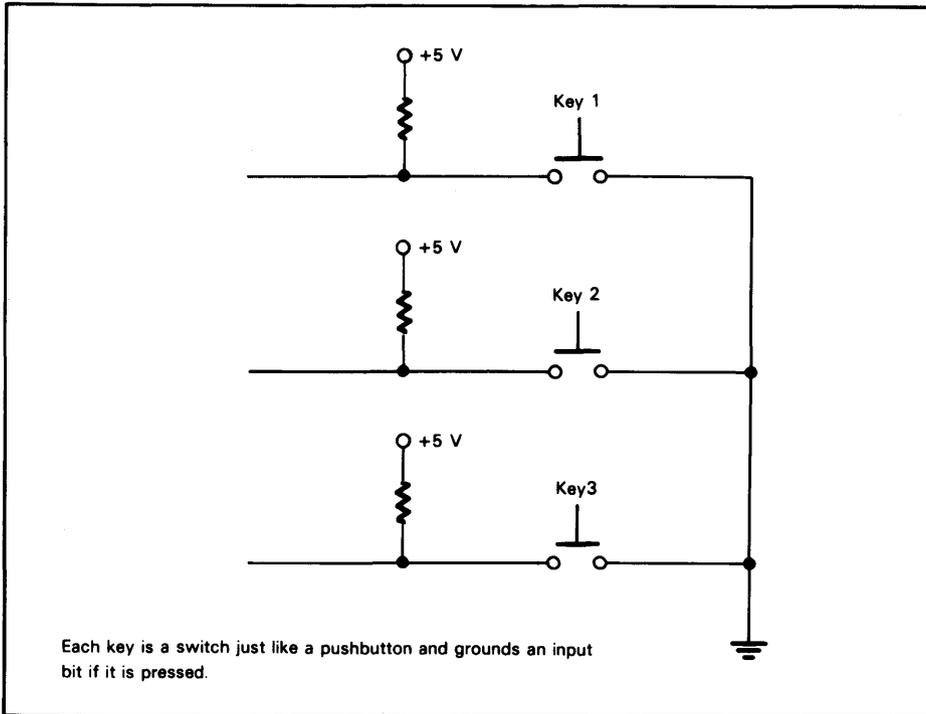


Figure 13-10. A Small Keyboard

saved at the top of the Hardware Stack, but it also increments the Hardware Stack Pointer and thus removes the data from the Stack.

Documenting Status and Control Transfers

Documentation is a serious problem in handling control and status information. **The meanings of status inputs or control outputs are seldom obvious. The programmer should clearly indicate the purposes of input and output operations in the comments,** for example, "CHECK IF READER IS ON," "CHOOSE EVEN PARITY OPTION," or "ACTIVATE BIT RATE COUNTER." The Logical and Shift instructions will otherwise be very difficult to remember, understand, or debug.

13-5. AN UNENCODED KEYBOARD

The processor will recognize a key closure from an unencoded 3×3 keyboard and place the number of the key that was pressed in Accumulator B.

Keyboards are just collections of switches (see Figure 13-10). Small numbers of keys are easiest to handle if each key is attached separately to a bit of an input port. Interfacing the keyboard is then the same as interfacing a set of switches.

Matrix Keyboard

Keyboards with more than eight keys require more than one input port and therefore multibyte operations. This is particularly wasteful if the keys are logically separate, as in a calculator or terminal keyboard where the user will only strike one at a time. **The number of input lines required may be reduced by connecting the keys into a matrix, as shown in Figure 13-11. Now each key represents a potential connection between a row and a column.** The keyboard matrix requires $n + m$ external lines, where n is the number of rows and m is the number of columns. This compares to $n \times m$ external lines if each key is separate. Table 13-10 compares the number of keys required by typical configurations.

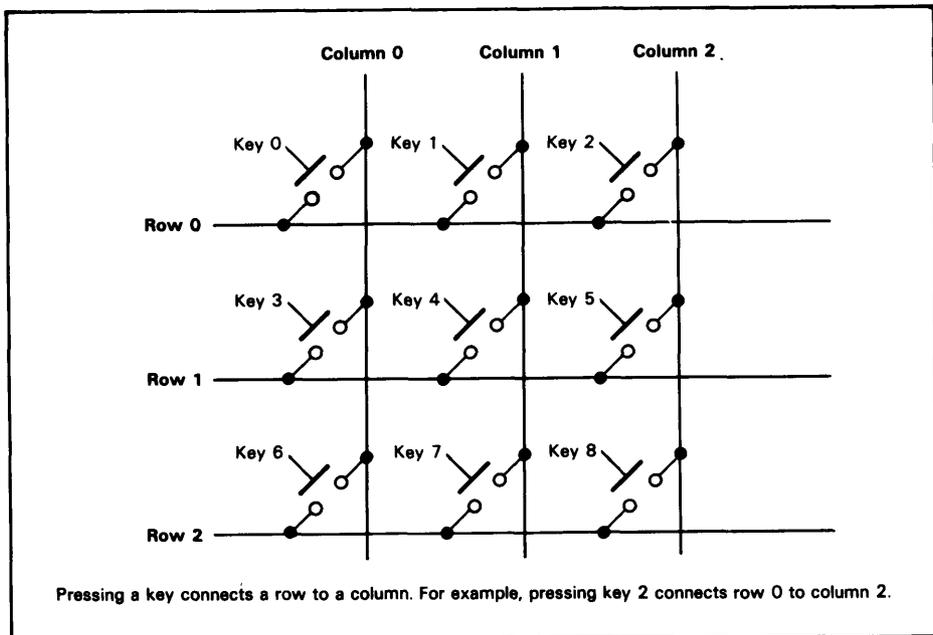


Figure 13-11. A Keyboard Matrix

Table 13-10. Comparison Between Independent Connections and Matrix Connections for Keyboards

Keyboard Size	Number of Lines with Independent Connections	Number of Lines with Matrix Connections
3 × 3	9	6
4 × 4	16	8
4 × 6	24	10
5 × 5	25	10
6 × 6	36	12
6 × 8	48	14
8 × 8	64	16

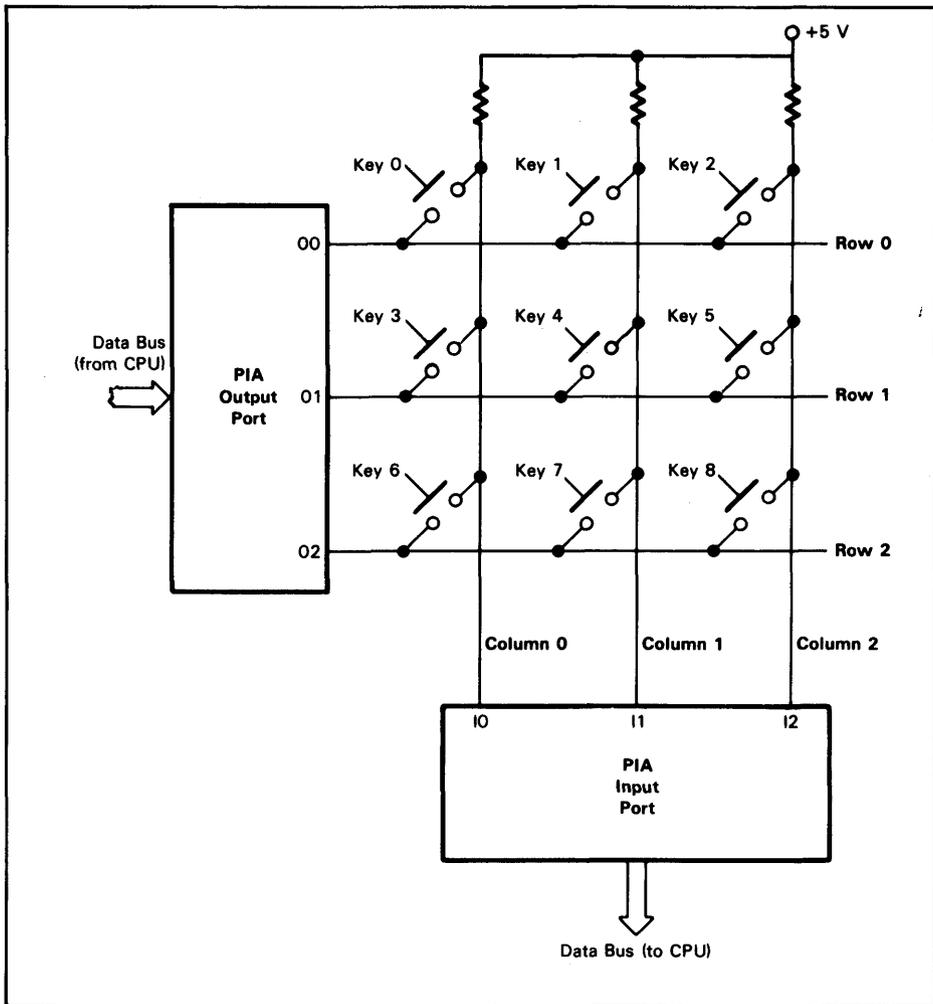


Figure 13-12. I/O Arrangement for a Keyboard Scan

Keyboard Scan

A program can determine which key has been pressed by using the external lines from the matrix. The usual procedure is a "keyboard scan." We ground Row 0 and examine the column lines. If any lines are grounded, a key in that row has been pressed, causing a row-to-column connection. We can determine which key was pressed by determining which column line is grounded; that is, which bit of the input port is zero. If no column line is grounded, we proceed to Row 1 and repeat the scan. Note that we can check to see if any keys at all have been pressed by grounding all the rows at once and examining the columns.

The keyboard scan requires that the row lines be tied to an output port and the column lines to an input port. Figure 13-12 shows the arrangement. The CPU can

ground a particular row by placing a zero in the appropriate bit of the output port and ones in the other bits.

The CPU can determine the state of a particular column by examining the appropriate bit of the input port.

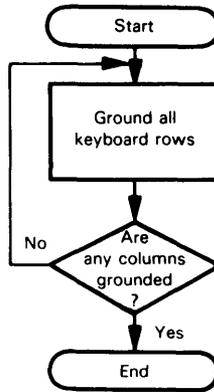
Task 13-5a. Wait for Key Closure

Purpose: Wait for a Key to be Pressed

The procedure is as follows:

1. Ground all the rows by clearing all the output bits.
2. Fetch the column inputs by reading the input port.
3. Return to Step 1 if all the column inputs are ones.

Flowchart:



Program 13-5a:

```

8001 PIACA EQU $8001
8000 PIADDA EQU $8000
8000 PIADA EQU $8000
8003 PIACB EQU $8003
8002 PIADDB EQU $8002
8002 PIADB EQU $8002
*
0000 ORG $0000
0000 7F 8001 CLR PIACA ADDRESS DATA DIRECTION REGISTERS
0003 7F 8003 CLR PIACB
0006 7F 8000 CLR PIADDA MAKE A SIDE DATA LINES INPUTS
0009 86 FF LDA #$FF
000B B7 8002 STA PIADDB MAKE B SIDE DATA LINES OUTPUTS
000E 86 04 LDA ##00000100 ADDRESS DATA REGISTERS
0010 B7 8001 STA PIACA
0013 B7 8003 STA PIACB
0016 7F 8002 CLR PIADB GROUND ALL KEYBOARD ROWS
0019 B6 8000 WAITK LDA PIADA GET DATA FROM KEYBOARD COLUMNS
001C 84 07 ANDA ##00000111 MASK COLUMN BITS
001E 81 07 CMPA ##00000111 ARE ANY KEYS CLOSED?
0020 27 F7 BEQ WAITK NO, WAIT
0022 3F SWI
    
```

Masking off the column bits eliminates any problems that could be caused by the states of the unused input lines.

13-36 6809 Assembly Language Programming

We could generalize the routine by naming the output and masking patterns:

```

ALLGND EQU %11111000
OPEN    EQU %00000111
    
```

We could then use these names in the program; changing to a different keyboard would require only a change in the definitions and a re-assembly.

Of course, a 3×3 or 4×4 keyboard only needs one port of a PIA. Rewrite the program to use only port A.

Task 13-5b. Identify Key

Purpose: Identify a key closure by placing the number of the key in Accumulator B. The procedure is as follows:

1. Set key number to 1, keyboard output port to all ones except for a zero in bit 0, and row counter to number of rows.
2. Fetch the column inputs by reading the input port.
3. If any column inputs are zero, proceed to Step 7.
4. Add the number of columns to the key number to reach next row.
5. Update the contents of the output port by shifting the zero bit left one position.
6. Decrement row counter. Go to Step 2 if any rows have not been scanned; otherwise, go to Step 9.
7. Add 1 to key number. Shift column inputs right one bit.
8. If Carry = 1, return to Step 7.
9. End of program.

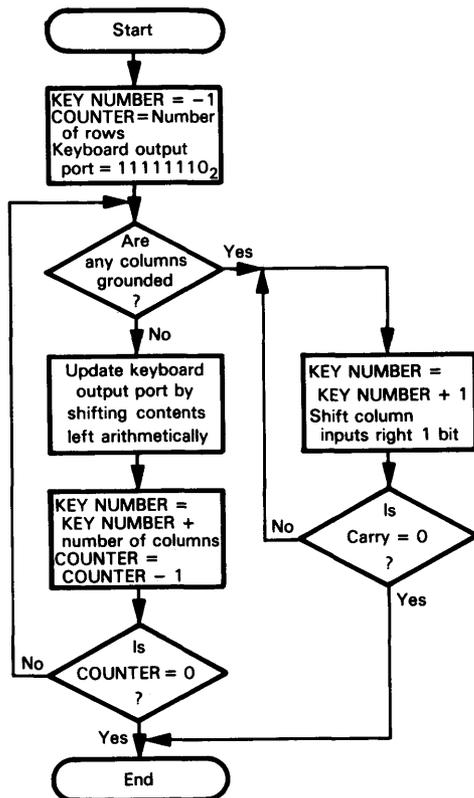
This program does not wait for the operator to press a key, so the key must be pressed before the program is executed. How would you modify the program to wait for at least one key to be pressed?

Program 13-5b:

```

      8001   PIADA EQU $8001
      8000   PIADDA EQU $8000
      8000   PIADA EQU $8000
      8003   PIACB EQU $8003
      8002   PIADDB EQU $8002
      8002   PIADB EQU $8002
      *
0000                ORG $0000
0000 7F 8001        CLR PIACA    ADDRESS DATA DIRECTION REGISTERS
0003 7F 8003        CLR PIACB
0006 7F 8000        CLR PIADDA   MAKE A SIDE DATA LINES INPUTS
0009 86 FF         LDA #$FF     MAKE B SIDE DATA LINES OUTPUTS
000B B7 8002       STA PIADDB
000E 86 04         LDA #%00000100 ADDRESS DATA REGISTERS
0010 B7 8001       STA PIACA
0013 B7 8003       STA PIACB
0016 86 FE         LDA #%11111110 START BY GROUNDING ROW ZERO
0018 B7 8002       STA PIADB
001B 86 03         LDA #3       COUNTER = NUMBER OF ROWS
001D 97 40         STA $40
001F C6 FF         LDB #$FF    KEY NUMBER = -1
0021 B6 8000       LDA PIADA    GET COLUMN INPUTS
0024 84 07         ANDA #%00000111 MASK OFF COLUMN BITS
0026 81 07         CMPA  #%00000111 ARE ANY KEYS CLOSED IN THIS
      *                               ROW?
    
```

0028 26 0A	BNE	FCOL	YES, DETERMINE WHICH ONE
002A CB 03	ADDB	#3	NO, PROCEED TO NEXT ROW
002C 78 8002	ASL	PIADB	UPDATE SCAN PATTERN
002F 0A 40	DEC	\$40	CONTINUE IF ANY ROWS NOT SCANNED
0031 26 EE	BNE	FROW	
0033 3F	SWI		
0034 5C	FCOL	INCB	KEY NUMBER = KEY NUMBER + 1
0035 44	LSRA		IS THIS COLUMN GROUNDED?
0036 25 FC	BCS	FCOL	NO, EXAMINE NEXT COLUMN
0038 3F	SWI		YES, KEY CLOSURE IDENTIFIED

Flowchart:

Each time a row scan fails, we must add the number of columns (3 in the example) to the key number to move to the next row (try the procedure on the keyboard in Figure 13-12).

We could generalize the program by making the number of rows, the number of columns, and the masking patterns into named parameters with Equate (EQU) directives.

What result does the program produce in Accumulator B if no keys are being pressed? Change the program so that it starts the scan over again in that case.

An alternative approach is to use the bidirectional capability of the PIA.⁷ The procedure would be:

1. Ground all columns and save the row inputs.

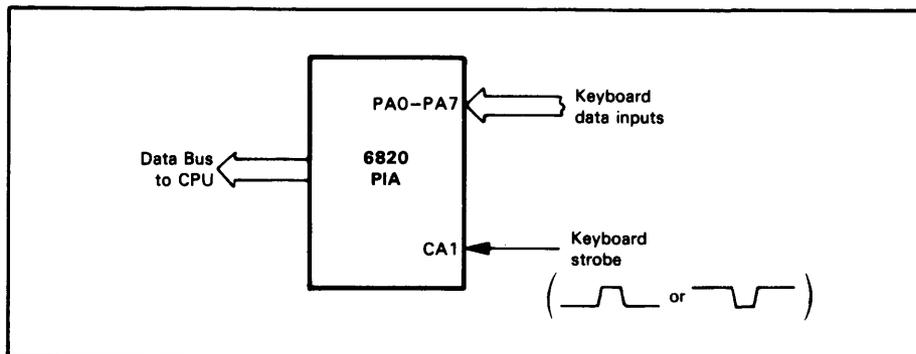


Figure 13-13. I/O Interface for an Encoded Keyboard

2. Ground all rows and save the column inputs.
3. Use both the row and the column inputs to determine the key number from a table.

Write a program to implement this procedure.

13-6. AN ENCODED KEYBOARD⁸

The processor will fetch data, when it is available, from an encoded keyboard that provides a strobe along with each data transfer.

An encoded keyboard provides a unique code for each key. It has internal electronics that perform the scanning and identification procedure of the previous example. The tradeoff is between the simpler software required by the encoded keyboard and the lower cost of the unencoded keyboard.

Encoded keyboards may use diode matrices, TTL encoders, or MOS encoders. The codes may be ASCII, EBCDIC, or a custom code. PROMs are often part of the encoding circuitry.

The encoding circuitry may do more than just encode key closures. It may also debounce the keys and handle "rollover," the problem of more than one key being struck at the same time. Common ways of handling rollover are "2-key rollover," in which two keys (but not more) struck at the same time are resolved into separate closures, and "n-key rollover," in which any number of keys struck at the same time are resolved into separate closures.

The encoded keyboard also provides a strobe with each data transfer. The strobe indicates that another key has been pressed. Figure 13-13 shows the interface between an encoded keyboard and the 6809 microprocessor. We tie the keyboard strobe line to input CA1; a pulse on the strobe line sets bit 7 of the PIA Control Register. Bit 1 of the Control Register determines which edge (leading or trailing) of the pulse the PIA recognizes. Bit 1 = 0 to recognize the trailing edge (high-to-low transition), and 1 to recognize the leading edge (low-to-high transition).

The PIA thus contains an edge-sensitive latched serial status port as well as a parallel data port. It also contains an inverter that allows it to recognize either edge of a pulse. A PIA can therefore replace many simple circuit elements, such as flip-flops, gates, inverters, and buffers. The designer can correct errors by changing the contents of

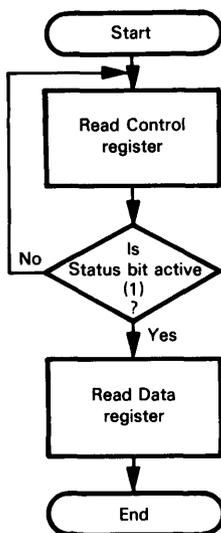
the PIA control register (a simple software change) rather than by rewiring a bread-board. For example, changing the active edge on the strobe line requires the changing of one bit in a program, whereas it might require additional parts and rewiring on a bread-board.

Be careful, however, of the fact that the PIA does not contain an input latch. An actual interface may require a latch if the keyboard is not guaranteed to hold its data. The latch can also be controlled by the strobe signal.

Task: Wait for an active-low strobe on control line CA1 and then load the keyboard data into Accumulator A.

Note that reading the data from the Data Register clears the status bit (this circuitry is part of the 6820 PIA).

Flowchart:



The hardware must hold the control lines in a logic one state while RESET is active to prevent the accidental setting of status flags. An initial read of the Data Registers in the startup routine may be used to clear the PIA status bits.⁹

Program 13-6:

```

      8001  PIACA EQU $8001
      8000  PIADDA EQU $8000
      8000  PIADA EQU $8000
      *
0000                                ORG $0000
0000 7F 8001  CLR PIACA  ADDRESS DATA DIRECTION REGISTER
0003 7F 8000  CLR PIADDA  MAKE ALL DATA LINES INPUTS
0006 86 04    LDA #00000100 ADDRESS DATA REGISTER
0008 B7 8001  STA PIACA
000B B6 8001  KBWAIT LDA PIACA  HAS KEY BEEN PRESSED?
000E 2A FB    BPL KBWAIT  NO, WAIT
0010 B6 8000  LDA PIADA  YES, FETCH DATA FROM KEYBOARD
0013 3F      SWI
  
```

To set control register bit 7 on low-to-high transitions on the keyboard strobe line, simply replace LDA #00000100 with LDA #00000110.

If we tied the keyboard strobe line to CA2, control register bit 6 would then serve as the status latch.

Show that reading the Data Register clears the status bit, indicating that the CPU has read the most recent data and allowing the next input operation to occur. Hint: Save the contents of the PIA control register in memory before and after the instruction LDA PIADA is executed. What happens if you replace LDA with STA? How about TST, CLR, COM, ADD? Remember that writing data into the Data Register does not clear the status bit, nor does writing data into or reading data from the control register. What happens if you replace LDA PIADA with LDA PIACA or STA PIACA?

One reason why we are concerned with the effects of instructions on PIA registers is that we may want to use the control lines for purposes that have nothing to do with the data ports. For example, we may be using a PIA to interface a simple peripheral (e.g., a set of switches or single LEDs) that does not require any status or control lines. The control lines are then available as serial I/O lines at no additional hardware cost. The only problem is that we must manipulate these lines using facilities that are provided on the assumption of a direct connection between the serial lines and the parallel data port.

13-7. A DIGITAL-TO-ANALOG CONVERTER¹⁰⁻¹³

The processor sends data to an 8-bit digital-to-analog converter, which has an active-low latch enable.

Digital-to-analog converters produce the continuous signals required by solenoids, relays, actuators, and other electrical and mechanical output devices. Typical converters consist of switches and resistor ladders with the appropriate resistance values. The user must generally provide a reference voltage and some other digital and analog circuitry, although complete units are becoming available at low cost.

Figure 13-14 describes the 8-bit Signetics NE5018 D/A converter, which contains an on-chip 8-bit parallel data input latch. A low level on the LE (Latch Enable) input gates the input data into the latches, where it remains after LE goes high.

D/A Converter Interface

Figure 13-15 illustrates the interfacing of the NE5018 to a 6809 system. Note that port B of the PIA automatically produces the active-low pulse required to latch the data into the D/A converter; CB2 acts as an OUTPUT READY signal, indicating that the CPU has sent data to the output port. Remember that in the brief pulse mode, CB2 goes low automatically on the clock pulse following a write operation on Data Register B, and remains low until the next clock pulse (see Table 13-5). The control register bits that cause the PIA to operate this way are:

Bit 5 = 1 to make CB2 an output

Bit 4 = 0 to make CB2 a pulse

Bit 3 = 1 to make CB2 a brief pulse that typically lasts one clock cycle. (Enable pulses and clock pulses are the same in typical 6809 systems).

Note that the PIA contains an output latch. The data therefore remains stable during and after conversion, even though the processor only leaves it on the data bus for one clock cycle. Output latches are essential in microcomputer systems, since the pro-

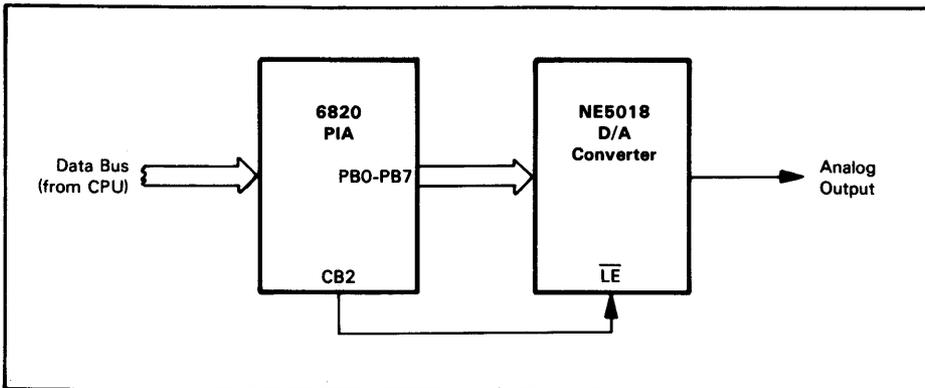


Figure 13-15. I/O Interface for an 8-Bit Digital-to-Analog Converter

cessor uses its data bus constantly to transfer data and instructions to and from memory. The converter typically requires only a few microseconds to produce analog outputs, but other peripherals may need the data for much longer periods.

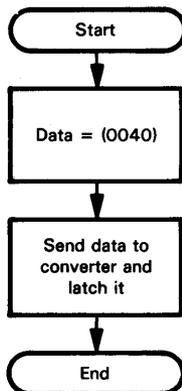
In applications where eight bits of resolution are not enough, you can use the widely available 10 to 16-bit converters. Ones that are advertised as “microprocessor-compatible” usually have separate data ports for the most and least significant bytes. Such devices are much easier to interface than devices that only accept all the data at one time through a single port.

The PIA serves as both a parallel data port and a serial control port. CB2 provides a pulse that lasts one clock cycle after the CPU latches output data into the PIA. This pulse is long enough to meet the requirements (typically 400 ns) of the NE5018 converter.

Task 13-7. Send Data to D/A Converter

Purpose: Send data from memory location 0040 to the D/A converter.

Flowchart:



Program 13-7:

```

      8003   PIACB EQU   $8003
      8002   PIADDB EQU  $8002
      8002   PIADB EQU   $8002
      *
0000                   ORG   $0000
0000 7F   8003         CLR   PIACB   ADDRESS DATA DIRECTION REGISTER
0003 86   FF           LDA   #$FF     MAKE ALL DATA LINES OUTPUTS
0005 B7   8002         STA   PIADDB
0008 86   2C           LDA   ##00101100 ADDRESS DATA REGISTER,
000A B7   8003         STA   PIACB   PROVIDE BRIEF STROBE
000D D6   40           LDB   $40     GET DATA
000F F7   8002         STB   PIADB   SEND DATA TO D/A CONVERTER
      *                                     AND LATCH
0012 3F                   SWI

```

The PIA produces the Load pulse automatically after the CPU stores the data in the Data Register. No explicit instructions are necessary. Although automatic operations like this save time and memory, they also result in documentation problems since there is no record in the program of when they occur. To understand the operation of this interface, you would need a detailed understanding of the 6820 device as well as a hardware schematic and a program listing. Such requirements make maintenance and updating difficult.

The automatic pulse lasts only one clock cycle. If this is not long enough (or if an active-high pulse is necessary), we could use the level output from CB2. This operating mode is often called a manual mode, since the PIA does not produce any pulses automatically. The program to use the mode would be:

```

CLR   PIACB           ADDRESS DATA DIRECTION REGISTER
LDA   #$FF           MAKE ALL DATA LINES OUTPUTS
STA   PIADDB
LDA   ##00110100     ADDRESS DATA REGISTER, PULSE LOW
STA   PIACB
LDB   $40           GET DATA
STB   PIADB         SEND DATA TO DAC
ORA   ##00001000     OPEN DAC LATCH (BRING ENABLE HIGH)
STA   PIACB
ANDA  ##11110111     LATCH DATA (BRING ENABLE LOW)
STA   PIACB
SWI

```

This approach requires more instructions, but it produces a longer pulse and is easier to understand. Here bit 4 of the PIA control register is set to make CB2 a level with the value of bit 3. We can then set and clear bit 3 using the logical instructions (OR with '1' to set, AND with '0' to clear).

In the level or manual mode, CB2 is completely independent of operations on the parallel data port. It is simply a serial output that is available for any purpose. The only precaution one must take in using it is to avoid changing any of the other bits in the PIA control register, since they have unrelated functions. Using the logical OR and AND instructions makes the procedure independent of the contents of the PIA control register, since only bit 3 is changed.

13-8. ANALOG-TO-DIGITAL CONVERTER¹⁴⁻¹⁹

The processor fetches data from an 8-bit analog-to-digital converter that requires a Start Conversion pulse to start the conversion process and provides an End of Conversion output to indicate the completion of the process and the availability of valid data.

Analog-to-digital converters handle the continuous signals produced by various types of sensors and transducers. The converter produces the digital input that the computer requires.

One form of analog-to-digital converter is the successive approximation device, which makes a direct 1-bit comparison during each clock cycle. Such converters are fast but have little noise immunity. Dual slope integrating converters are another form of analog-to-digital converter. These devices take longer to convert data but are more resistant to noise. Other techniques, such as the incremental charge balancing technique, are also used.

Analog-to-digital converters usually require some external analog and digital circuitry, although complete units are becoming available at low cost.

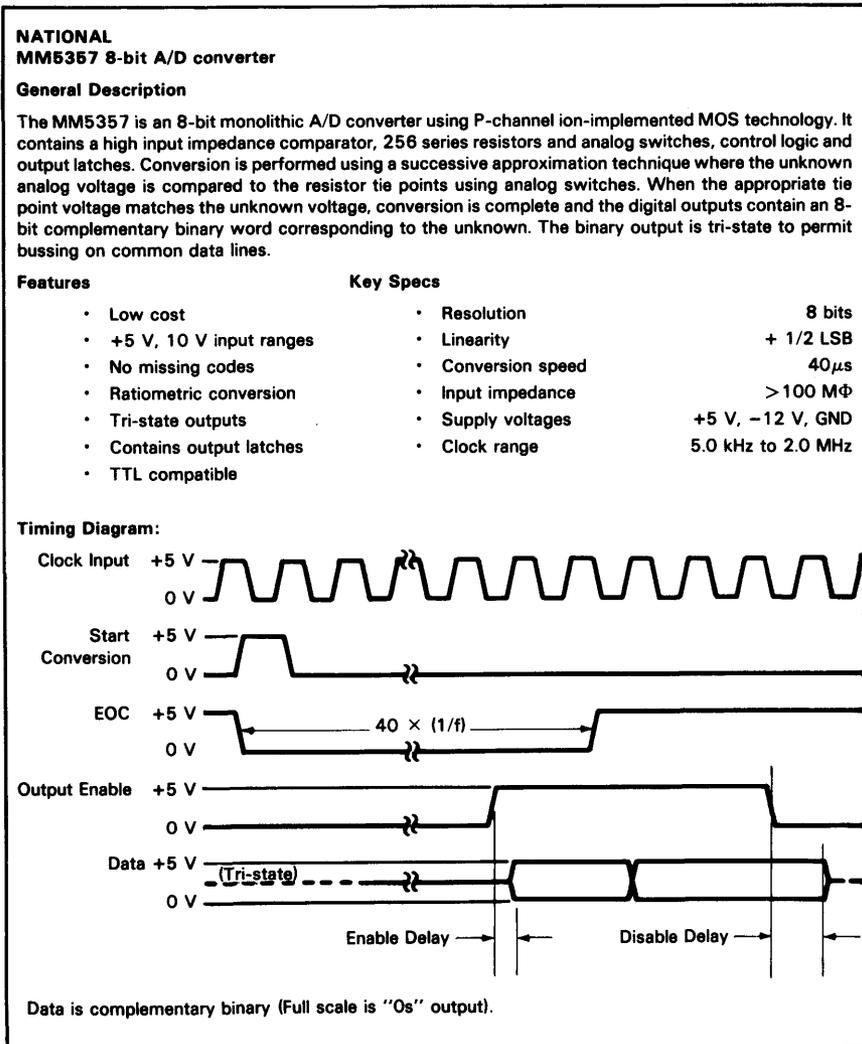


Figure 13-16. General Description and Timing Diagram for the National 5357 A/D Converter

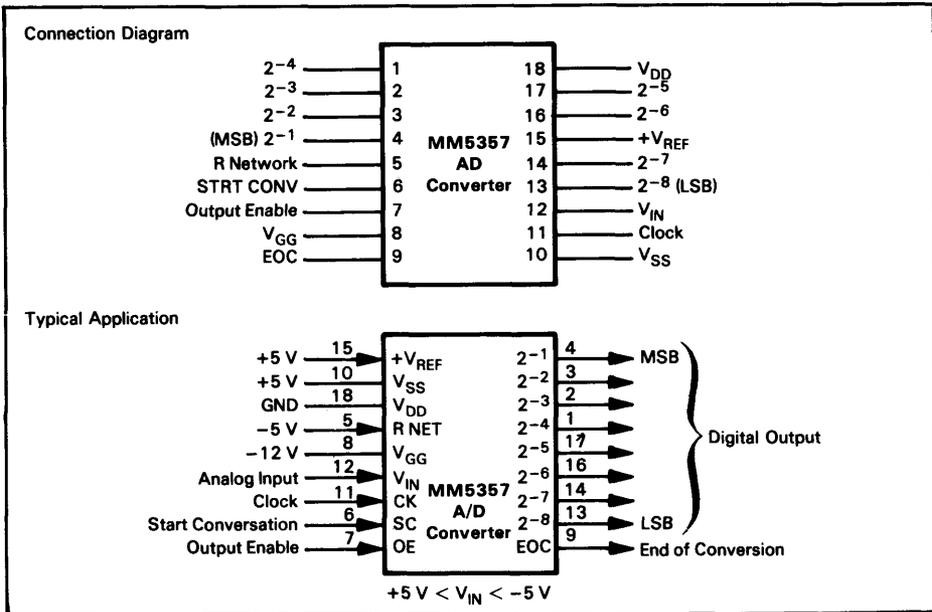


Figure 13-17. Connection Diagram and Typical Application for the National 5357 A/D Converter

Figure 13-16 contains a general description and a timing diagram for the National MM5357 8-bit A/D converter. The device contains output latches and tristate data outputs. A pulse on the Start Conversion (STRT CONV) line starts conversion of the analog input; after about 40 clock cycles (the converter requires a TTL level clock with a minimum pulse width of 400 ns), the result will go to the output latches and the End of Conversion (EOC) output will indicate this by going high. Data is read from the latches by applying a '1' to the OUTPUT ENABLE input. Figure 13-17 shows the connections for the device and some typical applications circuits.

A/D Converter Interface

Figure 13-18 shows the interface between the 6809 microprocessor and the 5357 A/D converter. Control line CA2 is used in the level (manual) output mode to provide an active-high Start Conversion pulse of sufficient length. The End of Conversion signal is tied to control line CA1 so bit 7 of PIA Control Register A is set when EOC goes high. The important transition on the End of Conversion line is the leading edge (low-to-high transition), which indicates the completion of the conversion. Here we are using the 6820 PIA to handle parallel input, serial input, and serial output, since the A/D converter requires a complete handshake. The OUTPUT ENABLE pin on the converter is tied high, since we are not placing the data directly on the microprocessor's tristate data bus. Note (see Fig. 13-16) that the converter's data outputs are complementary binary (an all zeros output is full scale).

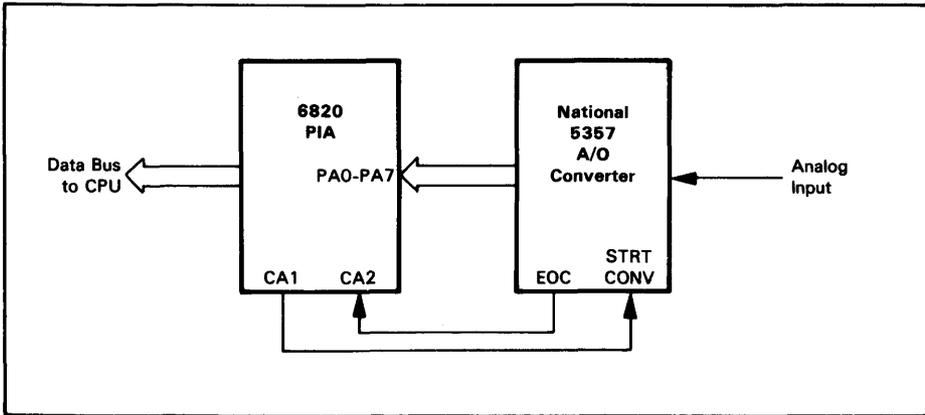


Figure 13-18. Interface for an 8-Bit Analog-to-Digital Converter

Task 13-8. Input from Converter

Purpose: Start the conversion process, wait for End of Conversion (EOC) to go high, then read the data and store it in memory location 0040.

Program 13-8:

```

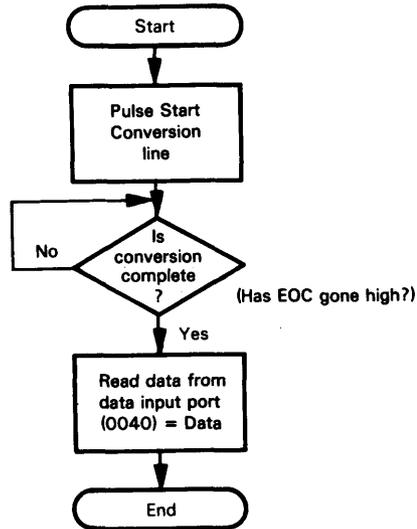
8001  PIACA EQU $8001
8000  PIADDA EQU $8000
8000  PIADA EQU $8000
*
0000  ORG $0000
0000 7F 8001 CLR PIACA ADDRESS DATA DIRECTION REGISTER
0003 7F 8000 CLR PIADDA MAKE ALL DATA LINES INPUTS
0006 86 36 LDA #$00110110 BRING START LOW, TRIGGER ON
0008 B7 8001 STA PIACA EOC GOING HIGH
0008 8A 08 ORA #$00001000 BRING START CONVERSION HIGH
000D B7 8001 STA PIACA
0010 84 F7 ANDA #$11110111 BRING START CONVERSION LOW
0012 B7 8001 STA PIACA
0015 B6 8001 WTEOC LDA PIACA HAS CONVERSION BEEN COMPLETED?
0018 2A FB BPL WTEOC NO, WAIT
001A B6 8000 LDA PIADA YES, FETCH DATA FROM CONVERTER
001D 43 COMA COMPLEMENT DATA TO PRODUCE TRUE
*
001E 97 40 STA $40 SAVE DATA IN MEMORY
0020 3F SWI
    
```

This program would use less time and memory if we used Index Register X to address the PIA. Rewrite the program to do this. We have used explicit addresses in the interest of clarity.

The PIA control register bits are determined as follows:

- Bit 5 = 1 to make CA2 an output.
- Bit 4 = 1 to make CA2 a level (i.e., to operate in the manual mode).
- Bit 3 = 0 to bring Start Conversion low initially.
- Bit 1 = 1 to set bit 7 on a low-to-high transition (leading edge) on the End of Conversion line.

A delay routine of appropriate length (longer than the maximum guaranteed conversion time) could replace the examination of the status bit.

Flowchart:

Here the PIA serves as a parallel data port, a serial status port, and a serial control port. An initial read of the Data Register in the startup routine would clear the status flags originally and eliminate problems that might be caused by stray pulses on the control lines.

13-9. A TELETYPEWRITER (TTY)

We will transfer data to and from a standard 10-character-per-second serial teletypewriter.

Standard TTY Character Format

The common teletypewriter transfers data in an asynchronous serial mode. The procedure is as follows:

1. The line is normally in the mark state (logic '1').
2. A Start bit (space state or logic '0').
3. The character is usually 7-bit ASCII with the least significant bit transmitted first.
4. The most significant bit is a Parity bit, which may be even, odd, or fixed at zero or one.
5. Two stop bits (logic '1's) follow each character to provide a minimum separation between characters.

Figure 13-19 shows the format. Note that each character requires the transmission of eleven bits, of which only seven contain information. Since the data rate is ten characters per second, the bit rate is 10×11 , or 110 Baud. Each bit therefore has a width of 1/110 of a second, or 9.1 milliseconds. This width is an average; the teletypewriter does not maintain it to any high level of accuracy.

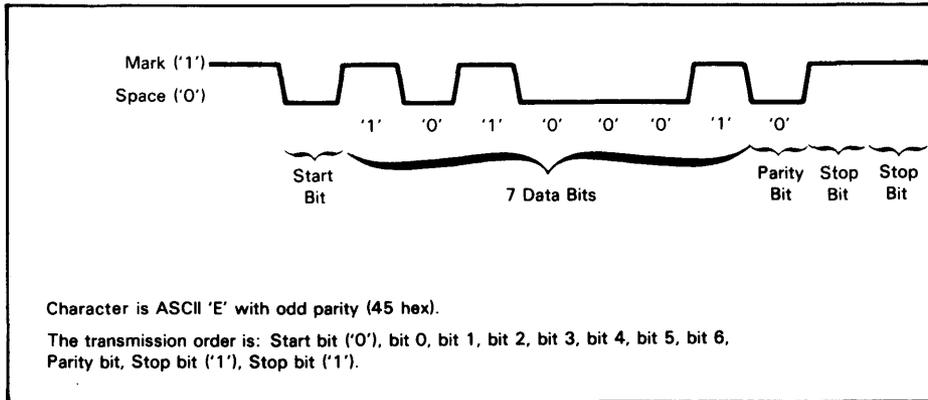


Figure 13-19. Teletypewriter Data Format

TTY Receive Mode

This is the receive procedure, flowcharted in Figure 13-20:

- Step 1. Look for a Start bit (a logic zero) on the data line.
- Step 2. Center the reception by waiting one-half bit time, or 4.55 milliseconds.
- Step 3. Fetch the data bits, waiting one bit time before each one. Assemble the data bits into a word by first shifting the bit to the Carry and then circularly shifting the data with the Carry. Remember that the least significant bit is received first.
- Step 4. Generate the received Parity and check it against the transmitted Parity. If they do not match, indicate a "Parity error."
- Step 5. Fetch the Stop bits (waiting one bit time between inputs). If they are not correct (if both Stop bits are not one), indicate a "framing error."

TTY Transmit Mode

This is the transmit procedure, flowcharted in Figure 13-21:

- Step 1. Transmit a Start bit (i.e., a logic zero).
- Step 2. Transmit the seven data bits, starting with the least significant bit.
- Step 3. Generate and transmit the Parity bit.
- Step 4. Transmit two Stop bits (i.e., logic ones).

The transmission routine must wait one bit time between output operations.

Task 13-9a. Read Data from TTY

Purpose: Fetch data from a teletypewriter using bit 7 of a PIA data port and place the data in memory location 0060. Figure 13-20 describes the procedure.

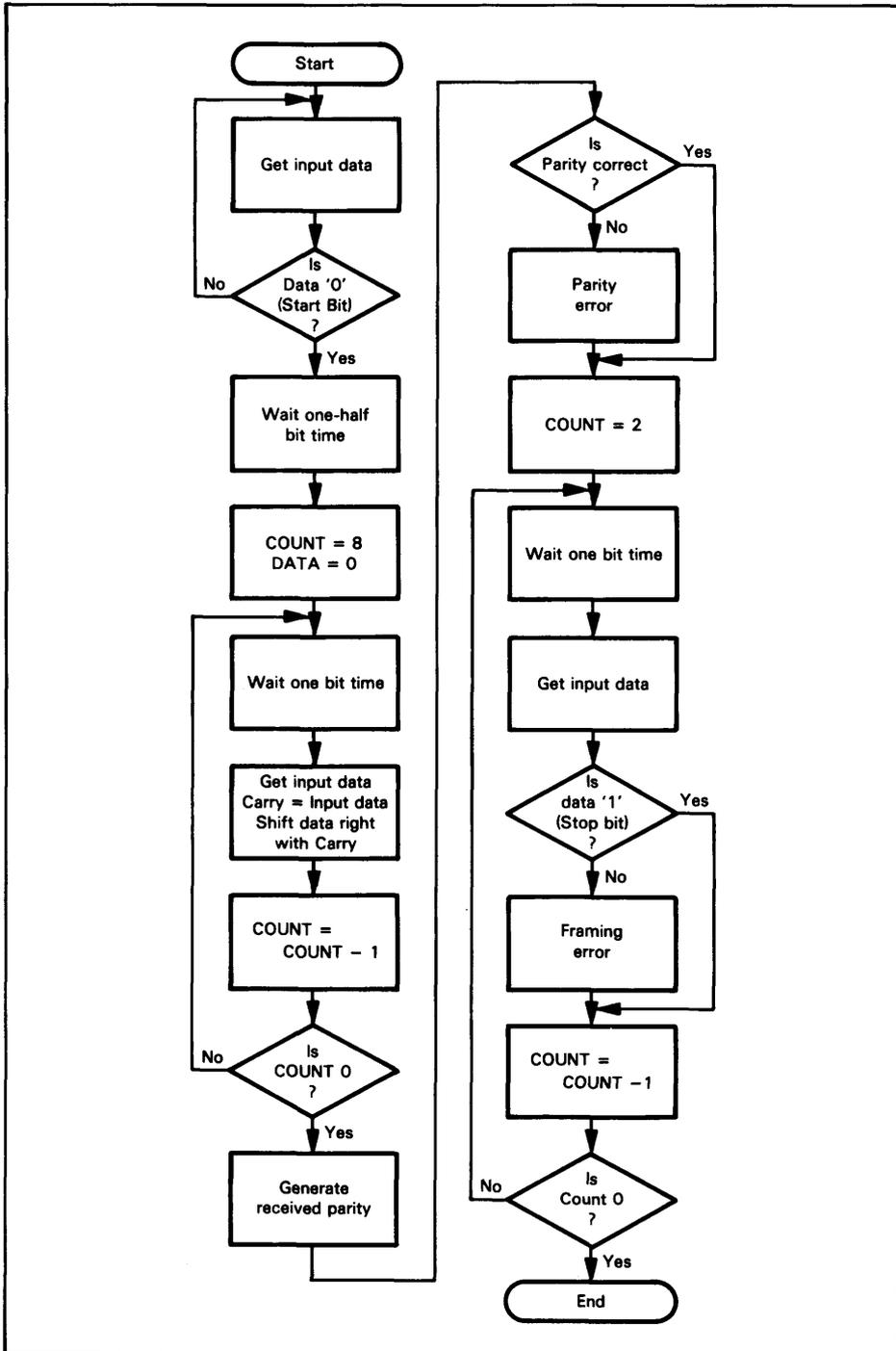


Figure 13-20. Flowchart for Receive Procedure

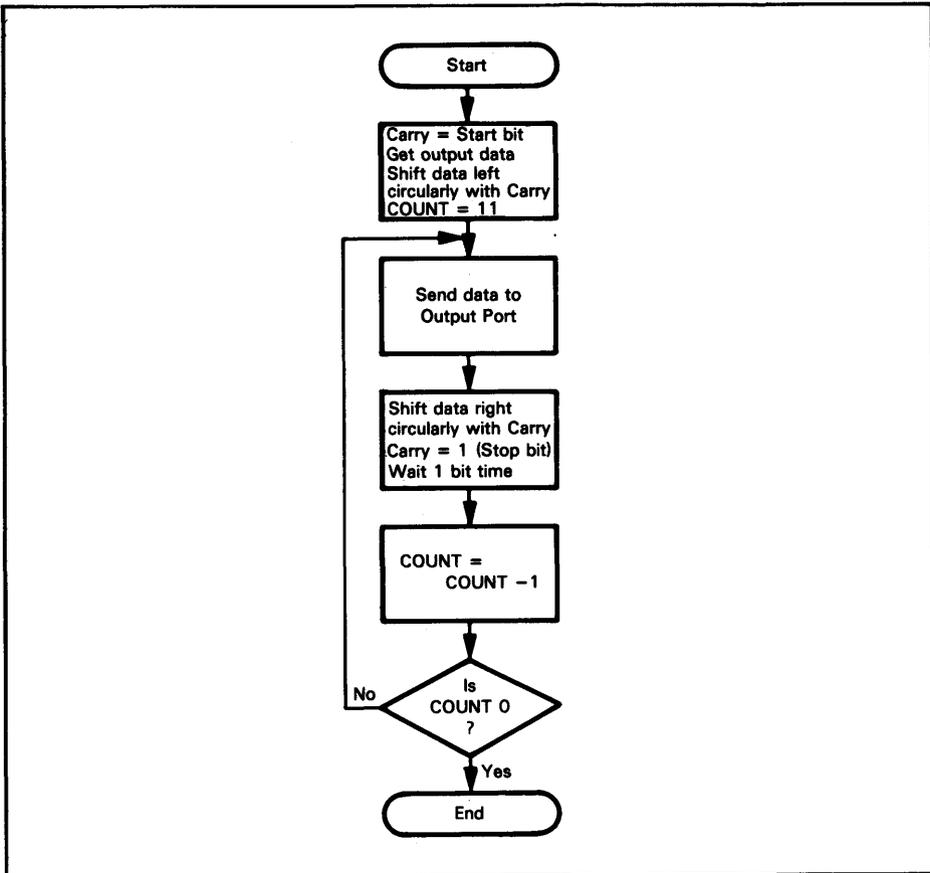


Figure 13-21. Flowchart for Transmit Procedure

Program 13-9a:

Assume that the serial port is bit 7 of the PIA and that no parity or framing check is necessary.

	8001	PIACA	EQU	\$8001		
	8000	PIADDA	EQU	\$8000		
	8000	PIADA	EQU	\$8000		
		*				
0000		ORG	\$0000			
0000	7F	8001	CLR	PIACA	ADDRESS DATA DIRECTION REGISTER	
0003	7F	8000	CLR	PIADDA	MAKE ALL DATA LINES INPUTS	
0006	86	04	LDA	#\$00000100	ADDRESS DATA REGISTER	
0008	B7	8001	STA	PIACA		
000B	B6	8000	WTSTB	LDA	PIADA	IS THERE A START BIT?
000E	2B	FB	BMI	WTSTB	NO, WAIT	
0010	BD	0030	JSR	DLY2	YES, DELAY HALF BIT TIME TO CENTER RECEPTION	
		*				
0013	86	80	LDA	#\$10000000	COUNT WITH '1' BIT IN MSB	
0015	BD	0035	TTYRCV	JSR	DELAY	WAIT 1 BIT TIME
0018	79	8000	ROL	PIADA	GET NEXT DATA BIT	
001B	46		RORA		COMBINE WITH PREVIOUS DATA	
001C	24	F7	BCC	TTYRCV	CONTINUE UNTIL COUNT BIT	
001E	97	60	STA	\$60	TRAVERSES DATA	
0020	3F		SWI			

(Delay program)

```

0030 8E 0236 DLY2 LDX #S0236 COUNT FOR 4.55 MS (CLOCK
0033 20 03 BRA DLY RATE = 1 MHZ)
0035 8E 046C DELAY LDX #S046C COUNT FOR 9.1 MS
0038 30 1F DLY LEAX -1,X
003A 26 FC BNE DLY
003C 39 RTS

```

Remember that bit 0 of the data is received first.

This program assumes that the monitor has initialized the Hardware Stack Pointer. If this is not the case, you will have to initialize the Hardware Stack Pointer with LDS as shown in Chapter 10.

We obtained the constants for the delay routine as described earlier in this chapter, assuming a clock rate of 1 MHz. You may want to check them for yourself. The delay times do not have to be highly accurate because the routine centers the reception, each character is handled separately, the bit rate is low, and the teletypewriter itself is not highly accurate.

How would you extend this program to check parity?

Task 13-9b. Write Data to TTY

Purpose: Transmit data to a teletypewriter using bit 0 of a PIA data port. The data is in memory location 0060.

Program 13-9b:

Assume that parity need not be generated.

```

0035 DELAY EQU $0035
8001 PIACA EQU $8001
8000 PIADA EQU $8000
8000 PIADDA EQU $8000
*
0000 ORG $0000
0000 7F 8001 CLR PIACA ADDRESS DATA DIRECTION REGISTER
0003 86 FF LDA #$FF MAKE ALL DATA LINES OUTPUTS
0005 B7 8000 STA PIADDA
0008 86 04 LDA #$0000100 ADDRESS DATA REGISTER
000A B7 8001 STA PIACA
000D 96 60 LDA $60 GET DATA
000F C6 0B LDB #11 COUNT = 11 BITS IN CHARACTER
0011 7F 8000 CLR PIADA SEND START BIT
0014 9D 35 TBIT JSR DELAY WAIT 1 BIT TIME
0016 1A 01 ORCC #$00000001 SET CARRY TO FORM STOP BIT
0018 46 RORA GET NEXT BIT OF CHARACTER
0019 79 8000 ROL PIADA SEND NEXT BIT TO TTY
001C 5A DECB
001D 26 F5 BNE TBIT
001F 3F SWI

```

The DELAY subroutine is the same as before. Remember that bit 0 of the data must be transmitted first.

In actual applications, you should place a logic '1' on the teletypewriter line as part of the startup routine, since that line should normally be in the mark (1) state.

Each character consists of 11 bits, beginning with a start bit ('0') and ending with two stop bits ('1's). The instruction ORCC #00000001 sets the least significant bit of the Condition Code Register (the Carry flag), thus generating a logic '1' which RORA then shifts into the most significant bit of Accumulator A.

We can generate parity by counting bits as shown in Chapter 6. The program is

as follows:

```

LDA    $60           GET DATA
CLR    CLR           BIT COUNT = ZERO INITIALLY
CHBIT  ASLA          SHIFT A DATA BIT TO CARRY
      ADCB    #0     IF BIT IS 1, ADD 1 TO BIT COUNT
      TSTA          KEEP COUNTING UNTIL DATA BECOMES ZERO
      BNE    CHBIT
      SWI

```

Accumulator B contains the number of '1' bits in the data. The least significant bit of Accumulator B is therefore an even Parity bit.

UART

These procedures are sufficiently common and complex to merit a special LSI device: the UART, or Universal Asynchronous Receiver/Transmitter.²⁰ The UART will perform the reception procedure and provide the data in parallel form and a Data Ready signal. It will also accept data in parallel form, perform the transmission procedure, and provide a Peripheral Ready signal when it can handle more data. UARTs may have many other features, including:

1. Ability to handle various bit lengths (usually 5 to 8), parity options, and numbers of Stop bits (usually 1, 1-1/2, and 2).
2. Indicators for framing errors, parity errors, and "overrun errors" (failure to read a character before another one is received).
3. RS-232²¹ compatibility; i.e., a Request-to-Send (RTS) output signal that indicates the presence of data to communications equipment and a Clear-to-Send (CTS) input signal that indicates, in response to RTS, the readiness of the communications equipment. There may be provisions for other RS-232 signals, such as Received Signal Quality, Data Set Ready, or Data Terminal Ready.
4. Tristate outputs and control compatibility with a microprocessor.
5. Clock options that allow the UART to sample incoming data several times in order to detect false Start bits and other errors.
6. Interrupt facilities and controls.

UARTs act as four parallel ports: an input data port, an output data port, an input status port, and an output control port. The status bits include error indicators as well as Ready flags. The control bits select various options. UARTs are inexpensive (\$5 to \$50, depending on features) and easy to use.

PROBLEMS

13-1. An On-Off Pushbutton

Purpose: Each closure of the pushbutton complements (inverts) all the bits in memory location 0040. The location initially contains zero. The program should continuously examine the pushbutton and complement location 0040 with each

closure. You may wish to complement a display output port instead, so as to make the results easier to see.

Sample Case:

Location 0040 initially contains zero.

The first pushbutton closure changes location 0040 to FF_{16} , the second changes it back to zero, the third back to FF_{16} , etc. Assume that the pushbutton is debounced in hardware. How would you include debouncing in your program?

13-2. Debouncing a Switch in Software

Purpose: Debounce a mechanical switch by waiting until two readings, taken a debounce time apart, give the same result. Assume that the debounce time (in ms) is in memory location 0040 and store the switch position in memory location 0041.

Sample Problem:

(0040) = 03 causes the program to wait 3 ms between readings

13-3. Control for a Rotary Switch

Purpose: Another switch serves as a Load switch for a four-position unencoded rotary switch. The CPU waits for the Load switch to close (be zero), and then reads the position of the rotary switch. This procedure allows the operator to move the rotary switch to its final position before the CPU tries to read it. The program should place the position of the rotary switch into memory location 0040. Debounce the Load switch in software.

Sample Problem:

Place rotary switch in position 2. Close Load switch.

Result: (0040) = 02

13-4. Record Switch Positions on Lights

Purpose: A set of eight switches should have their positions reflected on eight LEDs. That is to say, if the switch is closed (zero), the LED should be on; otherwise, the LED should be off. Assume that the CPU output port is connected to the cathodes of the LEDs.

Sample Problem:

SWITCH 0 CLOSED	Result: LED	0 ON
SWITCH 1 OPEN	LED	1 OFF
SWITCH 2 CLOSED	LED	2 ON
SWITCH 3 OPEN	LED	3 OFF
SWITCH 4 OPEN	LED	4 OFF
SWITCH 5 CLOSED	LED	5 ON
SWITCH 6 CLOSED	LED	6 ON
SWITCH 7 OPEN	LED	7 OFF

13-54 6809 Assembly Language Programming

How would you change the program so that a switch attached to bit 7 of Port A of PIA #2 determines whether the displays are active (i.e., if the control switch is closed, the displays attached to Port B reflect the switches attached to Port A; if the control switch is open, the displays are always off)? A control switch is useful when the displays may distract the operator, as in an airplane.

How would you change the program to make the control switch an on-off pushbutton; that is, each closure inverts the previous state of the displays? Assume that the displays start in the active state and that the program examines and debounces the pushbutton before sending data to the displays.

13-5. Count on a Seven-Segment Display

Purpose: The program should count from 0 to 9 continuously on a seven-segment display, starting with zero.

Hint: Try different timing lengths for the displays and see what happens. When does the count become visible? What happens if the display is blanked part of the time?

13-6. Separating Closures from an Unencoded Keyboard

Purpose: The program should read entries from an unencoded 3×3 keyboard and save them in an array. The number of entries required is in memory location 0040 and the array starts in memory location 0041.

Separate one closure from the next by waiting for the current closure to end. Remember to debounce the keyboard (this can be simply a 1 ms wait).

Sample Problem:

```
(0040) = 04
Entries are 7, 2, 2, 4
Result: (0041) = 07
        (0042) = 02
        (0043) = 02
        (0044) = 04
```

13-7. Read a Sentence from an Encoded Keyboard

Purpose: The program should read entries from an ASCII keyboard (7 bits with a zero Parity bit) and place them in an array until it receives an ASCII period (hex 2E). The array starts in memory location 0040. Each entry is marked by a strobe as in Example 13-6.

Sample Problem:

```
Entries are H, E, L, L, O.
Result: (0040) = 48 'H'
        (0041) = 45 'E'
        (0042) = 4C 'L'
        (0043) = 4C 'L'
        (0044) = 4F 'O'
        (0045) = 2E '.'
```

13-8. A Variable Amplitude Square Wave Generator

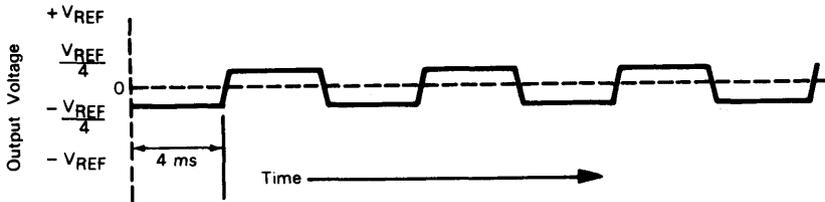
Purpose: The program should generate a square wave, as shown in the next figure, using a D/A converter. Memory location 0040 contains the scaled amplitude of the wave, memory location 0041 the length of a half cycle in milliseconds, and memory location 0042 the number of cycles.

Assume that a digital output of 80_{16} to the converter results in an analog output of zero volts. In general, a digital output of D results in an analog output of $(D-80)/80 \times -V_{REF}$ volts.

Sample Problem:

(0040) = $A0_{16}$
 (0041) = 04
 (0042) = 03

Result:



The base voltage is $80_{16} = 0$ volts.
 Full scale is $100_{16} = -V_{REF}$ volts.
 So $A0_{16} = (A0-80)/80 \times (-V_{REF}) = -V_{REF}/4$

The program produces 3 pulses of amplitude $V_{REF}/4$ with a half cycle length of 4 ms.

13-9. Averaging Analog Readings

Purpose: The program should take four readings from an A/D converter 10 milliseconds apart and place the average in memory location 0040. Assume that the A/D conversion time can be ignored.

Sample Problem:

Hexadecimal readings are 86, 89, 81, 84

Result: (0040) = 85_{16}

13-10. A 30 Character-per-Second Terminal

Purpose: Modify the transmit and receive routines of Example 13-9 to handle a 30 cps terminal that transfers ASCII data with one stop bit and even parity. How could you write the routines to handle either terminal depending on a flag in memory location 0061; e.g., (0061) = 0 for the 30 cps terminal, and (0061) = 1 for the 10 cps terminal?

REFERENCES

1. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, pp. 9-45 through 9-54.
2. J. Gilmore and R. Huntington. "Designing with the 6820 Peripheral Interface Adapter," *Electronics*, December 23, 1976, pp. 85-86.
3. *The TTL Data Book for Design Engineers*, Texas Instruments Inc., P.O. Box 5012, Dallas, Tex. 75222, pp. 7-151 through 7-156.
4. E. Dilatush. "Special Report: Numeric and Alphanumeric Displays," *EDN*, January 5, 1978, pp. 26-35.
5. *The TTL Data Book for Design Engineers*, Texas Instruments Inc., P.O. Box 5012, Dallas, Tex. 75222, pp. 7-22 through 7-34.
6. A. Pshaenich. "Interface Considerations for Numeric Display Systems," Motorola Semiconductor Products Inc., Application Note AN-741, Phoenix, Ariz. 1975.
7. Motorola Semiconductor Products Inc., *Microprocessor Applications Manual*, McGraw-Hill, New York, 1975, pp. 5-6 through 5-11.
8. Motorola Semiconductor Products Inc., *Microprocessor Applications Manual*, McGraw-Hill, New York, 1975, pp. 5-1 through 5-5.
9. See Reference 2.
10. J. Kane et al. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices*, Osborne/McGraw-Hill, Berkeley, Calif. 1979, Section E1.
11. E. R. Hnatek. *A User's Handbook of D/A and A/D Converters*, Wiley, New York, 1976.
12. P. H. Garrett. *Analog Systems for Microprocessors and Minicomputers*, Reston Publishing Co. (Prentice-Hall), Reston, VA, 1978.
13. B. Amazeen. "Monolithic D-A Converter Operates on Single Supply," *Electronics*, February 28, 1980, pp. 125-31.
14. See Reference 11.
15. See Reference 12.
16. J. Kane et al. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices*, Osborne/McGraw-Hill, Berkeley, Calif. 1979, Section E2.
17. D. Aldridge. "Analog to Digital Conversion Techniques with the M6800 Microprocessor System," Motorola Semiconductor Products Inc. Application Note AN-757, Phoenix, Ariz. 1975.
18. P. Bradshaw. "Two-Chip A/D Converter," *Electronic Design*, March 29, 1979, pp. 128-36.
19. M. Tuthill and D. P. Burton. "Low-Cost A/D Converter Links Easily with Microprocessors," *Electronics*, August 30, 1979, pp. 149-55.
20. For a discussion of UARTs, see P. Rony et al. "The Bugbook IIa," E and L Instruments Inc., 61 First Street, Derby, Conn. 06418 or D. G. Larsen et al. "INWAS: Interfacing with Asynchronous Serial Mode," *IEEE Transactions on Industrial Electronics and Control Instrumentation*, February 1977, pp. 2-12.

21. "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange," EIA RS-232C, Electronic Industries Association, 2001 I Street N.W., Washington, D.C. 20006, August 1969.
 - J. Kane et al. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices*, Osborne/McGraw-Hill, Berkeley, Calif. pp. J5-9 through J5-14.
 - G. Pickles. "Who's Afraid of RS-232?," *Kilobaud*, May 1977, pp. 50-54.
 - C. A. Ogdin. "Microcomputer Buses — Part II," *Mini-Micro Systems*, July 1978, pp. 76-80.



14

Using the 6850 ACIA

The **6850 ACIA**, or Asynchronous Communications Interface Adapter, (see Figure 14-1) is a UART specifically designed for use in 6800, 6809, and 6502-based microcomputers. It occupies two memory addresses and contains two read-only registers (received data and status) and two write-only registers (transmitted data and control). Tables 14-1 and 14-2 describe the contents of these registers.

ADDRESSING THE 6850 ACIA

The internal registers of the ACIA are addressed by means of the RS (register select) and R/W (read/write) lines (see Table 14-3). If, as is usual, RS is tied to A0, the least significant bit of the 6809's address bus, then the address of the Data Registers is one larger than the address of the Control and Status Registers. The use of R/W for addressing means that read and write cycles access different registers, so the program can neither read the transmitted data or control registers nor write into the received data or status registers. If the program must recall what it stored in the write-only registers, it must retain a copy in RAM. We will refer to the addresses as ACIADR (the receive data register when reading, the transmitted data register when writing), ACIASR (the read-only status register), and ACIACR (the write-only control register). ACIASR and ACIACR are the same physical address.

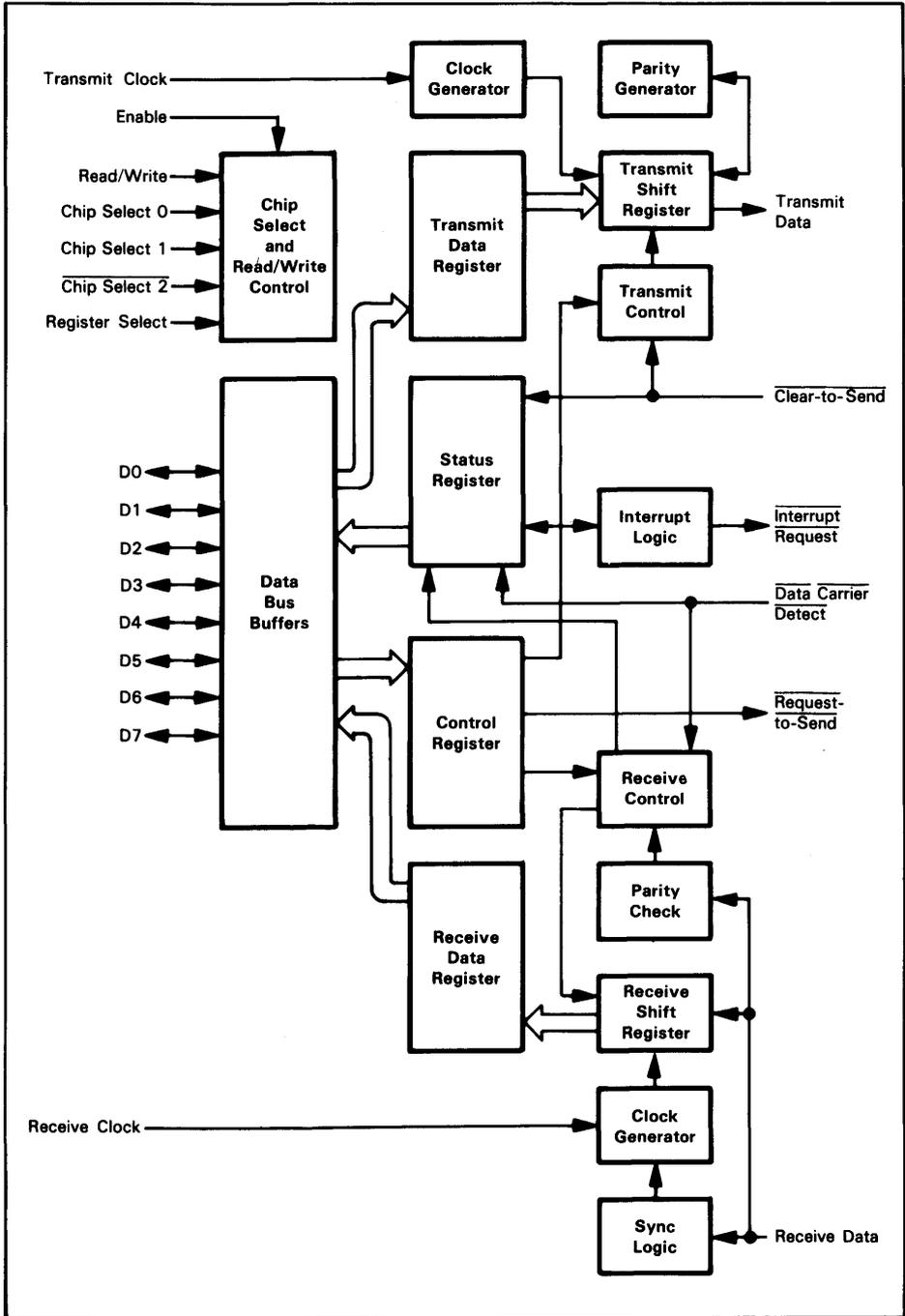


Figure 14-1. Block Diagram of the 6850 ACIA

Table 14-1. Definition of 6850 ACIA Register Contents

Data Bus Line Number	Buffer Address			
	RS· $\overline{R/W}$ Transmit Data Register	RS·R/W Receive Data Register	\overline{RS} · $\overline{R/W}$ Control Register	\overline{RS} ·R/W Status Register
	(Write Only)	(Read Only)	(Write Only)	(Read Only)
0	Data Bit 0*	Data Bit 0	Counter Divide Select 1 (CR0)	Receive Data Register Full (RDRF)
1	Data Bit 1	Data Bit 1	Counter Divide Select 2 (CR1)	Transmit Data Register Empty (TDRE)
2	Data Bit 2	Data Bit 2	Word Select 1 (CR2)	Data Carrier Detect (\overline{DCD})
3	Data Bit 3	Data Bit 3	Word Select 2 (CR3)	Clear-to-Send (CTS)
4	Data Bit 4	Data Bit 4	Word Select 3 (CR4)	Framing Error (FE)
5	Data Bit 5	Data Bit 5	Transmit Control 1 (CR5)	Receiver Overrun (OVRN)
6	Data Bit 6	Data Bit 6	Transmit Control 2 (CR6)	Parity Error (PE)
7	Data Bit 7***	Data Bit 7**	Receive Interrupt Enable (CR7)	Interrupt Request (IRQ)
<p>* Leading bit = LSB = Bit 0 ** Data bit will be zero in 7-bit plus parity modes *** Data bit is "don't care" in 7-bit plus parity modes</p>				

SPECIAL FEATURES

Note the following special features of the 6850 ACIA:

1. **Read and write cycles address physically distinct registers.** Therefore, you cannot use the ACIA registers as addresses for instructions like Increment, Decrement, or Shift, which involve both read and write cycles.
2. **The ACIA Control register cannot be read by the CPU.** You will have to save a copy of the Control register in memory if the program needs its value.
3. **The ACIA has no Reset input. It can be reset only by placing ones in Control register bits 0 and 1.** This procedure (called "Master Reset") is necessary before the ACIA is used, in order to avoid having a random starting character.
4. **The RS-232 signals are all active-low.** Request-to-Send (RTS), in particular, should be brought high to make it inactive if it is not in use.

Table 14-2. Meaning of the 6850 ACIA Control Register Bits

CR6	CR5	Function	
0	0	RTS = low. Transmitting Interrupt Disabled RTS = low. Transmitting Interrupt Enabled RTS = high. Transmitting Interrupt Disabled RTS = low. Transmits a Break level on the Transmit Data Output. Transmitting Interrupt Disabled	
0	1		
1	0		
1	1		
CR4	CR3	CR2	Function
0	0	0	7 Bits + Even Parity + 2 Stop Bits
0	0	1	7 Bits + Odd Parity + 2 Stop Bits
0	1	0	7 Bits + Even Parity + 1 Stop Bit
0	1	1	7 Bits + Odd Parity + 1 Stop Bit
1	0	0	8 Bits + 2 Stop Bits
1	0	1	8 Bits + 1 Stop Bit
1	1	0	8 Bits + Even Parity + 1 Stop Bit
1	1	1	8 Bits + Odd Parity + 1 Stop Bit
CR1	CR0	Function	
0	0	+1	
0	1	+16	
1	0	+64	
1	1	Master Reset	

Table 14-3. Internal Addressing for the 6850 ACIA

RS (Register Select)	R/W (Read/Write) 1 = Read, 0 = Write	Register Addressed	Indexed Offset from ACIA Control Register
0	0	Control Register (write-only)	0
0	1	Status Register (read-only)	0
1	0	Transmit Data Register (write-only)	1
1	1	Receive Data Register (read-only)	1

- The ACIA requires an external clock.** Typically, 1760 Hz is supplied and the ÷ 16 mode (Control register bit 1 = 0, bit 0 = 1) is used. The ACIA will use the clock to center the reception and to avoid false Start bits caused by noise on the lines.
- The Data Ready (receive data register full, or RDRF) flag is bit 0 of the Status register. The Peripheral Ready (transmit data register empty, or TDRE) flag is bit 1 of the Status register.**

PROGRAM EXAMPLES

14-1. RECEIVE DATA FROM TTY

Purpose: Receive data from a teletypewriter using a 6850 ACIA and store the data in memory location 0060.

Program 14-1:

```

      8010  ACIACR EQU  $8010
      8010  ACIASR EQU  $8010
      8011  ACIADR EQU  $8011

0000                                ORG  $0000
0000 86 03                          LDA  #$00000011  MASTER RESET ACIA
0002 B7 8010                        STA  ACIACR
0005 86 45                          LDA  #$01000101  ACIA OPERATING MODE -- TTY
0007 B7 8010                        STA  ACIACR          WITH ODD PARITY
000A B6 8010  WAITD LDA  ACIASR  GET STATUS OF ACIA
000D 44                                LSR  A          HAS DATA BEEN RECEIVED?
000E 24 FA                          BCC  WAITD      NO, WAIT
0010 B6 8011                        LDA  ACIADR      YES, READ THE DATA
0013 97 60                          STA  $60       AND SAVE IT IN MEMORY
0015 3F                                SWI

```

The program must reset the ACIA originally by placing ones in Control register bits 0 and 1. The ACIA does have an internal power-on reset which holds the ACIA in the reset state until Master Reset is applied.

The program determines the operating mode of the ACIA by setting the bits in the Control Register as follows:

- Bit 7 = 0 to disable the receiver interrupt
- Bit 6 = 1 to make Request-to-Send (RTS) high (inactive)
- Bit 5 = 0 to disable the transmitter interrupt
- Bit 4 = 0 for 7-bit words
- Bit 3 = 0, Bit 2 = 1 for odd parity with 2 Stop bits
- Bit 1 = 0, Bit 0 = 1 for ÷ 16 clock (1760 Hz must be supplied)

The received data status flag is bit 0 of the ACIA Status Register. What would happen if we tried to replace

```

      LDA  ACIASR
      LSR  A

```

with the single instruction

```

      LSR  ACIASR

```

Remember that the Status and Control registers share an address but are physically distinct.

Try adding an error-checking routine to the program. Set

```

(0061) = 0 if no errors occurred
        = 1 if a parity error occurred
          (Status register bit 6 = 1)
        = 2 if an overrun error occurred
          (Status register bit 5 = 1)
        = 3 if a framing error occurred
          (Status register bit 4 = 1)

```

Assume that the priority of the errors is from left to right in the ACIA status register (i.e., parity errors have priority over overrun errors which, in turn, have priority over framing errors if more than one error has occurred).

14-2. SEND DATA TO TTY

Purpose: Send data from memory location 0060 to a teletypewriter using a 6850 ACIA.

Program 14-2:

```

      8010   ACIACR EQU   $8010
      8010   ACIASR EQU   $8010
      8011   ACIADR EQU   $8011
          *
0000                   ORG   $0000
0000 86   03           LDA   #$00000011   MASTER RESET ACIA
0002 B7   8010        STA   ACIACR
0005 86   45           LDA   #$01000101   ACIA OPERATING MODE -- TTY
0007 B7   8010        STA   ACIACR           WITH ODD PARITY
000A 86   02           LDA   #$00000010   IS ACIA READY TO TRANSMIT?
000C B5   8010   WAITR BITA ACIASR           NO, WAIT
000F 27   FB           BEQ   WAITR           YES, GET DATA
0011 96   60           LDA   $60           AND TRANSMIT IT
0013 B7   8011        STA   ACIADR
0016 3F                   SWI

```

The transmitter status flag is bit 1 of the ACIA Status register. The Bit Test instruction is convenient here, since it performs a logical AND without changing the contents of the Accumulator. How could you modify the receive program to use the Bit Test instruction?

REFERENCES

- A. Osborne et al. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, pp. 9-55 through 9-61.
- K. Fronheiser. "Device Operation and System Implementation of the Asynchronous Communications Interface Adapter," Motorola Semiconductor Products Inc. Application Note AN-754, Phoenix, AZ, 1975.
- J. Volp. "Software Switches Baud Rate," *EDN*, November 5, 1979, p. 83.

15

Interrupts

Interrupts are inputs that the CPU examines as part of each instruction cycle. These inputs allow the CPU to react to asynchronous events more efficiently than by polling devices. The use of interrupts generally involves more hardware than does ordinary (programmed) I/O, but interrupts provide a faster and more direct response.¹

Why use interrupts? Interrupts allow events such as alarms, power failure, the passage of a certain amount of time, and peripherals having data or being ready to accept data to get the immediate attention of the CPU. The program does not have to examine (poll) every potential source, nor need the programmer worry about the system missing events.

An interrupt system is like the bell on a telephone — it rings when a call comes in so that you don't have to pick up the receiver occasionally to see if someone is on the line. The CPU can go about its normal business (and get a lot more done). When something happens, the interrupt alerts the CPU and forces it to service the input before resuming normal operations. Of course, this simple description becomes more complicated (just like a telephone switchboard) when there are many interrupts of varying importance and tasks that cannot be interrupted.

CHARACTERISTICS OF INTERRUPT SYSTEMS

The implementation of interrupt systems varies greatly. Among the questions that characterize a particular system are:

1. How many interrupt inputs are there?
2. How does the CPU respond to an interrupt?
3. How does the CPU determine the source of an interrupt if the number of sources exceeds the number of inputs?
4. Can the CPU differentiate between important and unimportant interrupts?
5. How and when is the interrupt system enabled and disabled?

There are many different answers to these questions. The aim of all the implementations, however, is to have the CPU respond rapidly to interrupts and resume normal activity afterwards.

The number of interrupt inputs on the CPU chip determines the number of different responses that the CPU can produce without any additional hardware or software. Each input can produce a different internal response. Unfortunately, most microprocessors have a very small number (one or two, typically) of separate interrupt inputs.

The ultimate response of the CPU to an interrupt must be to transfer control to the correct interrupt service routine and to save the current value of the Program Counter. The CPU must therefore execute a Jump-to-Subroutine or Call instruction with the beginning of the interrupt service routine as its address. This action will save the return address in the Stack and transfer control to the interrupt service routine. The amount of external hardware required to produce this response varies greatly. Some CPUs internally generate the instruction and the address; others require external hardware to form them. The CPU can only generate a different instruction or address for each separate input.

Polling and Vectoring

If the number of interrupting devices exceeds the number of inputs, the CPU will need extra hardware or software to identify the source of the interrupt. In the simplest case, the software can be a polling routine which checks the status of the device that may be interrupting. The only advantage of such a system over normal polling is that the CPU knows that at least one device is active. **The alternative solution is for additional hardware to provide a unique data input (or "vector") for each source.** The two alternatives can be mixed; the vectors can identify groups of inputs from which the CPU can identify a particular one by polling.

Priority

An interrupt system that can differentiate between important and unimportant interrupts is called a "priority interrupt system." Internal hardware can provide as many priority levels as there are inputs. External hardware can provide additional levels through the use of a Priority register and comparator. The external hardware does not allow the interrupt to reach the CPU unless its priority is higher than the contents of the Priority register. A priority interrupt system may need a special way to handle low-priority interrupts that may be ignored for long periods of time.

Enabling and Disabling

Most interrupt systems can be enabled or disabled. In fact, most CPUs automatically disable interrupts when a RESET is performed (so the startup routine can initialize the interrupt system) and on accepting an interrupt (so that the interrupt will not interrupt its own service routine). The programmer may wish to disable interrupts while preparing or processing data, performing a timing loop, or executing a multi-byte operation.

An interrupt that cannot be disabled (sometimes called a "nonmaskable interrupt") may be useful to warn of power failure, an event that obviously must take precedence over all other activities.

Disadvantages of Interrupts

The advantages of interrupts are obvious, but there are also disadvantages. These include:

1. Interrupt systems may require a large amount of extra hardware.
2. Interrupts still require data transfers under program control through the CPU. There is no speed advantage as there is with DMA.
3. Interrupts are random inputs, which make debugging and testing difficult. Errors may occur sporadically, and therefore may be very hard to locate and correct.²
4. Interrupts may involve a large amount of overhead if many registers must be saved and the source must be determined by polling.

6809 INTERRUPT SYSTEM

The 6809 microprocessor's internal response to an interrupt is moderately complex. The interrupt system consists of:

1. Two active-low maskable interrupts ($\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$) and an active-low nonmaskable interrupt (NMI).
2. Separate interrupt disable (or mask) bits for the two maskable interrupts ($\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$). If an interrupt mask bit is 1, the corresponding interrupt is disabled. The $\overline{\text{IRQ}}$ mask bit (or I flag) is bit 4 of the Condition Code Register; the $\overline{\text{FIRQ}}$ mask bit (or F flag) is bit 6 of the Condition Code Register. The E (entire) flag (bit 7 of the Condition Code Register) distinguishes $\overline{\text{FIRQ}}$ interrupts from other interrupts as we will describe later. As might be expected, Reset sets both the I and F bits, thus starting the processor with both interrupts disabled. This allows the programmer to initialize the system before allowing interrupts.

6809 INTERRUPT RESPONSE

The 6809 checks the current status of the interrupt system at the end of each instruction. If an interrupt input is active and enabled, the response is as follows:

1. The CPU disables the maskable interrupt ($\overline{\text{IRQ}}$); that is, it sets bit 4 (the I flag) of the Condition Code Register. If the active input is $\overline{\text{FIRQ}}$ or NMI, the CPU also disables the fast interrupt ($\overline{\text{FIRQ}}$); that is, it sets bit 6 (the F flag) of the Condition Code Register.
2. If the CPU is not executing CWAI or SYNC (we will discuss those instructions later), it clears the E flag in response to $\overline{\text{FIRQ}}$ and sets it otherwise. The E flag is bit 7 of the Condition Code Register.
3. The CPU saves either the Program Counter and the Condition Code Register (input is $\overline{\text{FIRQ}}$) or all the user registers (any other input or instruction) in the Hardware Stack. Figure 15-1 shows the order in which

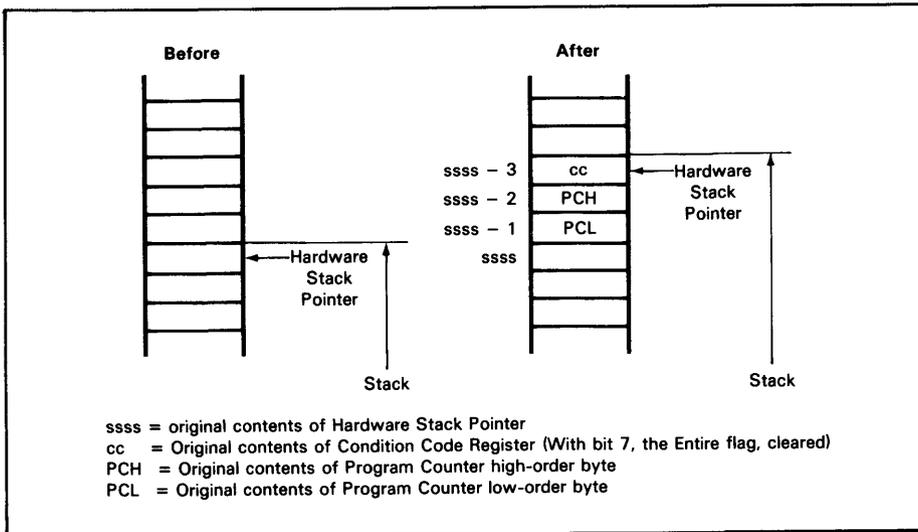


Figure 15-1. Saving the Limited Processor State in the Hardware Stack (Response to $\overline{\text{FIRQ}}$)

the limited state is saved after the recognition of $\overline{\text{FIRQ}}$ and Figure 15-2 shows the order in which the entire state is saved after other inputs or instructions. The E (Entire) flag distinguishes the two alternatives (E is 1 if the *entire* state has been saved and 0 if only the limited subset has been saved).

4. The CPU fetches an address from a specified pair of memory locations and loads that address into the Program Counter. Table 15-1 lists the locations assigned to the various inputs and to the SWI instructions.

Table 15-1. Memory Map for Interrupt Vectors

Memory Map for Vector Location		Interrupt Vector Description
MSBs	LSBs	
FFFE	FFFF	RESET
FFFC	FFFD	NMI
FFFA	FFFB	SWI
FFF8	FFF9	IRQ
FFF6	FFF7	$\overline{\text{FIRQ}}$
FFF4	FFF5	SWI2
FFF2	FFF3	SWI3
FFFO	FFF1	Reserved

The addresses are stored in the usual 6809 manner with the most significant bits at the lower address.

SPECIAL FEATURES

Note the following features of the 6809 interrupt system:

1. The 6809 automatically saves the entire or limited state of the processor in

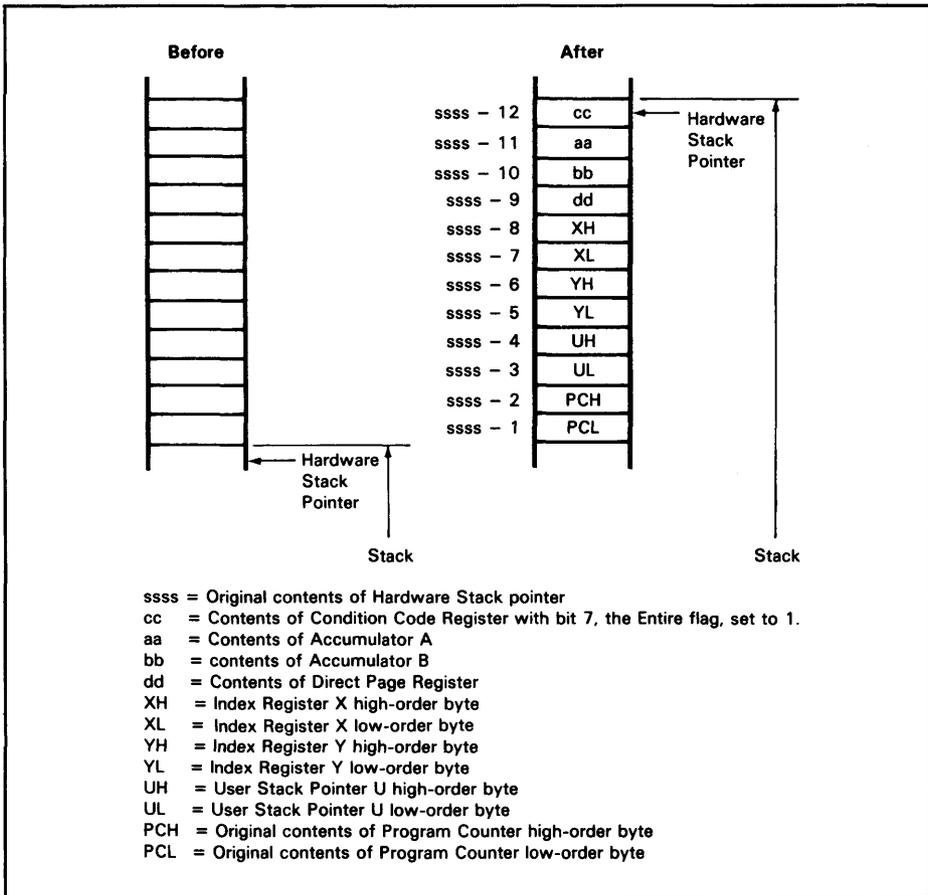


Figure 15-2. Saving the Entire Processor State in the Hardware Stack

the Hardware Stack. The Program Counter is always saved so the interrupted program can be resumed. The Condition Code Register (including the Interrupt Mask and Fast Interrupt Mask flags) is always saved as well.

2. **The Fast Interrupt Request not only provides a second maskable interrupt input, but it also provides a faster response** since only the Program Counter and the Condition Code Register are saved. Saving only the limited state reduces the response time by 9 clock cycles, since 1 clock cycle is needed to transfer each byte to the Stack.
3. **The 6809 provides external hardware signals (using the BUS AVAILABLE and BUS STATE lines) to indicate that it has accepted an interrupt.** These lines can be used to activate external hardware.
4. **The 6809 has no special internal provisions for determining the source of an interrupt when there are several sources tied to the same input.**

Interrupt-Related Instructions

The following special instructions can be used to manipulate the 6809 interrupt system:

1. **ANDCC #%11101111 (or CLI)** clears bit 4 of the Condition Code Register and thus enables the regular interrupt. **ANDCC # %10111111 (or CLF)** similarly clears bit 6 of the Condition Code Register and thus enables the fast interrupt. Of course, **ANDCC #%10101111 (or CLIF)** enables both interrupts at once.
2. **ORCC #%00010000 (or SEI)** sets bit 4 of the Condition Code Register and thus disables the regular interrupt. **ORCC #%01000000 (or SEF)** similarly sets bit 6 of the Condition Code Register and thus disables the fast interrupt. **ORCC #%01010000 (or SEIF)** disables both interrupts at once.
3. **SWI (Software Interrupt)** sets the Entire flag, saves all the user registers in the Hardware Stack, and disables the regular and fast interrupts. It then places the contents of addresses FFFA and FFFB in the Program Counter. **SWI2 (Software Interrupt 2)** and **SWI3 (Software Interrupt 3)** are similar, except that they do not affect the interrupt masks and they use different vector addresses (FFF4 and FFF5 for SWI2, FFF2 and FFF3 for SWI3).
4. **RTI (Return from Interrupt)** restores the registers from the Hardware Stack at the end of an interrupt service routine. If the recovered E flag is cleared, RTI restores only the Condition Code Register and the Program Counter. Thus RTI is similar to RTS, but RTI restores other registers as well as the Program Counter.
5. **CWAI (Clear and Wait for Interrupt)** logically ANDs a byte of immediate data with the Condition Code Register (usually enabling the regular or fast interrupts), saves all the user registers in the Hardware Stack, and waits for an interrupt to occur. The response to the interrupt is rapid (9 clock cycles), since the registers have already been saved. Note that, if a CWAI instruction has been executed and a fast interrupt occurs, the CPU will enter the fast interrupt service routine with all its registers saved (and with the E flag in the Stack set).
6. **SYNC (Synchronize to External Event)** causes the processor to stop executing instructions. The CPU simply waits for an interrupt. If the interrupt is masked or lasts less than 3 clock cycles, the CPU continues to the next instruction without stacking registers or performing an interrupt service routine. Otherwise, the CPU performs its normal interrupt response. SYNC allows an extremely fast response to a single (presumably high-priority) interrupt, since no stacking or vectoring is performed. Obviously, the use of SYNC is a one-time only approach, since the CPU does not save its previous state or identify the source. Figure 15-3 illustrates the use of the SYNC instruction in this manner.

The **SWI (Software Interrupt)** instructions produce almost exactly the same response as an interrupt signal (hence the name). The only difference is the locations from which the CPU obtains the new value of the Program Counter. SWI instructions are useful for debugging (see Chapter 19) and for returning control to a monitor or operating system while simultaneously saving the current state in the stack. **SWI**

instructions are also referred to as *traps*, since they can be used to trap the microcomputer to special routines in the event of hardware errors or other unusual events.³ SWI is commonly used in packaged monitors and operating systems to transfer control from user to system; SWI2 is supposed to be available to the end user, and hence should not be used in packaged systems software.

Fast Interrupt

The fast interrupt is a high-priority maskable interrupt. In response to it, the CPU clears the E flag and saves the Program Counter and Condition Code Register in the Hardware Stack (assuming that it is not executing a CWAI instruction). It then obtains the new value for the Program Counter from memory addresses FFF6 and FFF7. The differences between regular and fast interrupts are thus minimal from the programmer's point of view and we will not refer to fast interrupts again.

Nonmaskable Interrupt

The nonmaskable interrupt is an edge-sensitive input. The processor therefore only reacts to the edge of a pulse on this line, and the pulse will not interrupt its own ser-

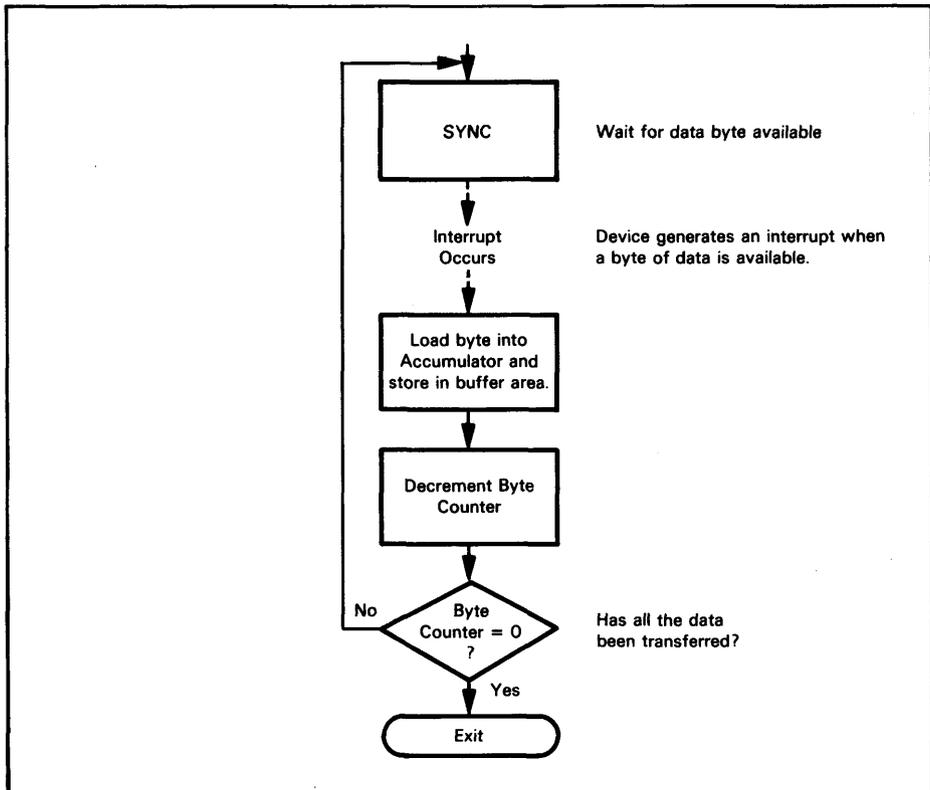


Figure 15-3. Using the SYNC Instruction in Interrupt-Driven Input/Output

vice routine. **Nonmaskable interrupts are useful for applications that must respond to loss of power** (usually by saving data in a low-power memory or switching to a backup battery).⁴ Typical applications are communications equipment that must retain codes and partially received messages, and test equipment that must keep track of partially completed tests. We will not discuss the nonmaskable interrupt any further. Henceforth, we will assume that all interrupt inputs are tied to \overline{IRQ} .

6820 PIA INTERRUPTS

Most 6809 interrupt systems involve 6820 PIAs. Each port of the 6820 PIA has the following features for use with interrupts:

1. An active-low interrupt output.
2. Interrupt enable bits (bit 0 of the Control Register for control line 1, bit 3 for control line 2 if it is an input).
3. Interrupt status bits (bit 7 of the Control Register for control line 1, bit 6 for control line 2).

Bits 1 (control line 1) and 4 (control line 2) determine whether a rising edge (low-to-high transition) or a falling edge (high-to-low transition) on the control line causes an interrupt.

Note the following:

1. **The PIA has interrupt enable bits, whereas the 6809 microprocessor has interrupt mask flags.** That is, the PIA bits must be '1' to allow interrupts, while the microprocessor flags must be '0' to have the same effect.
2. **RESET clears the PIA control register and thus disables all interrupts.** Even if the PIA interrupt outputs are tied to \overline{NMI} on the 6809 CPU, no interrupts will occur until the PIA enable bits have been set.
3. **The CPU can check bits 6 and 7 of the Control Register to see if a PIA has a pending interrupt.** Once a status bit has been set, it remains set until the CPU reads the corresponding PIA Data register.
4. **The PIA will remember an interrupt that occurs while PIA interrupts are disabled** and will provide an output as soon as the corresponding enable bit is set.

6850 ACIA INTERRUPTS

The 6850 ACIA can also produce interrupts. You should note the following features of the 6850 ACIA in interrupt-based systems:

1. The transmitter interrupt (signifying that the ACIA is ready for data) is enabled only if Control Register bit 6 = 0 and Control Register bit 5 = 1.
2. The receiver interrupt (signifying that the ACIA has received new data) is enabled only if Control Register bit 7 = 1.

3. Master reset does not affect the interrupt enable bits.
4. The occurrence of either interrupt sets bit 7 of the Status Register. Either reading data from the ACIA or writing data into the ACIA clears bit 7.

6809 POLLING INTERRUPT SYSTEMS

Most 6809 interrupt systems must poll each PIA and ACIA to determine which one caused an interrupt. The polling method is:

1. Check each PIA by examining bits 6 and 7 of the Control Register:

LDA	PIACRA	IS BIT 7 SET?
BMI	INTRP1	YES, INTERRUPT 1 HAS OCCURRED
ASLA		IS BIT 6 SET?
BMI	INTRP2	YES, INTERRUPT 2 HAS OCCURRED

2. Check each 6850 ACIA by examining bit 7 of the Status Register:

LDA	ACIASR	IS BIT 7 SET?
BPL	NXTCHK	NO, NO INTERRUPTS ON THIS ACIA
LSRA		YES, IS BIT 0 SET?
BCS	RCVINT	YES, RECEIVER INTERRUPT HAS OCCURRED
BRA	TXINT	NO, IT MUST HAVE BEEN TRANSMITTER INTERRUPT

Bit 7 of the ACIA Status Register indicates that either a receiver or a transmitter interrupt has occurred. Bit 0 will be set if a receiver interrupt has occurred and bit 1 will be set if a transmitter interrupt has occurred. Of course, the interrupt must be one or the other, so our program assumes a transmitter interrupt if it does not find a receiver interrupt.

The important features of a 6809 polling interrupt system are:

1. The order in which status bits are examined determines the priority of the interrupts. Obviously, the CPU will proceed no further if it finds an active interrupt; thus it ignores activity from sources later in the sequence. Priorities are easy to establish (merely by selecting the order of examination) but difficult to change or vary.
2. The service routine must clear a PIA interrupt by reading the corresponding Data Register, even if the port is being used for output or no data transfer is necessary. Otherwise, the interrupt will remain active. The programmer can use the TST (test zero or minus) instruction to read the PIA Data Register without changing its contents or the contents of a User Register.

DISADVANTAGES OF POLLING INTERRUPTS

Polling routines are adequate if the number of sources is small and the frequency of interrupts is low. If there are many sources or interrupts are frequent, polling routines are slow and awkward for the following reasons.

1. **The average number of polling operations increases linearly with the number of inputs.** On the average, of course, a polling routine will have to examine half of the inputs before finding the active one. You can reduce the average number of polling operations somewhat by checking the most frequent inputs first.
2. PIA and ACIA addresses are rarely consecutive or evenly spaced; therefore, **separate instructions are necessary to examine each input.** Polling routines are therefore difficult to expand. You can use tables of I/O addresses, accessed via one of the indexed addressing modes.
3. **Interrupts that are polled early may shut out those that are polled later unless the order of polling is varied.** However, varying the order of polling is difficult since the addresses are not consecutive.

6809 VECTORED INTERRUPT SYSTEMS

The problem of polling in 6809-based systems is typically solved by special methods, unique to a particular application or microcomputer. **The 6828 Priority Interrupt Controller⁵ provides an eight-level vectored interrupt system based on the regular interrupt input.** This device simply recognizes the addresses FFF8 and FFF9 (see Table 15-1) when they appear on the address bus and replaces them with one of the eight vectors. Special hardware can also utilize the interrupt acknowledge signal provided by the 6809 microprocessor. We will not discuss 6809 vectored interrupt systems any further.

COMMUNICATIONS BETWEEN MAIN PROGRAM AND SERVICE ROUTINES

A major problem in writing programs for interrupt-based systems is providing communications between the main program and the service routines. The criteria for communications methods are:

1. They should not interfere with the normal execution of the main program.
2. They should not depend on how the main program operates. For example, they should not require the main program to be inactive (i.e., executing CWAI or SYNC) or assume that certain registers are always available.
3. They should be well-defined and capable of handling varied amounts of data.
4. They should not require instantaneous action by the main program. The more patient the system is, the easier it will be to develop and maintain.

The idea is to make the service routines and the main program transparent to each other. This approach allows the programmer to change one without affecting the other. It also helps limit errors to one or the other, rather than to the connection between them.

SOFTWARE HANDSHAKE

A simple approach to communications is a software “handshake,” much like the hardware handshake used in asynchronous input/output as described in Chapter 12. The provider of data (the interrupt service routine for input or the main program for output) sets a flag to indicate that new data is available. The receiver or acceptor of data can then examine that flag (sometimes called a *semaphore*) and can clear it after transferring (or accepting) the data. The receiver may itself set another flag (an acknowledgment) to indicate that the most recent data has been processed and more can be sent.

Where do we place the flags and the data? A simple approach is to use a single memory location for each flag and for the data; the location can be a specific memory address or an address in the Hardware Stack that has been set aside for that purpose. The main program and the service routines then communicate through those locations, much as the processor communicates with I/O devices through I/O ports.

BUFFERED INTERRUPTS

The approach outlined above assumes that we handle I/O on a byte-by-byte basis. The processor must provide each output byte separately and must handle each input byte separately. Clearly, all operations must proceed at a rate that is guaranteed to be fast enough to avoid losing data. As with normal I/O, we can relax the time constraints by using buffers. In this approach, the service routine transfers data to or from a buffer and updates the buffer pointer for the next operation. The only time the main program has to be concerned is when an input buffer is full or an output buffer is empty. The service routines act as I/O devices that have their own local memory in which data can be stored temporarily. This approach is referred to as *buffered interrupts*.

Double Buffering

In fact, we can extend this approach. We can provide one buffer for the service routine and a separate buffer for the main program. Now even the filling or emptying of a buffer creates no problem as long as the other buffer is immediately available. In fact, the identities of the buffers can simply be interchanged when the service routine has filled or emptied its buffer. This approach is known as *double buffering*; it allows interrupt-driven input/output to proceed in almost total independence of the main program.

ENABLING AND DISABLING INTERRUPTS

A further problem in writing programs for interrupt-based systems is deciding when to enable or disable interrupts.

WHEN TO DISABLE INTERRUPTS

In general, you disable interrupts in the situations itemized below.

1. **During the initialization of the interrupt system itself.** This may involve loading initial values into pointers, flags, and counters or determining an initial order for polling or other operations. Remember that RESET automatically disables the CPU and PIA interrupts, so the system startup routine will have to explicitly enable them.
2. **During the servicing of an interrupt.** If interrupts are not disabled at least until the current one is cleared, the computer will enter an endless loop with the interrupt endlessly interrupting its own service routine. Remember that the 6809 microprocessor automatically disables the regular interrupt as part of its normal response. Note also that an $\overline{\text{NMI}}$ interrupt will not interfere with its own service routine, since the input is edge-sensitive, rather than level-sensitive.
3. **During operations that occur in real time (such as delay loops or high-speed synchronous I/O) or that could produce erroneous results if interrupted.** A typical example of the latter situation is the updating of multi-byte data that the interrupt service routine must use, such as the calendar time or geographical position. A partial update could produce a highly erroneous value, such as a clock time that is off by an hour or a day because the interrupt occurred before that part of the time was changed.

WHEN TO ENABLE INTERRUPTS

On the other hand, **you want to enable interrupts as soon as possible whenever they might occur.** Otherwise, the system could miss an interrupt and either lose some input or fail to produce the proper output data.

INITIALIZING THE INTERRUPT SYSTEM

The normal order in which you initialize a 6809-based interrupt system is as follows (starting from RESET):

1. Initialize all system parameters.
2. Enable the interrupts from each PIA and ACIA.
3. Enable the CPU interrupts by clearing the appropriate interrupt mask flags.

PIA INTERRUPTS

If you must disable a particular interrupt, you can disable it independently of other interrupts by clearing the interrupt enable flag for a specific port of a particular PIA. You can do this without affecting other control register bits by using logical operations.

1. **Disabling a PIA interrupt.**

Control line 1

```
LDA  PIACR
ANDA  #%11111110  DISABLE CONTROL LINE 1 INTERRUPT
STA  PIACR
```

or (if you know that the interrupt is currently enabled)

```
DEC  PIACR      DISABLE CONTROL LINE 1 INTERRUPT
```

Control line 2

```
LDA  PIACR
ANDA #%11110111  DISABLE CONTROL LINE 2 INTERRUPT
STA  PIACR
```

2. Enabling a PIA interrupt.

Control line 1

```
LDA  PIACR
ORA  #%00000001  ENABLE CONTROL LINE 1 INTERRUPT
STA  PIACR
```

or (if you know that the interrupt is currently disabled)

```
INC  PIACR      ENABLE CONTROL LINE 1 INTERRUPT
```

Control line 2

```
LDA  PIACR
ORA  #%00001000  ENABLE CONTROL LINE 2 INTERRUPT
STA  PIACR
```

The INC and DEC instructions take advantage of the fact that bit 0 is the interrupt enable for control line 1. However, these instructions can affect the entire PIA control register if they are inadvertently executed when the interrupt enable is already in the desired state. Consider, for example, what would happen if the CPU were to execute DEC PIACR when bit 0 of the control register was already 0. The INC and DEC instructions are thus less general, as well as more difficult for the casual reader to understand, than the logical instructions.

SAVING AND RESTORING INTERRUPT STATUS

A related problem⁶ is restoring the original state of the interrupt system after performing operations that require interrupts to be disabled. If, for example, an I/O or other subroutine must be executed with interrupts disabled, the subroutine must restore the original state of the interrupt system before returning control to the main program. Clearly, we do not want the subroutine to enable interrupts if they were disabled in the calling program or not re-enable interrupts if they were enabled in the calling program.

The solution is simple: save and restore the condition code register that contains the interrupt mask bits. We can save that register before disabling interrupts with the instruction PSHS CC; we can restore that register before returning control to the main program or performing operations that could be interrupted with the instruction PULS CC.

CHANGING THE VALUES IN THE STACK

The 6809 microprocessor automatically saves all or some of its registers in response to an interrupt; the RTI instruction at the end of the service routine restores those registers. Most service routines leave the registers in the stack alone to promote generality and simplicity. However, programmers occasionally find it necessary to alter some of the registers. Typical reasons are to force a return to a different address⁷ or to disable the entire interrupt system. In these cases, the programmer must know how to find the various registers in the Hardware Stack.

Table 15-2. Indexed Offsets for Entire Processor State

Register	Indexed Offset (Hexadecimal)
Condition Code	00
Accumulator A	01
Accumulator B	02
Direct Page Register	03
High-Order Byte of Index Register X	04
Low-Order Byte of Index Register X	05
High-Order Byte of Index Register Y	06
Low-Order Byte of Index Register Y	07
High-Order Byte of User Stack Pointer U	08
Low-Order Byte of User Stack Pointer U	09
High-Order Byte of Program Counter	0A
Low-Order Byte of Program Counter	0B

Table 15-3. Indexed Offsets for Limited Processor State

Register	Indexed Offset (Hexadecimal)
Condition Code	00
High-Order Byte of Program Counter	01
Low-Order Byte of Program Counter	02

Table 15-2 contains the indexed offsets required to access the registers in the case in which the processor has saved the entire state. Table 15-3 contains the indexed offsets required in the case (in response to $\overline{\text{FIRQ}}$) in which the processor has saved only the program counter and the condition code register. **Typical routines using the offsets in Table 15-2 are:**

1. **Changing the return address to EEXIT (an error exit routine):**

```
LDX #EEXIT      RETURN ADDRESS = ERROR EXIT
STX $0A,S
```

2. **Decrementing the return address by 1 (used in the event that an interrupt or SWI instruction has been used to replace an actual program instruction for debugging or testing purposes):**

```
TST $0B,S      ARE LSB'S OF RETURN ADDRESS ZERO?
BNE DECLSB    YES, REDUCE MSB'S BY 1
DEC $0A,S
DECLSB DEC $0B,S  REDUCE LSB'S OF RETURN ADDRESS BY 1
```

Only if the LSB's of the return address are zero is it necessary to decrement the MSB's in order to produce a correct 16-bit decrement.

3. **Disabling the regular interrupt by setting the regular interrupt mask bit (bit 4 of the Condition Code Register):**

```
LDA ,S
ORA #00010000  DISABLE REGULAR INTERRUPT
STA ,S
```

4. Disabling the fast interrupt by setting the fast interrupt mask bit (bit 6 of the Condition Code Register):

```
LDA    ,S
ORA    #%01000000  DISABLE FAST INTERRUPT
STA    ,S
```

Obviously, the programmer must be extremely careful when altering stack values, since these changes could have unforeseen side effects in the main program.

INTERRUPT OVERHEAD

Responding to an interrupt always involves some overhead cycles, since the CPU may have to fetch a new program counter value from memory and save registers in the Hardware Stack. Of course, the restoring of registers from the Hardware Stack, if necessary, also uses processor time. You can determine the amount of overhead involved in servicing interrupts from the time requirements in Table 15-4.

Table 15-4. Time Requirements for Interrupt-Related Operations

Operation	Number of Clock Cycles
Normal response to \overline{IRQ} or NMI	21
Normal response to \overline{FIRQ}	12
Response to any interrupt while executing $CWAI$	9
Escape from SYNC state if interrupts disabled	1
Execution of $CWAI$	20
Execution of RTI with E flag set (Entire state)	15
Execution of RTI with E flag cleared (Limited state)	6
Execution of SWI	19
Execution of $SWI2$ or $SWI3$	20
Execution of SYNC	2

PROGRAM EXAMPLES

15-1. A STARTUP INTERRUPT

Purpose: The computer waits for a PIA interrupt to occur before starting actual operations.

Often a system remains inactive until the operator actually starts it or until a DATA READY signal is received. On RESET, such a system must initialize the Stack Pointer, enable the startup interrupt, and execute a halt (CWAI) or an endless loop or jump-to-self instruction. Remember that RESET disables the processor interrupt (by setting I and F both to 1) as well as all the PIA interrupts (by clearing all the PIA interrupt enable bits). In the flowchart, the decision as to whether startup is active is made in hardware (by the CPU examining the interrupt input internally) rather than in software.

Program 15-1:

Main Program:

```

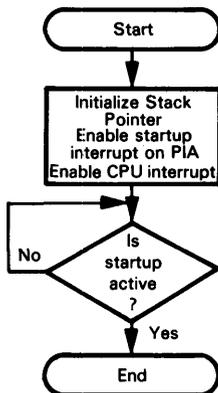
      8001   PIACA EQU   $8001
      8000   PIADA EQU   $8000
      0100   INTRP EQU   $0100
          *
0000          ORG   $0000
0000 10CE 0100   LDS   #$100      START STACK AT MEMORY LOCATION
          *                                     00FF
0004 86   05          LDA   %%00000101  ENABLE INTERRUPT FROM
0006 B7   8001        STA   PIACA      STARTUP PIA
0009 3C   EF          CWA.I %%11101111  ENABLE REGULAR INTERRUPT
          *                                     AND WAIT
000B 3F          SWI          DUMMY CONTINUATION
    
```

Interrupt Service Routine:

```

0100          ORG   INTRP
0100 B6   8000        LDA   PIADA      CLEAR STARTUP INTERRUPT
0103 3B          RTI          RETURN AND PROCEED
    
```

Flowchart:



The exact location (INTRP) of the interrupt service routine varies with the microcomputer. If your microcomputer has no monitor, you can simply place whatever address you want in memory locations FFF8 and FFF9 (or whatever locations respond to those addresses). You must then start the interrupt service routine at the address you chose. Of course, you should locate the routine so it does not interfere with fixed addresses or with other programs.

Interrupt Handling by Monitors

If your microcomputer has a monitor, the monitor will reserve addresses FFF8 and FFF9. Those addresses will either contain the starting address at which you must place your interrupt service routine, or will contain the starting address of a routine that allows you to choose the starting address of the interrupt service routine. A typical monitor routine would be:

```

MONINT JMP [USRINT]      JUMP TO USER-SUPPLIED SERVICE ADDRESS
    
```

You must then place the starting address of your service routine in memory locations `USRINT` and `USRINT + 1`. Remember that `MONINT` is an address in the monitor program and its value is in addresses `FFF8` and `FFF9`.

You can include the loading of memory locations `USRINT` and `USRINT + 1` in your main program:

```

LDX  #INTRP      GET STARTING ADDRESS OF SERVICE ROUTINE
STX  USRINT      STORE IT AT ADDRESS MONITOR USES
    
```

These instructions must precede the enabling of the interrupts.

Program Operation

The main program's only action is to enable the interrupt from the startup PIA. The program enables that interrupt by setting bit 0 of the PIA Control Register before enabling the CPU interrupt. Note that we must set the PIA interrupt enable and clear the CPU interrupt mask bit.

The `CWAI` instruction logically ANDs the Condition Code Register with the following byte of immediate data before halting instruction execution. Logically ANDing the Condition Code Register with `111011112` clears bit 4 of the Condition Code Register, thus enabling the regular interrupt. Similarly, logically ANDing with `101111112` would clear bit 6 of the Condition Code Register, thus enabling the fast interrupt. Logically ANDing with `101011112` would enable both maskable interrupts.

`CWAI` causes the 6809 CPU to save all its registers in the Hardware Stack and wait for an interrupt to occur.

In response to an interrupt (IRQ), the CPU disables IRQ and transfers control to the address in memory locations `FFF8` and `FFF9`. (Remember that all the registers have already been saved in the Hardware Stack.)

The service routine clears the startup interrupt by reading the appropriate PIA Data Register. This operation is necessary, even though no data transfer is required. Otherwise, the startup interrupt would remain active and would interrupt again as soon as the CPU interrupt was reenabled.

`RTI` restores all the user registers from the Hardware Stack, thus reenabling the CPU interrupt (since the old flag is restored) and transferring control to the instruction immediately following `CWAI`. Note that transferring control to the service routine does not change the contents of the user registers, but `RTI` does. The `LDA` instruction in the service routine affects Accumulator A and the Condition Code Register, but those effects are lost when `RTI` is executed.

15-2. A KEYBOARD INTERRUPT

Purpose: The main program clears a flag in memory location `0040` and waits for a keyboard interrupt. The interrupt service routine sets the flag in memory location `0040` to 1 and places the data from the keyboard in memory location `0041`.

Sample Problem:

```

Keyboard data = 43
Result: (0040) = 01  Flag indicating new keyboard data
        (0041) = 43  Keyboard data
    
```

15-18 6809 Assembly Language Programming

Program 15-2a:

Main Program:

```

      8001  PIACA EQU $8001
      8000  PIADDA EQU $8000
      8000  PIADA EQU $8000
      0100  INTRP EQU $0100
      *
0000          ORG $0000
0000 10CE 0100 *   LDS $100    START STACK AT MEMORY LOCATION
                        00FF
0004 0F 40      CLR $40     CLEAR DATA READY FLAG
0006 7F 8001    CLR PIACA   ADDRESS DATA DIRECTION REGISTER
0009 7F 8000    CLR PIADDA  MAKE ALL DATA LINES INPUTS
000C 86 05     LDA #$0000101  ENABLE KEYBOARD INTERRUPT
000E B7 8001    STA PIACA   ON PIA
0011 1C EF     ANDCC #$11101111  ENABLE CPU INTERRUPT
0013 0D 40     WTRDY TST $40   IS THERE DATA FROM THE KEYBOARD?
0015 27 FC     BEQ WTRDY    NO, WAIT
0017 3F        SWI         YES, PROCEED
    
```

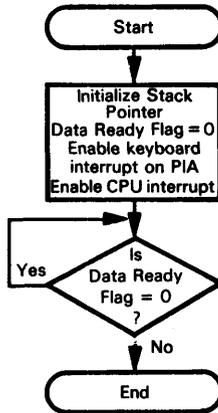
Interrupt Service Routine:

```

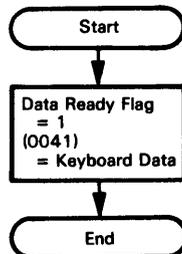
0100          ORG INTRP
0100 0C 40     INC $40     SET DATA READY FLAG
0102 B6 8000   LDA PIADA   FETCH DATA FROM KEYBOARD
0105 97 41     STA $41     SAVE DATA IN MEMORY
0107 3B       RTI
    
```

Flowchart:

Main Program:



Interrupt Service Routine:



You must initialize the PIA completely before enabling interrupts. This includes establishing the directions of ports and control lines and determining the transitions to be recognized on input strobes.

The main program clears the Data Ready Flag (memory location 0040) and then simply waits for the interrupt service routine to set it. The main program and the service routine communicate through two fixed memory addresses:

0040 is a flag that indicates whether new data has been received from the keyboard.

0041 is a single-location data buffer used to hold the value received from the keyboard.

Note the similarity between the Data Ready Flag in memory and the status bit in the control register of the keyboard PIA. The program does not have to test bit 7 of the PIA control register because there is a direct hardware (interrupt) connection between that bit and the CPU. Of course, we have also assumed that the keyboard is the only source of interrupts.

The RTI instruction at the end of the service routine transfers control back to the main program. If you want to transfer control somewhere else (perhaps an error routine), you can change the Program Counter in the Hardware Stack using the methods outlined earlier. If the entire state of the processor has been saved, the return address will be at offsets $0A_{16}$ and $0B_{16}$ from the Hardware Stack Pointer.

We do not use the registers to pass parameters and results. In the first place, the 6809 automatically restores the old register values when it executes RTI. Secondly, if we were to change the register values in the stack, we could interfere with the execution of the main program. In most applications, the main program is using the registers and random changes will cause havoc. At the very least, changing the registers lacks generality, since modifications to the main program surely could result in the use of registers that are currently available.

The service routine does not have to explicitly re-enable the interrupts. The reason is that RTI automatically restores the old Condition Code Register with the Interrupt Mask bit in its original (cleared) state. In fact, you will have to change the Interrupt Mask bit in the Stack (bit 4 of the top location) if you do not want the interrupts to be re-enabled.

You can save and restore other data (such as the contents of a memory location) by using the Hardware Stack. This method can be expanded indefinitely (as long as there is RAM available for the Stack), since nested service routines will not destroy the data saved by earlier routines.

Filling a Buffer via Interrupts

An alternative approach would be for the interrupt service routine to set memory location 0040 only after receiving an entire line of text (such as a string of characters ending with a carriage return). Here we use memory location 0040 as an end-of-line flag and memory locations 0041 and 0042 as a buffer pointer. We will assume that the buffer starts in memory location 0050.

Program 15-2b:**Main Program:**

```

      8001   PIACA EQU $8001
      8000   PIADDA EQU $8000
      8000   PIADA EQU $8000
      0100   INTRP EQU $0100
      000D   CR EQU $0D
      *
0000      ORG $0000
0000 10CE 0100      LDS #$100    START STACK AT MEMORY LOCATION
      *                                00FF
0004 0F 40      CLR $40    CLEAR END OF LINE FLAG
0006 8E 0050     LDX #$50    INITIALIZE BUFFER POINTER TO
0009 9F 41      STX $41    START OF BUFFER
000B 7F 8001     CLR PIACA   ADDRESS DATA DIRECTION REGISTER
000E 7F 8000     CLR PIADDA  MAKE ALL DATA LINES INPUTS
0011 86 05      LDA #00000101  ENABLE KEYBOARD INTERRUPT
0013 B7 8001     STA PIACA   FROM PIA
0016 1C EF      ANDCC #11101111  ENABLE CPU INTERRUPT
0018 0D 40      WTEOL TST $40   HAS A LINE BEEN RECEIVED FROM
      *                                THE KEYBOARD?
001A 27 FC      BEQ WTEOL    NO, WAIT
001C 3F         SWI

```

Interrupt Service Routine:

```

0100      ORG INTRP
0100 9E 41      LDX $41    GET BUFFER POINTER
0102 B6 8000     LDA PIADA   FETCH DATA FROM THE KEYBOARD
0105 A7 80      STA ,X+    SAVE DATA IN BUFFER AND
      *                                INCREMENT POINTER
0107 9F 41      STX $41    STORE ADJUSTED BUFFER POINTER
0109 81 0D      CMPA #CR    IS DATA A CARRIAGE RETURN?
010B 26 02      BNE DONE
010D 0C 40      INC $40    YES, SET END OF LINE FLAG
010F 3B         DONE RTI

```

This program fills a buffer starting at memory location 0050 until it receives a carriage return character (CR). Memory locations 0041 and 0042 hold the current buffer pointer. The interrupt service routine increments that pointer (with autoincrementing) after each use.

In a real application, the CPU could perform other tasks between interrupts. It could, for example, edit, move, or transmit a line from one buffer while the interrupt was filling another buffer. This is the double buffering approach. The main program only has to ensure that no buffers ever overflow.

An alternative approach would be for memory location 0040 to contain a counter rather than a flag. The contents of that location would then indicate to the main program how many bytes of data had been received. The main program could then deal with the buffer whenever a certain number of new data bytes were in it. The service routine would simply increment the counter as well as the buffer pointer as part of each input operation.

15-3. A PRINTER INTERRUPT

Purpose: The main program clears a flag in memory location 0040 and waits for a printer interrupt. This interrupt service routine sets the flag in memory location 0040 to 1 and sends the contents of memory location 0041 to the printer.

Sample Problem:

(0041) = 51

Result: (0040) = 01 Flag indicating last data item has been sent

Printer receives a 51₁₆ (ASCII Q) when it is ready.

Program 15-3a:

Main Program:

```

            8003   PIACB EQU   $8003
            8002   PIADDB EQU  $8002
            8002   PIADB  EQU   $8002
            0100   INTRP EQU   $0100
            *
0000        ORG    $0000
0000 10CE 0100 *   LDS    #$100    START STACK AT MEMORY LOCATION
                                00FF
                                CLR    $40    CLEAR DATA ACCEPTED FLAG
0004 0F   40      *   CLR    PIACB   ADDRESS DATA DIRECTION REGISTER
0006 7F   8003    LDA    #$FF    MAKE ALL DATA LINES OUTPUTS
0009 86   FF      STA    PIADDB
000B B7   8002    LDA    ##00000101  ENABLE PRINTER INTERRUPT
000E 86   05      STA    PIACB   ON PIA
0010 B7   8003    ANDCC  ##11101111  ENABLE CPU INTERRUPT
0013 1C   EF      WTACK TST    $40    HAS THE PRINTER ACCEPTED THE
0015 0D   40      *   BEQ    WTACK   DATA?
                                NO, WAIT
0017 27   FC      *   SWI
0019 3F

```

Interrupt Service Routine:

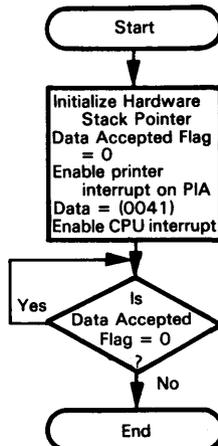
```

0100        ORG    INTRP
0100 0C   40      INC    $40    SET DATA ACCEPTED FLAG
0102 96   41      LDA    $41    GET DATA FOR PRINTER
0104 B7   8002    STA    PIADB   SEND DATA TO PRINTER
0107 B6   8002    LDA    PIADB   CLEAR PRINTER INTERRUPT
010A 3B
RTI

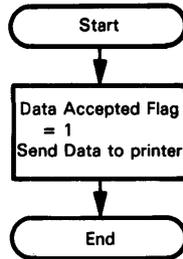
```

Flowchart:

Main Program:



Interrupt Service Routine:



The only differences from the keyboard interrupt routines are the meaning of the flag, the direction of the data transfer, and the need for the dummy instruction LDA PIADB to clear bit 7 of the PIA Control Register. Remember that an input operation automatically clears that bit, but an output operation does not.

Here the flag in memory location 0040 indicates that the CPU has data available that has not yet been sent to the printer. When the interrupt service routine sets the flag, the main program knows the data has been sent. The flag acts as an acknowledgement from the printer or a Data Accepted indicator.

Remember that you may find it necessary to place a dummy read at the start of the main program to clear stray interrupts. LDA PIADB or TST PIADB will do the job, as long as you place it after the instruction that addresses the data register but before the instruction that enables CPU interrupts.

Emptying a Buffer with Interrupts

As in the keyboard example, we could have the interrupt service routine set the Data Accepted flag after it sends the printer an entire line of data ending with a carriage return. Here again we use memory location 0040 as an end-of-line flag and memory locations 0041 and 0042 as a buffer pointer. We will assume that the buffer starts in memory location 0050.

Program 15-3b:

Main Program:

```

      8003   PIACB EQU   $8003
      8002   PIADDB EQU  $8002
      8002   PIADB EQU   $8002
      0100   INTRP EQU   $0100
      000D   CR     EQU   $0D
      *
0000                                ORG   $0000
0000 10CE 0100                       *   LDS   #$100   START STACK AT MEMORY LOCATION
                                          OFF
0004 0F   40                         CLR   $40   CLEAR END OF LINE FLAG
0006 8E   0050                       LDX   #$50   INITIALIZE BUFFER POINTER TO
0009 9F   41                         STX   $41   START OF BUFFER
000B 7F   8003                       CLR   PIACB ADDRESS DATA DIRECTION REGISTER
000E 86   FF                         LDA   #$FF  MAKE ALL DATA LINES OUTPUTS
0010 B7   8002                       STA   PIADDB
0013 86   05                         LDA   #$00000101 ENABLE PRINTER INTERRUPT
0015 B7   8003                       STA   PIACB FROM PIA
0018 1C   EF                         ANDCC #$11101111 ENABLE CPU INTERRUPT
001A 0D   40   WTEOL TST   $40   HAS A LINE BEEN PRINTED?
001C 27   FC                         BEQ   WTEOL NO, WAIT
001E 3F                               SWI
  
```

Interrupt Service Routine:

```

0100                ORG     INTRP
0100 9E    41      LDX     $41      GET BUFFER POINTER
0102 A6    80      LDA     ,X+     GET DATA FROM BUFFER AND
                                *      INCREMENT POINTER
0104 B7    8002   STA     PIADB    SEND DATA TO PRINTER
0107 7D    8002   TST     PIADB    CLEAR PRINTER INTERRUPT
010A 9F    41     STX     $41     STORE ADJUSTED BUFFER POINTER
010C 81    0D     CMPA    #CR     IS DATA A CARRIAGE RETURN?
010E 26    02     BNE     DONE    *
0110 0C    40     INC     $40     YES, SET END OF LINE FLAG
0112 3B                DONE    RTI

```

We could use double buffering to allow I/O and processing to occur independently without ever halting the CPU.

Fixed-Length Buffer

Still another approach uses memory location 0040 as a buffer counter. For example, the following program waits for 20 characters to be sent to the printer.

Program 15-3c:

Main Program:

```

                8003   PIACB EQU   $8003
                8002   PIADDB EQU  $8002
                8002   PIADB EQU   $8002
                0100   INTRP EQU   $0100
                *
0000                ORG     $0000
0000 10CE 0100      LDS     #$100   START STACK AT MEMORY LOCATION
                                *      00FF
0004 0F    40      CLR     $40     CLEAR BUFFER COUNTER
0006 8E    0050   LDX     #$50   INITIALIZE BUFFER POINTER TO
0009 9F    41     STX     $41     START OF BUFFER
000B 7F    8003   CLR     PIACB   ADDRESS DATA DIRECTION REGISTER
000E 86    FF     LDA     #$FF   MAKE ALL DATA LINES OUTPUTS
0010 B7    8002   STA     PIADDB
0013 86    05     LDA     #$0000101  ENABLE PRINTER INTERRUPT
0015 B7    8003   STA     PIACB   FROM PIA
0018 1C    EF     ANDCC  #$11101111  ENABLE CPU INTERRUPT
001A 8C    14     LDA     #20   TARGET COUNT = 20
001C 91    40     WTCNT  CMPA    $40   HAS TARGET COUNT BEEN REACHED?
001E 26    FC     BNE     WTCNT  NO, WAIT
0020 3F                SWI                YES, PROCEED

```

Interrupt Service Routine:

```

0100                ORG     INTRP
0100 9E    41      LDX     $41      GET BUFFER POINTER
0102 A6    80      LDA     ,X+     GET DATA FROM BUFFER AND
                                *      INCREMENT POINTER
0104 B7    8002   STA     PIADB    SEND DATA TO PRINTER
0107 7D    8002   TST     PIADB    CLEAR PRINTER INTERRUPT
010A 9F    41     STX     $41     STORE ADJUSTED BUFFER POINTER
010C 0C    40     INC     $40     INCREMENT BUFFER COUNTER
010E 3B                RTI

```

15-4. A REAL-TIME CLOCK INTERRUPT

Purpose: The computer waits for an interrupt from a real-time clock.

Real-Time Clock

A real-time clock simply provides a regular series of pulses. The interval between the pulses can be used as a time reference. Real-time clock interrupts can be counted to give any multiple of the basic time interval. A real-time clock can be produced by dividing down the CPU clock, by using a timer like the 6840 device or the one included in the 6846 multifunction support device, or by using external sources such as the AC line frequency.

Note the tradeoffs involved in determining the frequency of the real-time clock. A high frequency (say 10 kHz) allows the creation of a wide range of time intervals of high accuracy. On the other hand, the overhead involved in counting real-time clock interrupts may be considerable, and the counts will quickly exceed the capacity of a single 8-bit register or memory location. **The choice of frequency depends on the precision and timing requirements of your application.** The clock may, of course, consist partly of hardware; a counter may count high frequency pulses and interrupt the processor only occasionally. A program will have to read the counter to measure time to high accuracy.

One problem is synchronizing operations with the real-time clock. Clearly, there will be some effect on the precision of the timing interval if the CPU starts the measurement randomly during a clock period, rather than exactly at the beginning. **Some ways to synchronize operations are:**

1. **Start the CPU and clock together.** RESET or a startup interrupt can start the clock as well as the CPU.
2. **Allow the CPU to start and stop the clock under program control.**
3. **Use a high-frequency clock** so that an error of less than one clock period will be small.
4. **Line up the clock** (by waiting for an edge or interrupt) **before starting the measurement.**

A real-time clock interrupt should have very high priority, since the precision of the timing intervals will be affected by any delay in servicing the interrupt. **The usual practice is to make the real-time clock the highest priority interrupt except for power failure.** The clock interrupt service routine is generally kept extremely short so that it does not interfere with other CPU activities.

15-4a. Wait for Real-Time Clock

Program 15-4a:

Main Program:

```

      8001  PIACA  EQU  $8001
      8000  PIADA  EQU  $8000
      0100  INTRP  EQU  $0100
          *
0000          ORG  $0000
0000 10CE 0100  LDS  #$1000  START STACK AT MEMORY LOCATION
          *
0004 0F   40          CLR  $40    CLEAR CLOCK COUNTER TO START
0006 86   05          LDA  #80000101  ENABLE REAL-TIME CLOCK
0008 B7   8001        STA  PIACA  INTERRUPT
000B 1C   EF          ANDCC #11101111  ENABLE CPU INTERRUPT
000D 0D   40          WTCLK TST  $40    HAS CLOCK BEEN INCREMENTED?
000F 27   FC          BEQ  WTCLK  NO, WAIT
0011 3F          SWI          YES, PROCEED

```

Interrupt Service Routine:

```

0100                                ORG    INTRP
0100 B6    8000                    LDA    PIADA    CLEAR CLOCK INTERRUPT
0103 0C    40                      INC    $40     INCREMENT CLOCK COUNTER
0105 3B                                RTI

```

Memory location 0040 contains the clock counter.

If bit 1 of the PIA Control Register is 0, the interrupt will occur on the high-to-low (falling) edge of the clock. If that bit is 1, the interrupt will occur on the low-to-high (rising) edge of the clock.

The interrupt service routine must explicitly clear bit 7 of the PIA Control Register since no data transfer is necessary.

You could still use the PIA data port as long as you did not accidentally clear the status bit from the real-time clock before it was recognized. This would be no problem if the port were used for output to a simple peripheral (such as a set of LEDs), since output operations do not affect the status bits anyway.

Clearly, we can easily extend this routine to handle more counts and provide greater precision by using more memory locations for the clock counter and a different test in the main program.

15-4b. Wait for 10 Clock Interrupts

Program 15-4b:

Main Program:

```

                                8001    PIACA EQU    $8001
                                8000    PIADA EQU    $8000
                                0100    INTRP EQU    $0100
                                *
0000                                ORG    $0000
0000 10CE 0100                    LDS    #$100    START STACK AT MEMORY LOCATION
                                *                                00FF
0004 0F    40                      CLR    $40    CLEAR CLOCK COUNTER TO START
0006 86    05                      LDA    #$00000101  ENABLE REAL-TIME CLOCK
0008 B7    8001                    STA    PIACA    INTERRUPT
000B 1C    EF                      ANDCC  #$111101111  ENABLE CPU INTERRUPT
000D 86    0A                      LDA    #10     TARGET COUNT = 10
000F 91    40                      WTCNT CMPA   $40    HAS CLOCK COUNTER REACHED TARGET
                                *                                COUNT?
0011 26    FC                      BNE   WTCNT    NO, WAIT
0013 3F                                SWI

```

Interrupt Service Routine:

```

0100                                ORG    INTRP
0100 B6    8000                    LDA    PIADA    CLEAR CLOCK INTERRUPT
0103 0C    40                      INC    $40     INCREMENT CLOCK COUNTER
0105 3B                                RTI

```

15-4c. Maintaining Real Time

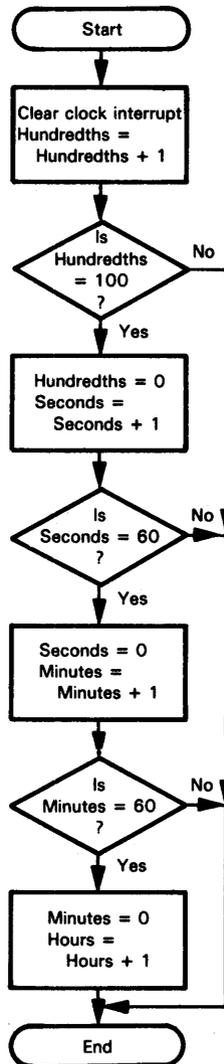
A more realistic real-time clock interrupt routine could keep track of the passage of time using several memory locations. For example, the following routine uses addresses 0040 through 0043 to maintain clock (calendar) time as follows:

- 0040 - hundredths of seconds
- 0041 - seconds
- 0042 - minutes
- 0043 - hours

15-26 6809 Assembly Language Programming

We assume that a 100 Hz input clock provides the regular source of interrupts.

Flowchart:



Program 15-4c:

Interrupt Service Routine:

```

            8000   PIADA EQU   $8000
            0100   INTRP EQU   $0100
            *
0100        ORG    INTRP
0100 B6     8000   LDA    PIADA    CLEAR CLOCK INTERRUPT
0103 8E     0040   LDX    #$40
0106 6C     84    INC     ,X        UPDATE HUNDREDTHS OF SECONDS
0108 86     64    LDA    #100     IS THERE A CARRY TO SECONDS?
010A A1     84    CMPA   ,X
010C 26     16    BNE   ENDINT    NO, DONE
010E 6F     84    CLR     ,X        YES, MAKE HUNDREDTHS ZERO
0110 6C     01    INC     1,X      UPDATE SECONDS
0112 86     3C    LDA    #60     IS THERE A CARRY TO MINUTES?
0114 A1     01    CMPA   1,X
0116 26     0C    BNE   ENDINT    NO, DONE
0118 6F     01    CLR     1,X      YES, MAKE SECONDS ZERO
011A 6C     02    INC     2,X      UPDATE MINUTES
011C A1     02    CMPA   2,X      IS THERE A CARRY TO HOURS?
011E 26     04    BNE   ENDINT    NO, DONE
0120 6F     02    CLR     2,X      YES, MAKE MINUTES ZERO
0122 6C     03    INC     3,X      UPDATE HOURS
0124 3B                    ENDINT RTI
    
```

Now we could produce a delay of 300 ms in the main program with the routine:

```

                                LDA    $40        GET CURRENT TIME
                                ADDA   #30        DESIRED TIME IS 30 COUNTS LATER
                                CMPA   #100       MOD 100
                                BCS    WTCNT
                                SUBA   #100
WTCNT CMPA   $40        HAS DESIRED TIME BEEN REACHED?
                                BNE    WTCNT       NO, WAIT
    
```

This approach is the same one you would take if you had to let something cook for 20 minutes. You must determine the current time by reading your watch (the counter), calculate the target time by adding 20 (mod 60, so 20 minutes past 6:50 is 7:10), and wait for your watch to reach the target time. Change the program so it produces a 20 minute delay (an obvious requirement for a microprocessor-controlled microwave oven).

Of course, the program could perform other tasks and only check the elapsed time occasionally. How would you produce a delay of seven seconds? Of three minutes? Many applications do not require long delays to be highly accurate; for example, the operator of a microwave oven does not care if the time intervals are off by a few seconds.

Sometimes you may want to keep time either as BCD digits or as ASCII characters. How would you revise the last interrupt service routine to handle these alternatives?

Service Time for the Real-Time Clock

The complete service routine for a real-time clock may seem long, but it actually uses very little processor time. The execution times are as follows:

Condition	Frequency	Number of clock cycles required
No additional updating	Every 10 ms	59
Update seconds	Every second	82
Update minutes	Every minute	104
Update hours	Every hour	118

Much of the execution time (see Table 15-4) goes to the interrupt response (21 clock cycles) and to the RTI instruction (15 clock cycles), since these require the transfer of many registers to and from the Hardware Stack. Thus the largest number of clock cycles ever used by the real-time clock service routine is 118 during a total period of 10 milliseconds. This is 1.18% of the available processor time if the clock frequency is 1 MHz. The average requirement is half the maximum, since the service routine requires its minimum execution time 99 times out of 100 and only requires its maximum execution time once every 360,000 times (once per hour). Thus a real-time clock generally does not burden the processor very much, unless its frequency is high.

High-Frequency Clock

Even a high-frequency real-time clock can be handled without much processor intervention. The usual method is to have the clock increment a set of counters which then interrupt the processor at a much lower frequency. For example, the input frequency could be 1 MHz; that input frequency would then be passed through 3 decimal counters and the output of the last one would be tied to the PIA. The PIA would receive a single clock pulse for every 1000 input pulses (that is, when the 3 decimal counters overflow). The processor can determine the time to greater precision than 1 ms by reading the counters, since they contain the less significant digits. As usual, some additional hardware (counters and input ports) is necessary to reduce the burden on the CPU. This is a typical tradeoff; the additional hardware is worthwhile only if the application requires precise timing.

15-5. A TELETYPEWRITER INTERRUPT

15-5a. ACIA Interrupt Routine

Purpose: The main program clears a flag in memory location 0040 and waits for an interrupt from a 6850 ACIA. The interrupt service routine sets the flag in memory location 0040 to 1 and places the data from the ACIA in memory location 0041. The characters are 7-bits in length with odd parity and two stop bits.

Program 15-5a:

Main Program:

```

            8010   ACIACR EQU   $8010
            8011   ACIADR EQU   $8011
            0100   INTRP  EQU   $0100
            *
0000                ORG   $0000
0000 10CE 0100      LDS   #$100   START STACK AT MEMORY LOCATION
            *                                     00FF
0004 0F   40        CLR   $40     CLEAR DATA READY FLAG
0006 86   03        LDA   #00000011 MASTER RESET ACIA
0008 B7   8010      STA   ACIACR
000B 86   C5        LDA   #11000101  ENABLE ACIA RECEIVER
000D B7   8010      STA   ACIACR     INTERRUPT
0010 1C   EF        ANDCC #11101111  ENABLE CPU INTERRUPT
0012 0D   40        WTRDY TST   $40   IS THERE DATA FROM THE ACIA?
0014 27   FC        BEQ   WTRDY     NO, WAIT
0016 3F                SWI                YES, PROCEED

```

Interrupt Service Routine:

```

0100          ORG     INTRP
0100 B6 8011    LDA     ACIADR    GET DATA FROM ACIA
0103 97 41     STA     $41      SAVE DATA IN MEMORY
0105 0C 40     INC     $40      SET DATA READY FLAG
0107 3B        RTI
    
```

Since the 6850 ACIA has no RESET input, a MASTER RESET (setting Control register bits 0 and 1 to one simultaneously) is necessary before the ACIA is initialized.

We then initialize the bits in the ACIA control register as follows:

- Bit 7 = 1 to enable the receiver interrupt
- Bit 6 = 1 and Bit 5 = 0 to disable the transmitter interrupt
- Bit 4 = 0, Bit 3 = 0, and Bit 2 = 1 to select 7-bit data with odd parity and two stop bits
- Bit 1 = 0 and Bit 0 = 1 to select the divide by 16 clock mode (a 1760 Hz clock must be supplied for a 110 Baud data rate).

To determine if a particular 6850 ACIA is the source of an interrupt, the program must examine the interrupt request bit (bit 7 of the Status Register). To differentiate between receiver and transmitter interrupts, the program must examine the Receive Data Register Full bit (bit 0 of the Status Register). Either reading the Receive Data Register or writing into the Transmit Data Register clears the ACIA's interrupt request bit.

15-5b. PIA Start Bit Interrupt

Received data is tied to both data bit 7 and control line 1 of the PIA.

Purpose: The main program clears a flag in memory location 0040 and waits for a teletypewriter interrupt. The interrupt service routine sets the flag in memory location 0040 to 1 and places the data from the teletypewriter in memory location 0041.

Program 15-5b:

Main Program:

```

          8001    PIACA EQU     $8001
          8000    PIADDA EQU    $8000
          8000    PIADA EQU    $8000
          0100    INTRP EQU    $0100
          0030    TTYRCV EQU   $0030
          *
0000          ORG     $0000
0000 10CE 0100  *     LDS     #$100    START STACK AT MEMORY LOCATION
                                OFF
0004 0F 40     CLR     $40    CLEAR DATA READY FLAG
0006 7F 8001   CLR     PIACA   ADDRESS DATA DIRECTION REGISTER
0009 7F 8000   CLR     PIADDA  MAKE ALL DATA LINES INPUTS
000C 86 05     LDA     #$0000101  ENABLE START BIT INTERRUPT
000E B7 8001   STA     PIACA   FROM PIA
0011 1C EF     ANDCC  #$11101111  ENABLE CPU INTERRUPT
0013 0D 40     WTSTB  TST     $40    HAS START BIT BEEN RECEIVED?
0015 27 FC     BEQ     WTSTB    NO, WAIT
0017 9D 30     JSR     TTYRCV   YES, FETCH DATA FROM TTY
0019 97 41     STA     $41      SAVE DATA IN MEMORY
001B 3F        SWI
    
```

Interrupt Service Routine:

```

0100                ORG     INTRP
0100 B6    8000     LDA     PIADA    CLEAR START BIT INTERRUPT
0103 0C    40       INC     $40     SET DATA READY FLAG
0105 86    04       LDA     #00000100  DISABLE START BIT INTERRUPT
0107 B7    8001     STA     PIACA
010A 3B                RTI

```

Subroutine TTYRCV is the teletypewriter receive routine shown in Chapter 13, except that we have assumed a version that leaves the data in Accumulator A. The edge used to cause the interrupt is very important here. The transition from the normal '1' (MARK) state to the '0' (SPACE) state must cause the interrupt, since this transition signifies the start of the transmission. No '0' to '1' transition will occur until a non-zero data bit is received.

The service routine must disable the PIA interrupt, since otherwise each '1' to '0' transition in the character will cause an interrupt. Note that reading the data bits will clear any status flags set by the ignored transitions. Of course, the program must reenale the PIA interrupt (by setting bit 0 of the control register) to allow receipt of the next character, but this should be done after the current character has been read.

As we mentioned earlier in this chapter, we can also disable PIA interrupts by using logical functions or the INC and DEC instructions. The following programs are independent of the contents of the PIA Control Register.

1. Disabling the PIA interrupt from control line 1.

```

                LDA     PIACR
                ANDA    #11111110  DISABLE PIA INTERRUPT
                STA     PIACR
or
                DEC     PIACR        DISABLE PIA INTERRUPT

```

2. Enabling the PIA interrupt from control line 1.

```

                LDA     PIACR
                ORA     #00000001  ENABLE PIA INTERRUPT
                STA     PIACR
or
                INC     PIACR        ENABLE PIA INTERRUPT

```

The DEC instruction only works correctly if you know that the interrupt is enabled, while the INC instruction only works correctly if you know that the interrupt is disabled. If the interrupt is already in the desired state, INC or DEC can have curious effects (try it!), whereas the logical functions have no effect in that case.

MORE GENERAL SERVICE ROUTINES⁸

More general interrupt service routines that are part of a complete interrupt-driven system must handle the following tasks:

1. **Saving any needed data in the Stack so the interrupted program can be resumed correctly.** The 6809 microprocessor saves all the user registers automatically in response to IRQ or NMI and as part of the execution of CWAI and the Software Interrupt instructions. An interrupt service routine for

$\overline{\text{FIRQ}}$ will have to save and restore any registers it uses besides the Program Counter and the Condition Code Register.

2. **Restoring data and registers (if not automatically saved) before executing RTI and returning control to the interrupted program.**
3. **Establishing the priority of the interrupt, perhaps by writing that priority into an external register.**

The program can then reenble the rest of the interrupt system. Remember, however, that to restore the old priority correctly, you must save it in the stack along with the other status. The program must save a copy of the current priority in RAM if the external priority register is write-only.

4. **Restoring the old priority before returning control to the interrupted program.**
5. **Enabling and disabling interrupts appropriately.** Remember that the CPU automatically disables $\overline{\text{IRQ}}$ after accepting an interrupt on that line and automatically disables $\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$ after accepting an interrupt on $\overline{\text{FIRQ}}$ or $\overline{\text{NMI}}$.

The service routines should be transparent as far as the interrupted program is concerned; that is, they should have no incidental effects.

Any standard subroutines that an interrupt service routine uses must be reentrant. If some subroutines cannot be made reentrant, the interrupt service routine must have separate versions to use.

PROBLEMS

15-1. A TEST INTERRUPT

Purpose: The computer waits for a PIA interrupt to occur, then executes the endless loop instruction:

HERE BRA HERE

until the next interrupt occurs.

15-2. A KEYBOARD INTERRUPT

Purpose: The computer waits for a 4-digit entry from a keyboard and places the digits into memory locations 0050 through 0053 (first one received in 0050). Each digit entry causes an interrupt. The fourth entry should also result in the disabling of the keyboard interrupt.

Sample Problem:

Keyboard data = 04, 06, 01, 07

Result: (0050) = 04
 (0051) = 06
 (0052) = 01
 (0053) = 07

15-3. A PRINTER INTERRUPT

Purpose: The computer sends four characters from memory locations 0050 through 0053 (starting with 0050) to the printer. Each character is requested by an interrupt. The fourth transfer also disables the printer interrupt.

15-4. A REAL-TIME CLOCK INTERRUPT

Purpose: The computer clears memory location 0040 initially and then complements that location each time the real-time clock interrupt occurs. How would you change the program so that it complements memory location 0040 after every ten interrupts? How would you change the program so it leaves 0040 at zero for ten clock periods, FF_{16} for five clock periods, and so on continuously? You may want to use a display rather than memory location 0041 to make it easier to see.

15-5. A TELETYPEWRITER INTERRUPT

Purpose: The computer receives TTY data from an interrupting 6850 ACIA and stores the characters in a buffer starting in memory location 0050. The process continues until the computer receives a carriage return ($0D_{16}$). Assume that the characters are 7-bit ASCII with odd parity. How would you change your program to use a PIA? Assume that subroutine TTYRCV is available, as in the example. Include the carriage return as the final character in the buffer.

REFERENCES

1. A. Osborne. *An Introduction to Microcomputers: Volume 1 — Basic Concepts*, Osborne/McGraw-Hill, Berkeley, Calif., 1980, Chapter 5.
2. R. L. Baldrige. "Interrupts Add Power, Complexity to Microcomputer Software Design," *EDN*, August 5, 1977, pp. 67-73.
3. R. Morris. "6800 Routine Supervises Service Requests," *EDN*, October 5, 1979, pp. 73-81.
4. I. P. Breikss. "Nonmaskable Interrupt Saves Processor Register Contents," *Electronics*, July 21, 1977, p. 104.
5. A. Osborne. *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, Osborne/McGraw-Hill, Berkeley, Calif., pp. 9-71 through 9-77.
6. R. Grappel. "Technique Avoids Interrupt Dangers," *EDN*, May 5, 1979, p. 88.
7. G. Horner. "Online Control of a Laboratory Instrument by a Timesharing Computer," *Computer Design*, February 1980, pp. 90-106.
8. For further discussion and some real-life examples of designing 6800-based systems with interrupts, see the following:
S. C. Baunach. "An Example of an M6800-based GPIB Interface", *EDN*, September 20, 1977, pp. 125-28.

L. E. Cannon and P. S. Kreager. "Using a Microprocessor: a Real-Life Application, Part 2 — Software," *Computer Design*, October 1975, pp. 81-89.

D. Fullager, et al. "Interfacing Data Converters and Microprocessors," *Electronics*, December 8, 1976, pp. 81-89.

S. A. Hill. "Multiprocess Control Interface Makes Remote μ P Command Possible," *EDN*, February 5, 1976, pp. 87-89.

W. S. Holderby. "Designing a Microprocessor-based Terminal for Factory Data Collection," *Computer Design*, March 1977, pp. 81-88.

A. Lange. "OPTACON Interface permits the Blind to 'Read' Digital Instruments," *EDN*, February 5, 1976, pp. 84-86.

J. D. Logan and P. S. Kreager. "Using a Microprocessor: a Real-Life Application, Part 1 — Hardware," *Computer Design*, September 1975, pp. 69-77.

A. Moore and M. Eidson. "Printer Control," Application Note available from Motorola Semiconductor Products, Phoenix, Ariz.

M. C. Mulder and P. P. Fasang. "A Microprocessor Controlled Substation Alarm Logger," IECI '78 Proceedings — Industrial Applications of Microprocessors, March 20-22, 1978, pp. 2-6.

P. J. Zsombar-Murray et al. "Microprocessor Based Frequency Response Analyzer," IECI '78 Proceedings — Industrial Applications of Microprocessors, March 20-22, 1978, pp. 36-44.

The Proceedings of the IEEE's Industrial Electronics and Control Instrumentation Group's Annual Meeting on "Industrial Applications of Microprocessors" contain many interesting articles. Volumes (starting with 1975) are available from IEEE Service Center, CP Department, 445 Hoes Lane, Piscataway, N. J. 08854.



IV

Software Development

The previous chapters have described how to write short assembly language programs. While this is an important topic, it is only a small part of software development. Although writing assembly language programs is a major task for the beginner, it soon becomes simple. By now you should be familiar with standard methods for programming in assembly language on the 6809 microprocessor. **The next six chapters will describe how to formulate tasks as programs and how to combine short programs to form a working system.**

THE STAGES OF SOFTWARE DEVELOPMENT

Software development consists of many stages. **Figure IV-1 is a flowchart of the software development process. Its stages are:**

- **Problem definition**
- **Program design**
- **Coding**
- **Debugging**
- **Testing**
- **Documentation**
- **Maintenance and redesign**

Each of these stages is important in the construction of a working system. Coding, the writing of programs in a form that the computer understands, is only one stage in a long process.

IV-2. 6809 Assembly Language Programming

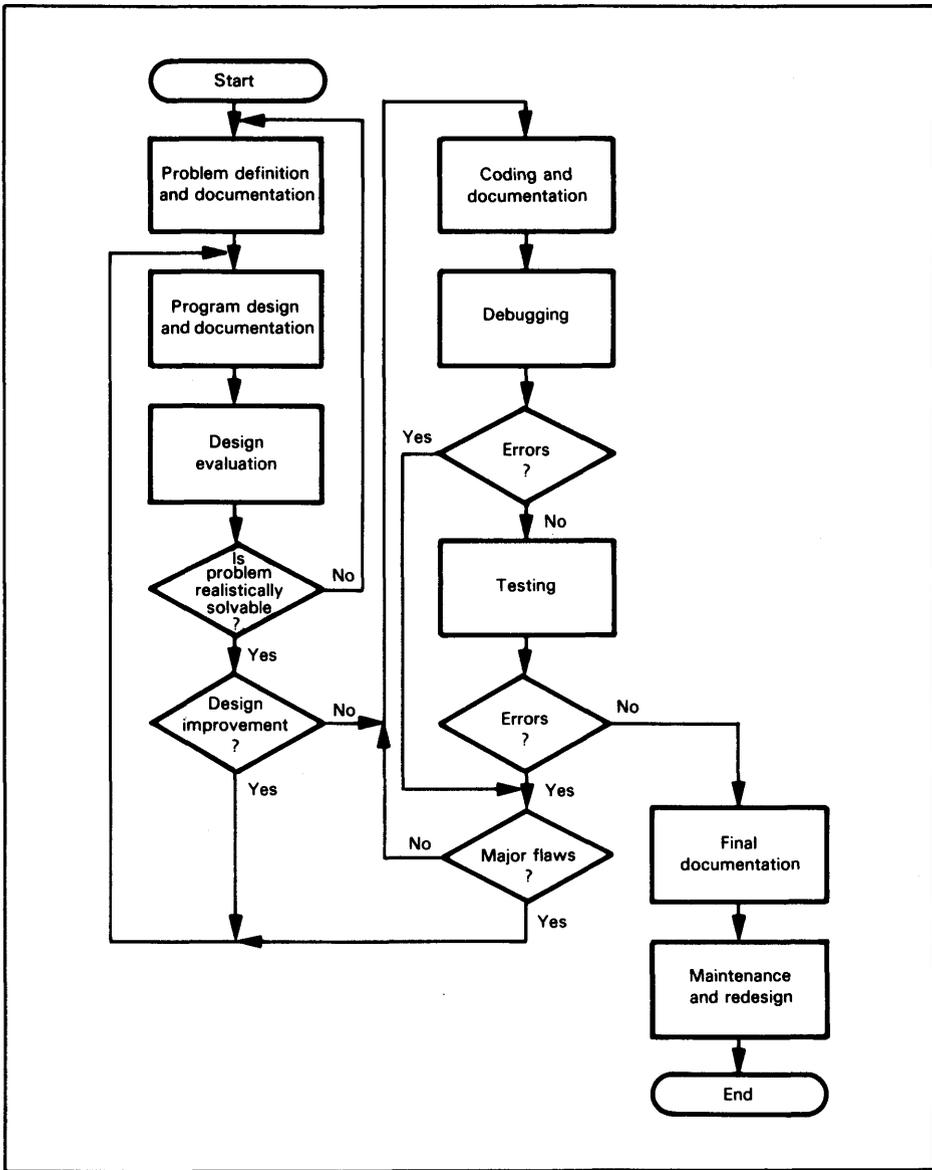


Figure IV-1. Flowchart of Software Development

RELATIVE IMPORTANCE OF CODING

Coding is usually the easiest stage to define and perform. The rules for writing computer programs are easy to learn. They vary somewhat from computer to computer, but the basic techniques remain the same. Few software projects run into trouble because of coding; indeed, coding is not the most time-consuming part of software development. Experts estimate that a programmer can write one to ten fully debugged and documented statements per day. Clearly, the mere coding of one to ten statements is hardly a full day's effort. On most software projects, coding occupies less than 25% of the programmer's time.

MEASURING PROGRESS IN OTHER STAGES

Measuring progress in other stages is difficult. You can say that half of the program has been written, but you can hardly say that half of the errors have been removed or half of the problem has been defined. Timetables for such stages as program design, debugging, and testing are difficult to produce. Many days or weeks of effort may result in no clear progress. Furthermore, an incomplete job in one stage may result in tremendous problems later. For example, poor problem definition or program design can make debugging and testing very difficult. Time saved in one stage may be spent many times over in later stages.

DEFINITION OF THE STAGES

Problem Definition

Problem definition is the formulation of the requirements that the task places on the computer. For example, what is necessary to make a computer control a tool, run a series of electrical tests, or handle communications between a central controller and a remote instrument? Problem definition requires that you determine the forms and rates of inputs and outputs, the amount and speed of processing that is needed, and the types of possible errors and their handling. Problem definition takes a vague idea of building a computer-controlled system and defines the tasks and requirements for the computer.

Program Design

Program design is the outline of the computer program that will meet the requirements. In the design stage, the tasks are described in a way that can easily be converted into a program. **Among the useful techniques in this stage are flowcharting, structured programming, modular programming, and top-down design.**

Coding

Coding is the writing of the program in a form that the computer can either directly understand or translate. The form may be machine language, assembly language, or a high-level language.

Debugging

Debugging, also called program verification, is **making the program perform according to the design**. In this stage, you use such tools as breakpoints, traces, simulators, logic analyzers, and in-circuit emulators. The end of the debugging stage is hard to define, since you never know when you have found the last error.

Testing

Testing, also referred to as program validation, is **ensuring that the program performs the overall system tasks correctly**. The designer uses simulators, exercisers, and statistical techniques to measure the program's performance. This stage is like quality control for hardware.

Documentation

Documentation is the description of the program in the proper form for users and maintenance personnel. Documentation also allows the designer to develop a program library so that subsequent tasks will be far simpler. Flowcharts, comments, memory maps, and library forms are some of the tools used in documentation.

Maintenance and Redesign

Maintenance and redesign are the servicing, improvement, and extension of the program. Clearly, the designer must be ready to handle field problems in computer-based equipment. Special diagnostic modes or programs and other maintenance tools may be required. Upgrading or extension of the program may be necessary to meet new requirements or handle new tasks.

16

Problem Definition

Typical microprocessor tasks require a lot of definition. For example, what must a program do to control a scale, a cash register, or a signal generator? Clearly, we have a long way to go just to define the tasks involved.

INPUTS

How do we start the definition? The obvious place to begin is with the inputs. We should begin by listing all the inputs that the computer may receive in this application.

Examples of inputs are:

- Data blocks from transmission lines
- Status words from peripherals
- Data from A/D converters

Then we may ask the following questions about each input:

1. What is its form; that is, what signals will the computer actually receive?
2. When is the input available and how does the processor know it is available? Does the processor have to request the input with a strobe signal? Does the input provide its own clock?
3. How long is it available?

4. How often does it change, and how does the processor know that it has changed?
5. Does the input consist of a sequence or block of data? Is the order important?
6. What should be done if the data contains errors? These may include transmission errors, incorrect data, sequencing errors, extra data, etc.
7. Is the input related to other inputs or outputs?

OUTPUTS

The next step to define is the output. **We must list all the outputs that the computer must produce.** Examples of outputs include:

- Data blocks to transmission lines
- Control words to peripherals
- Data to D/A converters

Then we may ask the following questions about each output:

1. What is its form; that is, what signals must the computer produce?
2. When must it be available, and how does the peripheral know it is available?
3. How long must it be available?
4. How often must it change, and how does the peripheral know that it has changed?
5. Is there a sequence of outputs?
6. What should be done to avoid transmission errors or to sense and recover from peripheral failures?
7. How is the output related to other inputs and outputs?

PROCESSING SECTION

Between the reading of input data and the sending of output results is the processing section. **Here we must determine exactly how the computer must process the input data. The questions are:**

1. What is the basic procedure (algorithm) for transforming input data into output results?
2. What time constraints exist? These may include data rates.
3. What memory constraints exist? Do we have limits on the amount of program memory or data memory, or on the size of buffers?
4. What standard programs or tables must be used? What are their requirements?
5. What special cases exist, and how should the program handle them?
6. How accurate must the results be?
7. How should the program handle processing errors or special conditions such as overflow, underflow, or loss of significance?

ERROR HANDLING

An important factor in many applications is the handling of errors. Clearly, the designer must make provisions for recovering from common errors and for diagnosing malfunctions. **Among the questions that the designer must ask at the definition stage are:**

1. What errors could occur?
2. Which errors are most likely? If a person operates the system, human error is the most common. Following human errors, communications or transmission errors are more common than mechanical, electrical, mathematical, or processor errors.
3. Which errors will not be immediately obvious to the system? A special problem is the occurrence of errors that the system or operator may not recognize as incorrect.
4. How can the system recover from errors with a minimum loss of time and data and yet be aware that an error has occurred?
5. Which errors or malfunctions cause the same system behavior? How can these errors or malfunctions be distinguished for diagnostic purposes?
6. Which errors involve special system procedures? For example, do parity errors require retransmission of data?

Another question is: How can the field technician systematically find the source of malfunctions without being an expert? Built-in test programs, special diagnostics, or signature analysis can help.¹

HUMAN FACTORS/OPERATOR INTERACTION

Many microprocessor-based systems involve human interaction. **Human factors must be considered throughout the development process for such systems. Among the questions that the designer must ask are:**

1. What input procedures are most natural for the human operator?
2. Can the operator easily determine how to begin, continue and end the input operations?
3. How is the operator informed of procedural errors and equipment malfunctions?
4. What errors is the operator most likely to make?
5. How does the operator know that data has been entered correctly?
6. Are displays in a form that the operator can easily read and understand?
7. Is the response of the system adequate for the operator?
8. Is the system easy for the operator to use?
9. Are there guiding features for an inexperienced operator?
10. Are there shortcuts and reasonable options for the experienced operator?
11. Can the operator always determine or reset the state of the system after interruptions or distractions?

Building a system for people to use is difficult. The microprocessor can make the system more powerful, more flexible, and more responsive. However, **the designer still must add the human touches that can greatly increase the usefulness and attractiveness of the system and the productivity of the human operator.**²

The processor, of course, has no intrinsic preference in situations involving human characteristics or cultural choices. The processor does not prefer left-to-right over right-to-left, forward over backward, increasing order over decreasing order, or decimal numbers over other number systems. Nor does the processor recognize the operator's preference for simplicity, consistency, compatibility with previous experience, and "logical" order of operations. The processor never gets distracted, disoriented, confused, or bored. The designer must allow for all these considerations in the design and development of interactive systems.

EXAMPLES

DEFINING A SWITCH AND LIGHT SYSTEM

Figure 16-1 shows a simple system in which the input is from a single SPST switch and the output is to a single LED display. In response to a switch closure, the processor turns the display on for one second. This system should be easy to define.

Switch Input

Let us first examine the input and answer each of the questions previously presented:

1. The input is a single bit, which may be either '0' (switch closed) or '1' (switch open).
2. The input is always available and need not be requested.
3. The input is available for at least several milliseconds after the closure.
4. The input will seldom change more than once every few seconds. The processor has to handle only the bounce in the switch. The processor must monitor the switch to determine when it is closed.
5. There is no sequence of inputs.
6. The obvious input errors are switch failure, failure in the input circuitry, and the operator attempting to close the switch again before a sufficient amount of time has elapsed. We will discuss the handling of these errors later.
7. The input does not depend on any other inputs or outputs.

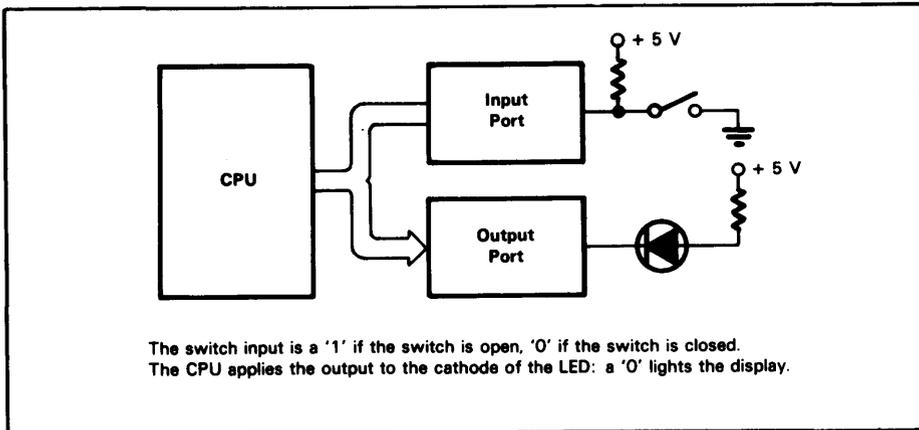


Figure 16-1. The Switch and Light System

Light Output

The next requirement in defining the system is to examine the output. The answers to our questions are:

1. The output is a single bit, which is '0' to turn the display on, '1' to turn it off.
2. There are no time constraints on the output. The peripheral does not need to be informed of the availability of data.
3. If the display is an LED, the data need be available for only a few milliseconds at a pulse rate of about 100 times per second. The observer will see a continuously lit display.
4. The data must change (go off) after one second.
5. There is no sequence of outputs.
6. The possible output errors are display failure and failure in the output circuitry.
7. The output depends only on the switch input and time.

Processing

The processing section is extremely simple. As soon as the switch input becomes a logic '0', the CPU turns the light on (a logic '0') for one second. No time or memory constraints exist.

Error Handling

Let us now look at the possible errors and malfunctions. These are:

- Another switch closure before one second has elapsed
- Switch failure
- Display failure
- Computer failure

Surely the first error is the most likely. The simplest solution is for the processor to ignore switch closures until one second has elapsed. This brief unresponsive period will hardly be noticeable to the human operator. Furthermore, ignoring the switch during this period means that no debouncing circuitry or software is necessary, since the system will not react to the bounce anyway.

Clearly, the last three failures can produce unpredictable results. The display may stay on, stay off, or change state randomly. Some possible ways to isolate the failures would be:

- Lamp-test hardware to check the display; i.e., a button that turns the light on independently of the processor
- A direct connection to the switch to check its operation
- A diagnostic program that exercises the input and output circuits

If both the display and switch are working, the computer is at fault. A field technician with proper equipment can determine the cause of the failure.

DEFINING A SWITCH-BASED MEMORY LOADER

Figure 16-2 shows a system that allows the user to enter data into any memory location in a microcomputer. One input port, DPORT, reads data from eight toggle switches. The other input port, CPORT, is used to read control information. There are three momentary switches: High Address, Low Address and Data. The output is the value of the last completed entry from the data switches; eight LEDs are used for the display.

The system will also, of course, require resistors, buffers, and drivers.

Inputs

The characteristics of the switches are the same as in the previous example. To simplify the debouncing procedure and force the operator to release the buttons, we have the system respond only after a button is released; this is a common technique that reduces wear on the switches as well, since the operator is less tempted to press a button repeatedly. In this system there is a distinct sequence of inputs, as follows:

1. The operator must set the data switches according to the eight most significant bits of an address, then
2. press and release the High Address button. The high address bits will appear on the lights, and the program will interpret the data as the high byte of the address.
3. Then the operator must set the data switches with the value of the least significant byte of the address and
4. press and release the low Address button. The low address bits will appear on the lights, and the program will consider the data to be the low byte of the address.
5. Finally, the operator must set the desired data into the data switches and
6. press and release the Data button. The display will now show the data, and the program stores the data in memory at the previously entered address.

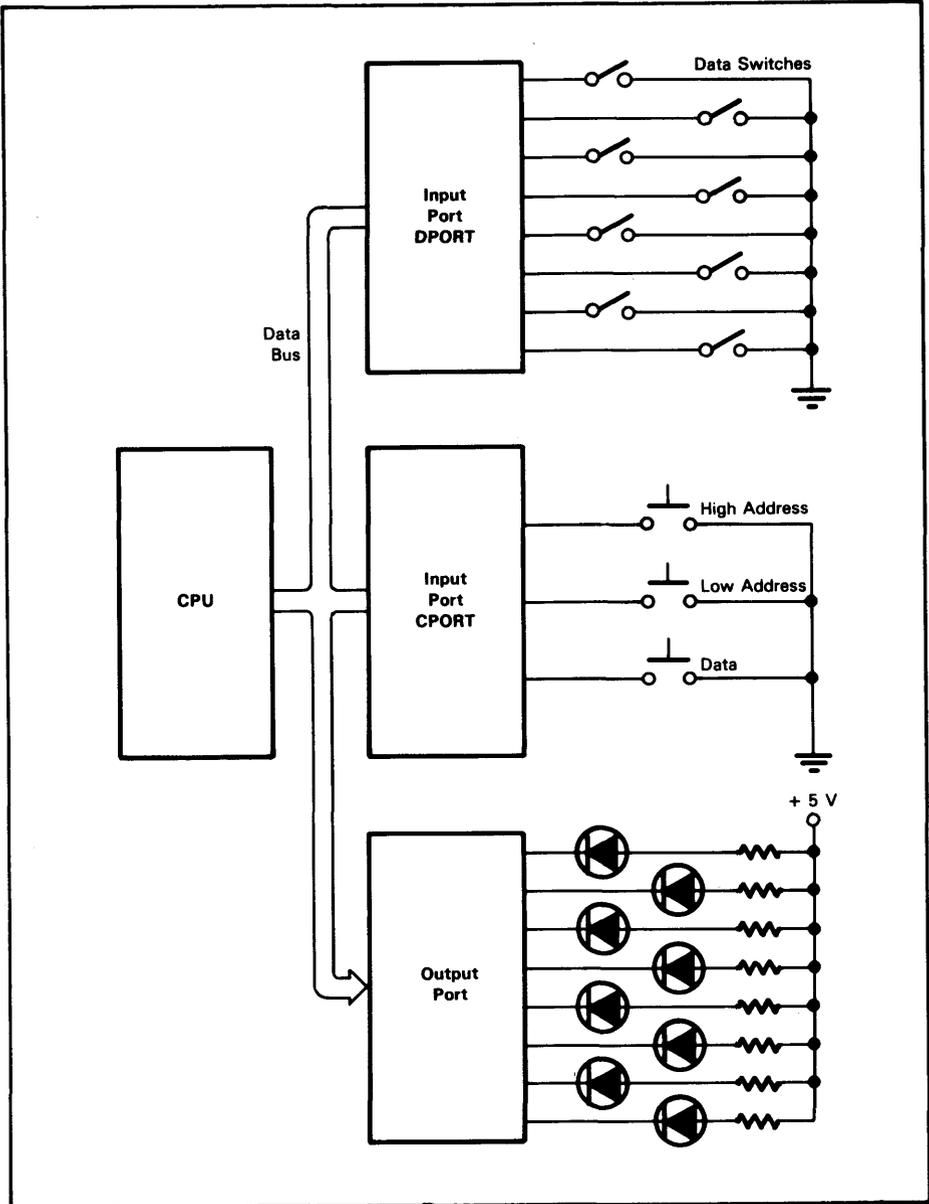


Figure 16-2. The Switch-Based Memory Loader

The operator may repeat the process to enter an entire program. Clearly, even in this simplified situation, we will have many possible sequences to consider. How do we cope with erroneous sequences and make the system easy to use?

Output

Output is no problem. After each input, the program sends to the displays the complement (since the displays are active-low) of the input bits. The output data remains the same until the next input operation.

Processing

The processing section remains quite simple. There are no time or memory constraints. The program can debounce the switches by waiting for a few milliseconds, and must provide complemented data to the displays.

Error Handling

The most likely errors are operator mistakes. These include:

- Incorrect entries
- Incorrect order
- Incomplete entries; for example, forgetting the data

The system must be able to handle these problems in a reasonable way, since they are certain to occur in actual operation.

The designer must also consider the effects of equipment failure. Just as before, the possible difficulties are:

- Switch failure
- Display failure
- Computer failure

In this system, however, we must pay more attention to how these failures affect the system. A computer failure will cause a complete system breakdown that will be easy to detect. A display failure may not be immediately noticeable; here a Lamp Test feature will allow the operator to check the operation. Note that we would like to test each LED separately, in order to diagnose the case in which output lines are shorted together. In addition, the operator may not immediately detect switch failure; however, the operator should soon notice it and establish which switch is faulty by a process of elimination.

Operator Error Correction

Let us look at **some of the possible operator errors.** Typical errors will be:

- Erroneous data
- Wrong order of entries or switches
- Trying to go on to the next entry without completing the current one

The operator will presumably notice erroneous data as soon as it appears on the displays. **What is a viable recovery procedure? Some options are:**

1. The operator must complete the entry procedure; i.e., enter Low Address and Data if the error occurs in the High Address. Clearly, this procedure is wasteful and annoying.
2. The operator may restart the entry process by returning to the high address entry steps. This solution is useful if the error was in the High Address, but forces the operator to re-enter earlier data if the error was in the Low Address or Data stage.
3. The operator may enter any part of the sequence at any time simply by setting the Data switches with the desired data and pressing the corresponding button. This procedure allows the operator to make corrections at any point in the sequence.

This type of procedure should always be preferred over one that does not allow immediate error correction, has a variety of concluding steps, or enters data into the system without allowing the operator a final check. Any added complication in hardware or software will be justified in increased operator efficiency. You should always prefer to let the microcomputer do the tedious work and recognize arbitrary sequences; it never gets tired and never forgets the operating procedures.

A further helpful feature would be status lights that would define the meaning of the display. Three status lights, marked "High Address," "Low Address," and "Data," would let the operator know what had been entered without having to remember which button was pressed. The processor would have to monitor the sequence, but the added complication in software would simplify the operator's task. Clearly, three separate sets of displays plus the ability to examine a memory location would be even more helpful to the operator.

We should note that, although we have emphasized human interaction, machine or system interaction has many of the same characteristics. The microprocessor should do the work. If complicating the microprocessor's task makes error recovery simple and the causes of failure obvious, the entire system will work better and be easier to maintain. Note that you should not wait until after the software has been completed to consider system use and maintenance; instead, you should include these factors in the problem definition stage.

DEFINING A VERIFICATION TERMINAL

Figure 16-3 is a block diagram of a simple credit-verification terminal. One input port derives data from a keyboard (see Figure 16-4); the other input port accepts verification data from a transmission line. One output port sends data to a set of displays (see Figure 16-5); another sends the credit card number to the central computer. A third output port turns on one light whenever the terminal is ready to accept an inquiry, and another light when the operator sends the information. The "busy" light is turned off when the terminal receives a response. Clearly, the input and output of data will be more complex than in the previous case, although the processing is still simple.

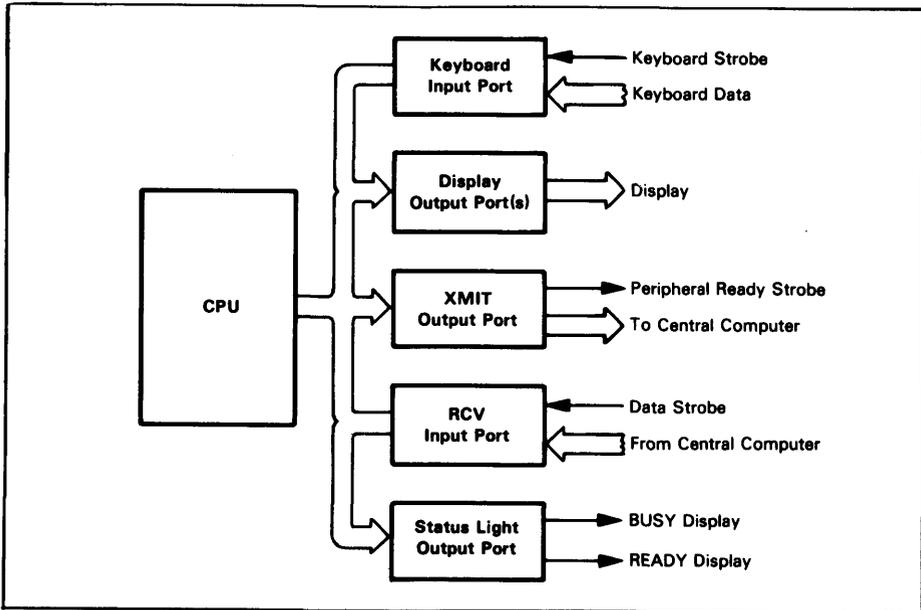


Figure 16-3. Block Diagram of a Verification Terminal

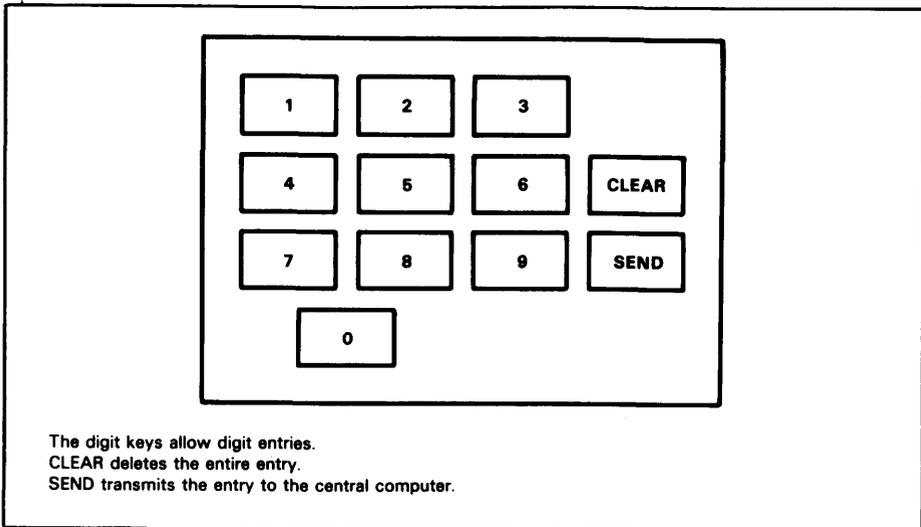


Figure 16-4. Verification Terminal Keyboard

Additional displays may be useful to emphasize the meaning of the response. Many terminals use a green light for “Yes,” a red light for “No,” and a yellow light for “Consult Store Manager.” Note that these lights will still have to be clearly marked with their meanings to allow for a color-blind operator.

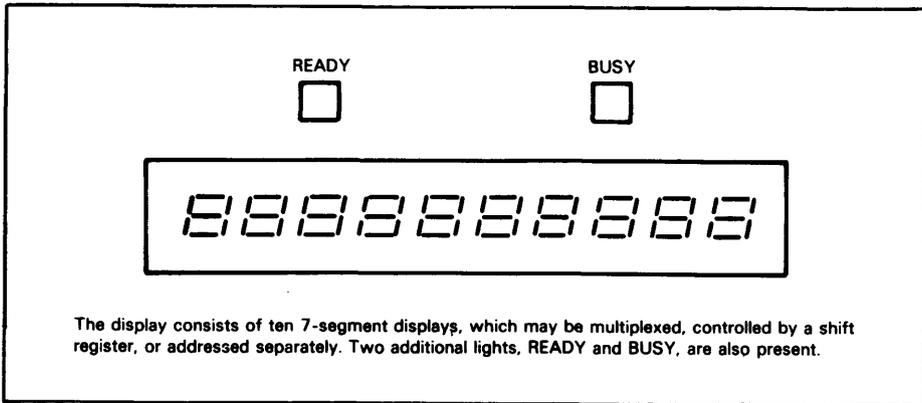


Figure 16-5. Verification Terminal Display

Inputs

Let us first look at the keyboard input. This is, of course, different from the switch input, since the CPU must have some way of distinguishing new data. We will assume that each key closure provides a unique hexadecimal code (we can code each of the 12 keys into one digit) and a strobe. The program will have to recognize the strobe and fetch the hexadecimal number that identifies the key. There is a time constraint, since the program cannot miss any data or strobes. The constraint is not serious, since keyboard entries will be at least several milliseconds apart.

The transmission input similarly consists of a series of characters, each identified by a strobe (perhaps from a UART). The program will have to recognize each strobe and fetch the character. The data being sent across the transmission lines is usually organized into messages. A possible message format is:

- Introductory characters, or header
- Terminal destination address
- Coded yes or no
- Ending characters, or trailer

The terminal will check the header, read the destination address, and see if the message is intended for it. If the message is for the terminal, the terminal accepts the data. The address could be (and often is) hard-wired into the terminal so that the terminal receives only messages intended for it. This approach simplifies the software at the cost of some flexibility.

Outputs

The output is also more complex than in the earlier examples. If the displays are multiplexed, the processor must not only send the data to the display port but must also direct the data to a particular display. We will need either a separate control port or a counter and decoder to handle this. Note that hardware blanking controls can blank leading zeros as long as the first digit in a multi-digit number is never zero. Software can also handle this task. Time constraints include the pulse length and frequency required to produce a continuous display for the operator.

The communications output will consist of a series of characters with a particular format. The program will also have to consider the time required between characters. A possible format for the output message is:

- Header
- Terminal address
- Credit card number
- Trailer

A central communication computer may poll the terminals, checking for data ready to be sent.

Processing

The processing in this system involves many new tasks, such as:

- Identifying the control keys by number and performing the proper actions
- Adding the header, terminal address, and trailer to the outgoing message
- Recognizing the header and trailer in the returning message
- Checking the incoming terminal address

Note that none of the tasks involves any complex arithmetic or any serious time or memory constraints.

Error Handling

The number of possible errors in this system is, of course, much larger than in the earlier examples. Let us first consider the possible operator errors. These include:

- Entering the credit card number incorrectly
- Trying to send an incomplete credit card number
- Trying to send another number while the central computer is processing one
- Clearing nonexistent entries

Some of these errors can be handled easily by organizing the program correctly. For example, the program should not accept the Send key until the credit card number has been completely entered, and it should ignore any additional keyboard entries until the response comes back from the central computer. Note that the operator will know that the entry has not been sent, since the Busy light will not go on. The operator will also know when the keyboard has been locked out (the program is ignoring keyboard entries), since entries will not appear on the display and the Ready light will be off.

Correcting Keyboard Errors

Incorrect entries are an obvious problem. **If the operator recognizes an error, he or she can use the Clear key to make corrections. The operator would probably find it more convenient to have two Clear keys, one that cleared the most recent key and one that cleared the entire entry.** This would allow both for the situation in which the operator recognizes the error immediately and for the situation in which the operator

recognizes the error late in the procedure. **The operator should be able to correct errors immediately and have to repeat as few keys as possible. The operator will, however, make a certain number of errors without recognizing them. Most credit card numbers include a self-checking digit; the terminal could check the number before permitting it to be sent to the central computer.** This step would save the central computer from wasting processing time checking the number.

This requires, however, that the terminal have some way of informing the operator of the error, perhaps by flashing one of the displays or by providing some other special indicator that the operator is sure to notice.

Still another problem is how the operator knows that an entry has been lost or processed incorrectly. Some terminals simply unlock after a maximum time delay. The operator notes that the Busy light has gone off without an answer being received. The operator is then expected to try the entry again. After one or two further attempts, the operator should report the failure to supervisory personnel.

Many equipment failures are also possible. Besides the displays, keyboard, and processor, there now exist the problems of communications errors or failures and central computer failures.

Correcting Transmission Errors

The data transmission will probably have to include error checking and correcting procedures. Some possibilities are:

1. Parity provides an error detection facility but no correction mechanism. The receiver will need some way of requesting retransmission, and the sender will have to save a copy of the data until proper reception is acknowledged. Parity is, however, very simple to implement.
2. Short messages may use more elaborate schemes. For example, the yes/no response to the terminal could be coded to provide error detection and correction capability.
3. An acknowledgement and a limited number of retries could trigger an indicator that would inform the operator of a communications failure (inability to transfer a message without errors) or central computer failure (no response within a certain period of time). Such a scheme, along with the Lamp Test, would allow simple failure diagnosis.

A communications or central computer failure indicator should also "unlock" the terminal, that is, allow it to accept another entry. This is necessary if the terminal will not accept entries while a verification is in progress. **The terminal may also unlock after a certain maximum time delay. Certain entries could be reserved for diagnostics;** i.e., certain credit card numbers could be used to check the internal operation of the terminal and test the displays.

REVIEW

Problem definition is as important a part of software development as it is of any other engineering task. Note that it does not require any programming or knowledge of the computer; rather, it is based on an understanding of the system and sound engineering judgment. Microprocessors offer flexibility and local intelligence that the designer can use to provide a wide range of features.

Problem definition is independent of any particular computer, computer language, or development system. It should, however, provide guidelines as to what type or speed of computer the application will require and what kind of hardware/software tradeoffs the designer can make. The problem definition stage should not even depend on whether a computer is used, although a knowledge of the capabilities of the computer can help the designer in suggesting possible implementations of procedures.

REFERENCES

1. D. R. Ballard. "Designing Fail-Safe Microprocessor Systems," *Electronics*, January 4, 1979, pp. 139-43.
"A Designer's Guide to Signature Analysis," Hewlett-Packard Application Note 222, Hewlett-Packard, Inc, Palo Alto, CA, 1977.
Donn, E. S. and M. D. Lippman. "Efficient and Effective Microcomputer Testing Requires Careful Preplanning," *EDN*, February 20, 1979, pp. 97-107 (includes self-test examples for 6502).
Gordon, G. and H. Nadig. "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems," *Electronics*, March 3, 1977, pp. 89-96.
Neil, M. and R. Goodner. "Designing a Serviceman's Needs into Microprocessor-Based Systems," *Electronics*, March 1, 1979, pp. 122-28.
Schweber, W. and L. Pearce. "Software Signature Analysis Identifies and Checks PROMs," *EDN*, November 5, 1978, pp. 79-81.
Srini, V. P. "Fault Diagnosis of Microprocessor Systems," *Computer*, January 1977, pp. 60-65.
2. For a brief discussion of human factors considerations, see G. Morris. "Make Your Next Instrument Design Emphasize User Needs and Wants," *EDN*, October 20, 1978, pp. 100-05.

17

Program Design

Program design is the stage in which the problem definition is formulated as a program. If the program is small and simple, this stage may involve little more than the writing of a one-page flowchart. If the program is larger or more complex, the designer should consider more elaborate methods.

We will discuss flowcharting, modular programming, structured programming, and top-down design. We will try to indicate the reasoning behind these methods, and their advantages and disadvantages. We will not, however, advocate any particular method since there is no evidence that one method is always superior to all others. You should remember that the goal is to produce a good working system, not to follow religiously the tenets of one methodology or another.

BASIC PRINCIPLES

All the methodologies are based on common principles, many of which apply to any kind of design. Among these principles are:

- 1. Proceed in small steps.** Do not try to do too much at one time.
- 2. Divide large jobs into small, logically separate tasks.** Make the sub-tasks as independent of one another as possible, so that they can be tested separately and so that changes can be made in one without affecting the others.
- 3. Keep the flow of control simple** to make programs easy to follow and errors easy to locate and correct.
- 4. Use pictorial or graphic design descriptions** as much as possible. They are easier to visualize than word descriptions. This is the great advantage of flowcharts.

5. **Emphasize clarity and simplicity at first.** You can improve performance (if necessary) once the system is working.
6. **Proceed in a thorough and systematic manner.** Use checklists and standard procedures.
7. Do not tempt fate. **Either do not use methods that you are not sure of, or use them very carefully. Watch for situations that might cause confusion, and clarify them as soon as possible.**
8. Keep in mind that **the system must be debugged, tested and maintained. Plan for these later stages.**
9. **Use simple and consistent terminology and methods.** Repetitiveness is no fault in program design, nor is complexity a virtue.
10. **Have your design completely formulated before you start coding.** Resist the temptation to start writing down instructions: it makes no more sense than making parts lists or laying out circuit boards before you know exactly what will be in the system.
11. Be particularly careful of factors that may change. **Make the implementation of likely changes as simple as possible.**
12. **Keep the overall task in mind.** Build a total framework in which individual pieces can be defined and tested. Do not leave the entire system integration to the end.
13. If the data is complex or there are numerous relationships between data items, **you must organize your data just as carefully as you organize your program. We will briefly discuss the design of data structures at the end of this chapter.**

FLOWCHARTING

Flowcharting is certainly the best-known of all program design methods. Programming textbooks describe how programmers first write complete flowcharts and then start writing the actual program. In fact, few programmers have ever worked this way, and flowcharting has often been more of a joke or a nuisance to programmers than a design method. We will try to describe both the advantages and disadvantages of flowcharts, and show the place of this technique in program design.

ADVANTAGES OF FLOWCHARTING

The basic advantage of the flowchart is that it is a pictorial representation. People find such representations much more meaningful than written descriptions. The designer can visualize the whole system and see the relationships of the various parts. Logical errors and inconsistencies often stand out instead of being hidden in a printed page. At its best, the flowchart is a picture of the entire system.

Some specific advantages of flowcharts are:

1. Standard symbols exist (see Figure 17-1) so that flowcharting forms are widely recognized.

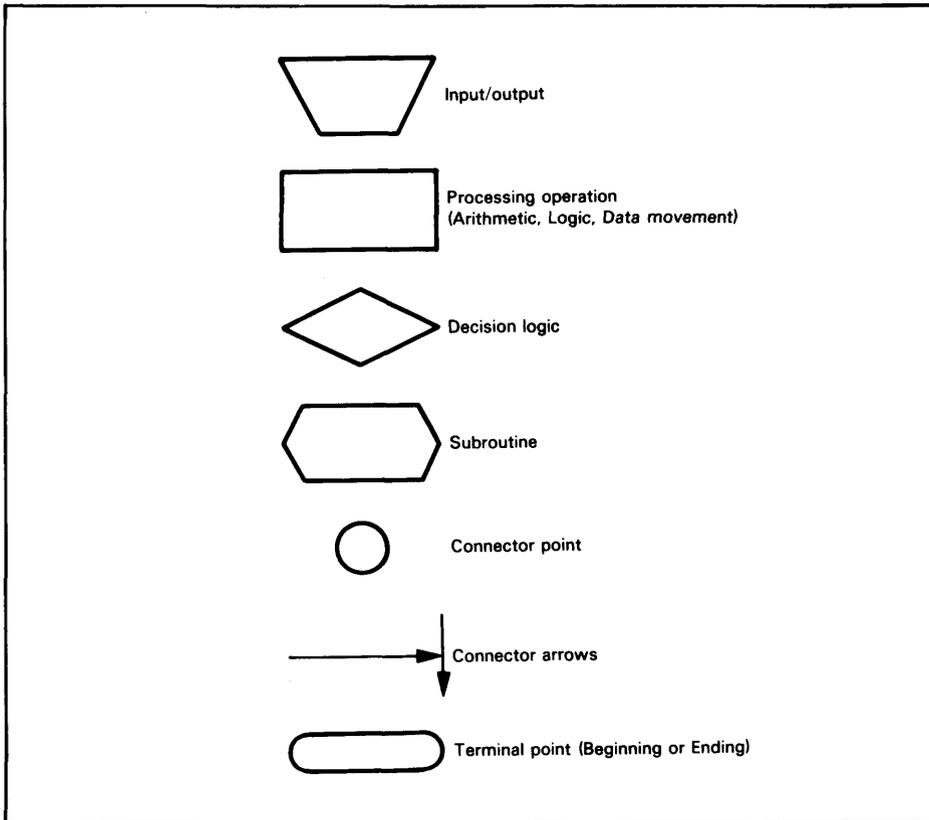


Figure 17-1. Standard Flowchart Symbols

2. Flowcharts can be understood by someone without a programming background.
3. Flowcharts can be used to divide the entire project into sub-tasks. The flowchart can then be examined to measure overall progress.
4. Flowcharts show the sequence of operations and can therefore aid in locating the source of errors.
5. Flowcharting is widely used in other areas besides programming.
6. There are many tools available to aid in flowcharting, including programmer's templates and automated drawing packages.

DISADVANTAGES OF FLOWCHARTING

These advantages are all important. There is no question that flowcharting will continue to be widely used. But we should note some disadvantages of flowcharting as a program design method:

1. Flowcharts are difficult to design, draw, or change in all except the simplest situations.

2. There is no easy way to debug or test a flowchart.
3. Flowcharts tend to become cluttered. Designers find it difficult to balance between the amount of detail needed to make the flowchart useful and the amount that makes the flowchart little better than a program listing.
4. Flowcharts show only the program organization. They do not show the organization of the data or the structure of the input/output modules.
5. Flowcharts do not help with hardware or timing problems or give hints as to where these problems might occur.
6. Flowcharts allow unstructured design. There are no rules governing the numbers of entries and exits, the number or type of interconnections, or the logic that may be employed.
7. There is no obvious way to represent the simple repetition of a loop.

MAKING FLOWCHARTS USEFUL

The most useful flowcharts may ignore program variables and ask questions directly. Of course, compromises are often necessary here. **Two versions of the flowchart are sometimes helpful — one general version in layman's language, which will be useful to non-programmers, and one programmer's version in terms of the program variables, which will be useful to other programmers.**

A third type of flowchart, a data flowchart, may also be helpful. This flowchart serves as a cross-reference for the other flowcharts, since it shows how the program handles a particular type of data. Ordinary flowcharts show how the program proceeds, handling different types of data at different points. **Data flowcharts, on the other hand, show how particular types of data move through the system, passing from one part of the program to another.** Such flowcharts are very useful in debugging and maintenance, since errors most often show up as a particular type of data being handled incorrectly.

Thus flowcharting is a helpful technique that you should not try to extend too far. Flowcharts are useful as program documentation, since they have standard forms and are comprehensible to non-programmers. As a design tool, however, flowcharts cannot provide much more than a starting outline; the programmer cannot debug a detailed flowchart and the flowchart is often more difficult to design than the program itself.

EXAMPLES

Flowcharting the Switch and Light System

This simple task, in which a single switch turns on a light for one second, is easy to flowchart. In fact, such tasks are typical examples for flowcharting books, although they form a small part of most systems. The data structure here is so simple that it can be safely ignored.

Figure 17-2 is the flowchart. There is little difficulty in deciding on the amount of detail required. The flowchart gives a straightforward picture of the procedure, which anyone could understand.

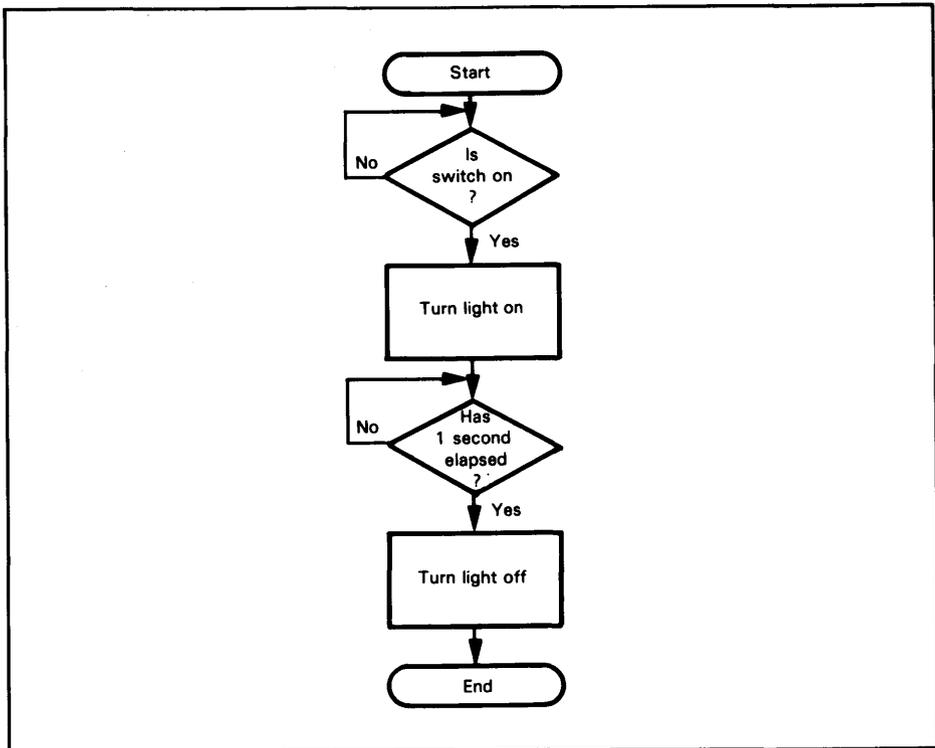


Figure 17-2. Flowchart of One-Second Response to a Switch

Flowcharting the Switch-Based Memory Loader

This system (see Figure 16-2) is considerably more complex than the previous example, and involves many more decisions. **The flowchart (see Figure 17-3) is more difficult to draw and is not as straightforward as the previous example.** In this example, we face the problem that there is no way to debug or test the flowchart.

The flowchart in Figure 17-3 includes the improvements we suggested as part of the problem definition. Clearly, this flowchart is beginning to get cluttered and lose its advantages over a written description. Adding other features that define the meaning of the entry with status lights and allow the operator to check entries after completion would make the flowchart even more complex. Drawing the complete flowchart from scratch could quickly become a formidable task. However, once the program has been written, the flowchart is useful as documentation.

Flowcharting the Verification Terminal

In this application (see Figures 16-3 through 16-5) the flowchart will be even more complex than in the switch-based memory loader case. Here, the best idea is to flowchart sections separately so that the flowcharts remain manageable. However, the presence of data structures (as in the multi-digit display and the messages) will make the gap between flowchart and program much wider.

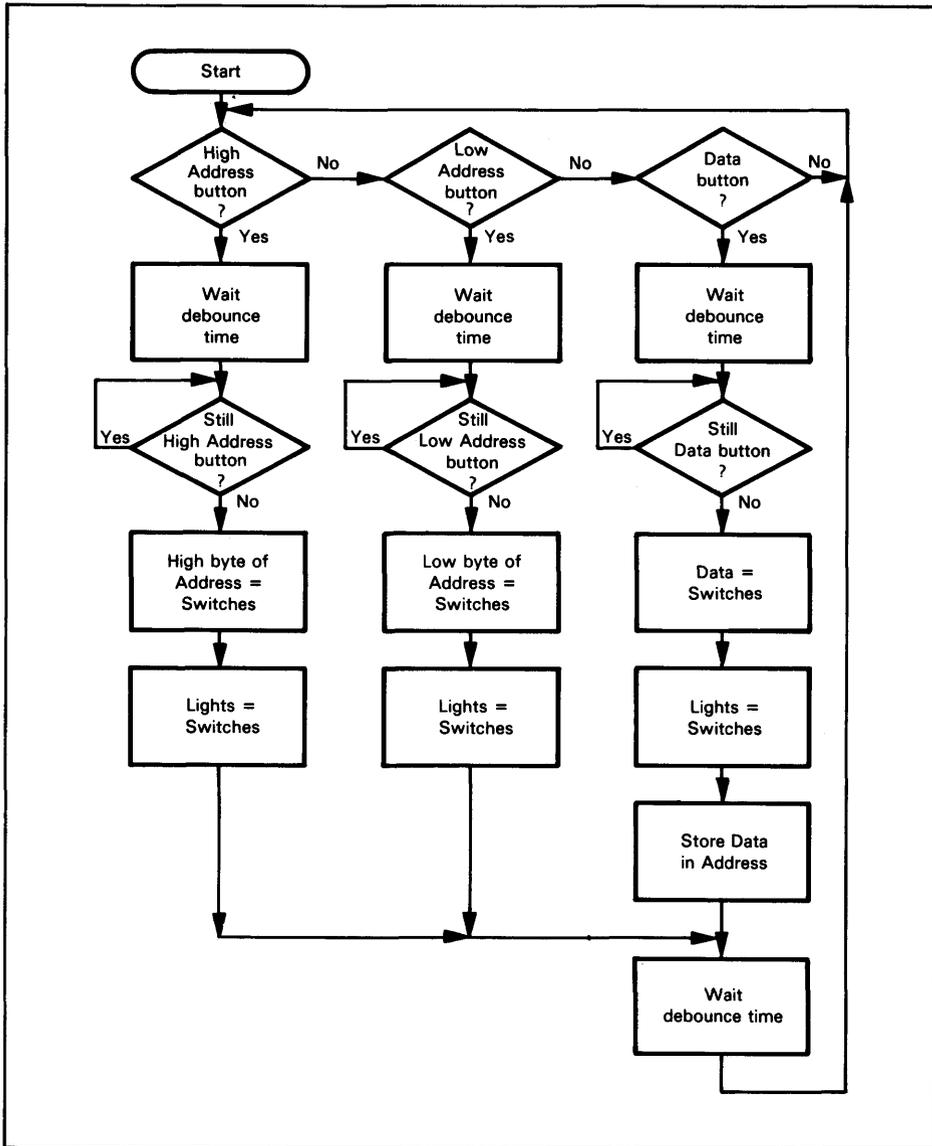


Figure 17-3. Flowchart of Switch-Based Memory Loader

Let us look at some of the sections. **Figure 17-4 shows the keyboard entry process for the digit keys.** The program must fetch the data after each strobe and place the digit into the display array if there is room for it. If there are already ten digits in the array, the program simply ignores the entry.

The actual program will have to handle the displays at the same time. Note that either software or hardware must de-activate the keyboard strobe after the processor reads a digit.

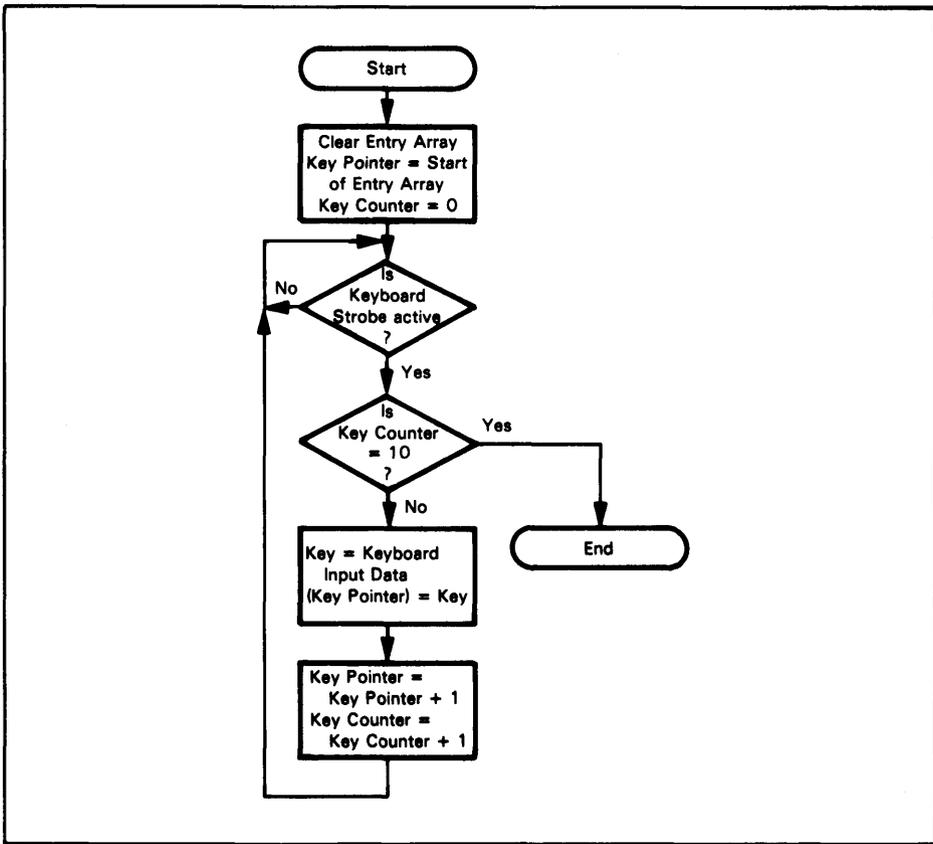


Figure 17-4. Flowchart of Keyboard Entry Process

Figure 17-5 adds the Send key. This key, of course, is optional. The terminal could just send the data as soon as the operator enters a complete number. However, that procedure would not give the operator a chance to check the entire entry. The flowchart with the Send key is more complex because there are two alternatives.

1. If the operator has not entered ten digits, the program must ignore the Send key and place any other key into the entry.
2. If the operator has entered ten digits, the program must respond to the Send key by transferring control to the Send routine; and ignore all other keys.

Note that the flowchart has become much more difficult to organize and to follow. There is also no obvious way to check the flowchart.

Figure 17-6 shows the flowchart of the keyboard entry process with all the function keys. In this example, the flow of control is not simple. Clearly, some written description is necessary. The organization and layout of complex flowcharts requires careful planning. We have followed the process of adding features to the flowchart one at a time, but this still results in a large amount of redrawing. Again we should remember that throughout the keyboard entry process, the program must also refresh the displays if they are multiplexed and not controlled by shift registers or other hardware.

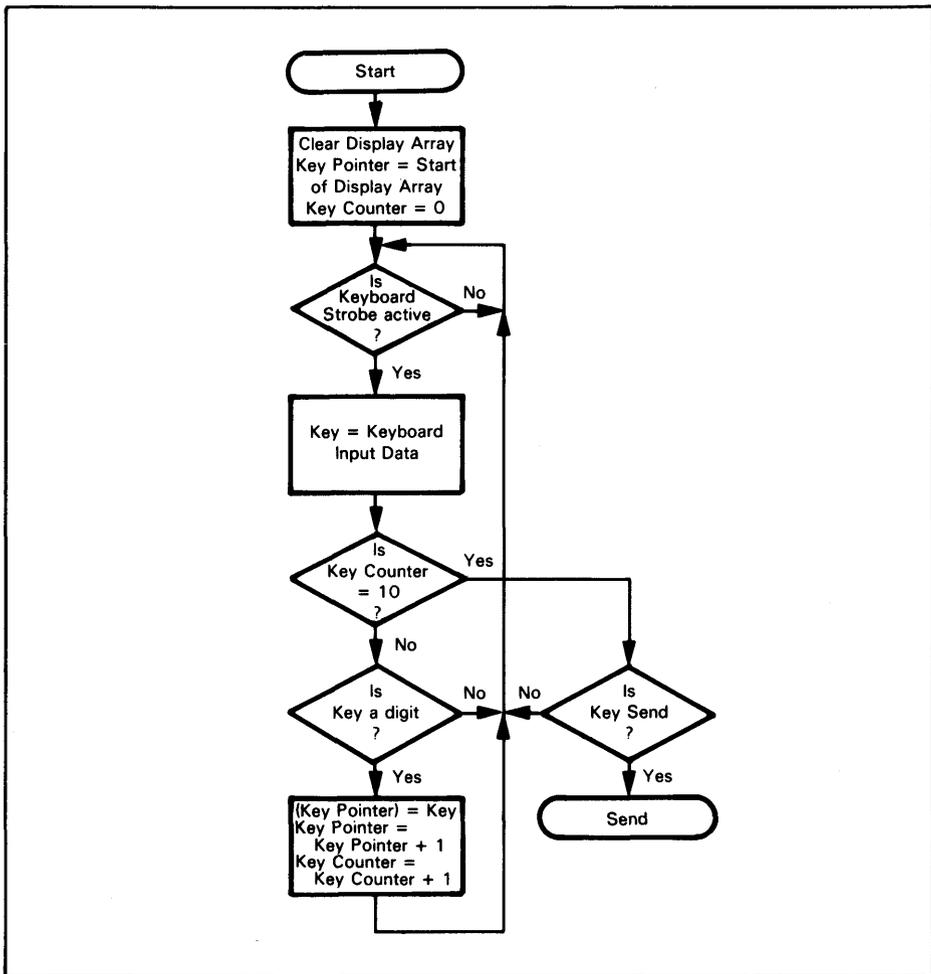


Figure 17-5. Flowchart of Keyboard Entry Process with Send Key

Figure 17-7 is the flowchart of a receive routine. We assume that the serial/parallel conversion and error checking are done in hardware (e.g., by a UART). The processor must:

1. Look for the header. (We assume that it is a single character.)
2. Read the destination address (we assume that it is three characters long) and see if the message is meant for this terminal; i.e., if the three characters agree with the terminal address.
3. Wait for the trailer character.
4. If the message is meant for the terminal, turn off the Busy light and go to Display Answer routine.
5. In the event of any errors, request retransmission by going to the appropriate RTRANS routine.

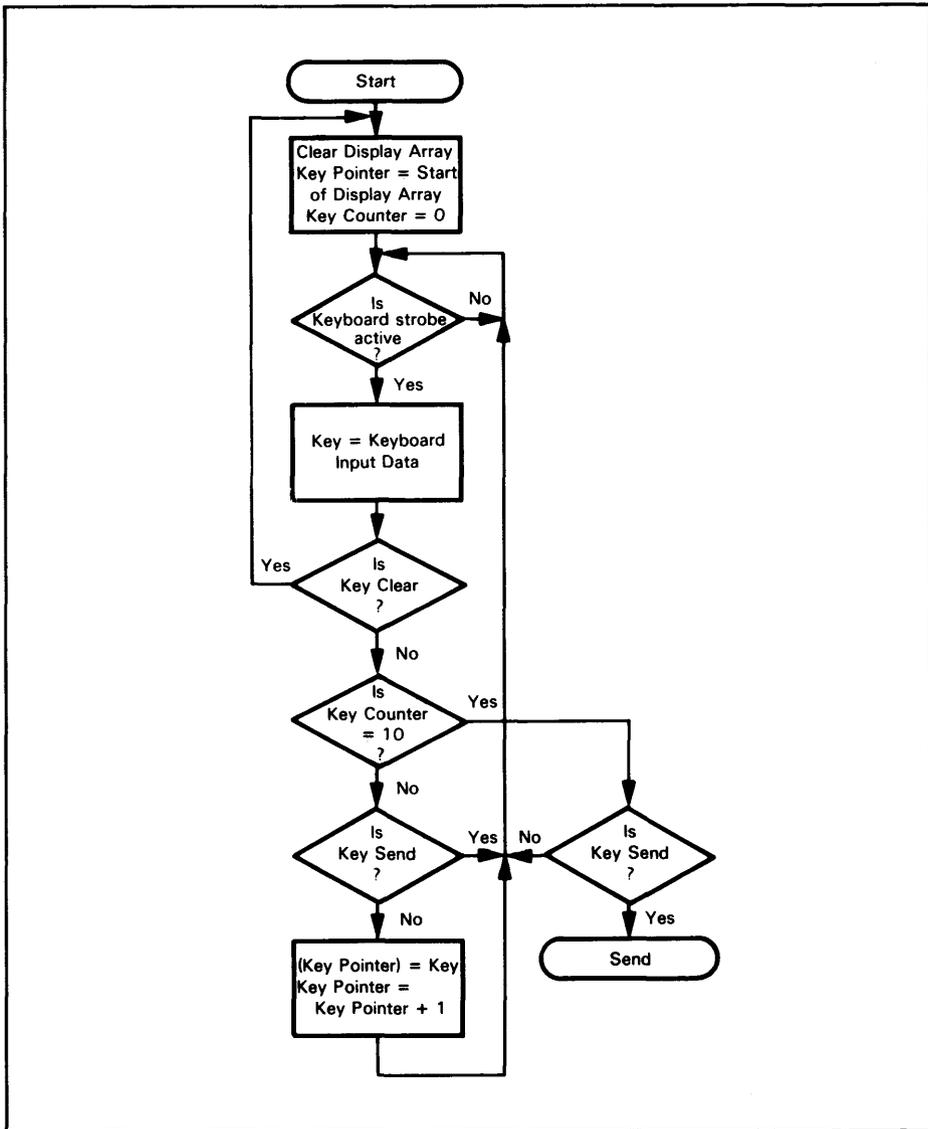


Figure 17-6. Flowchart of Keyboard Entry Process with Function Keys

This routine involves a large number of decisions, and the flowchart is neither simple nor obvious.

Clearly, we have come a long way from the simple flowchart (Figure 17-2) of the first example. A complete set of flowcharts for the transaction terminal would be a major task. It would consist of several interrelated charts with complex logic, and would require a large amount of effort. Such an effort would be just as difficult as writing a preliminary program, and not as useful, since you could not check the flowcharts on the computer.

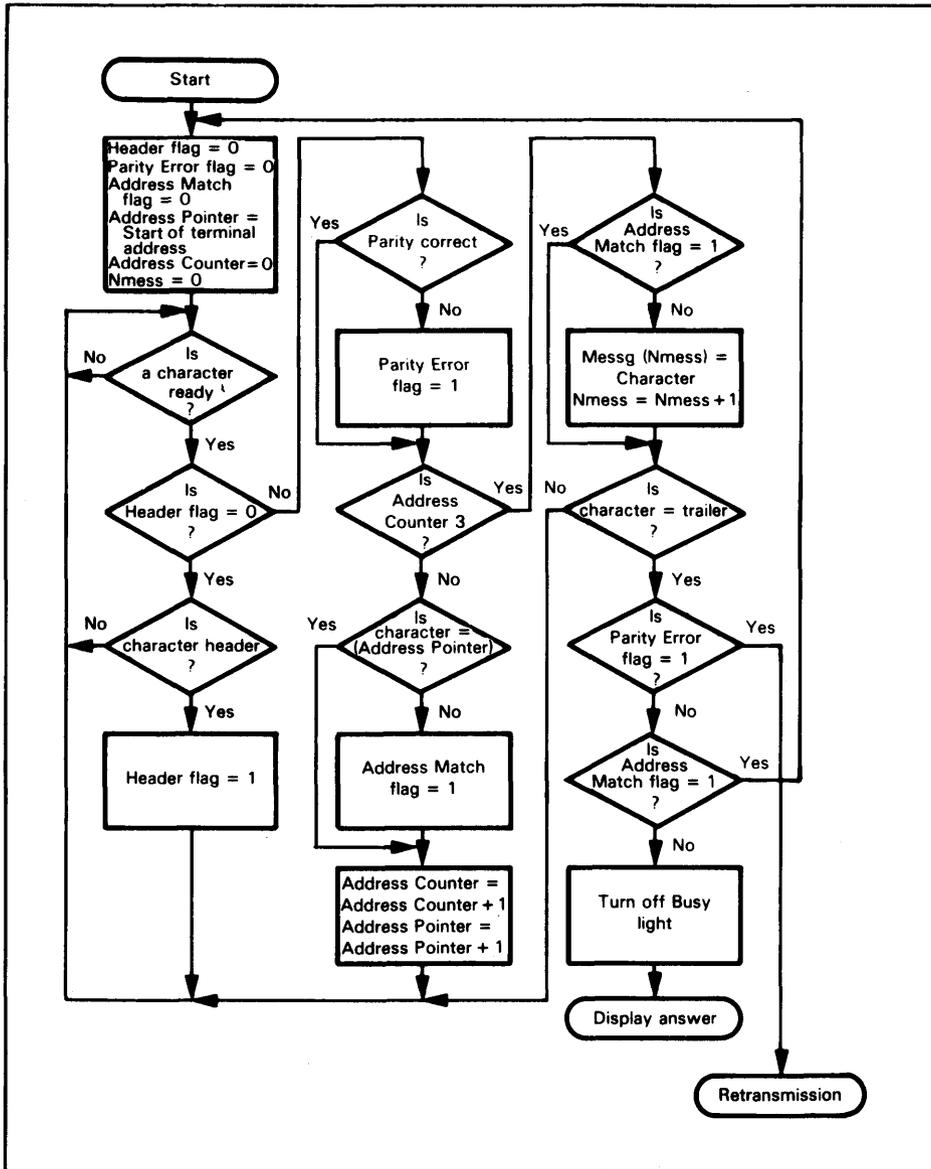


Figure 17-7. Flowchart of Receive Routine

MODULAR PROGRAMMING

Once programs become large and complex, flowcharting is no longer a satisfactory design tool. However, the problem definition and the flowchart can help you divide the program into reasonable sub-tasks. **The division of the entire program into sub-tasks or modules is called “modular programming.”** Clearly, most of the programs we presented in earlier chapters would typically be modules in a large program. **The problems that the designer faces in modular programming are how to divide the program into modules and how to put the modules together.**

ADVANTAGES OF MODULAR PROGRAMMING

The advantages of modular programming are obvious:

1. A single module is easier to write, debug, and test than an entire program.
2. A module is likely to be useful in many places and in other programs, particularly if it is reasonably general and performs a common task. You can build a library of standard modules.
3. Modular programming allows the programmer to divide tasks and use previously written programs.
4. Changes can be incorporated into one module rather than into the entire system.
5. Errors can often be isolated and then attributed to a single module.
6. Modular programming helps with project management, since it results in obvious goals and milestones.

DISADVANTAGES OF MODULAR PROGRAMMING

The idea of modular programming is so simple that its disadvantages are often ignored. These include:

1. Fitting the modules together can be a major problem, particularly if different people write the modules.
2. Modules require very careful documentation, since they may affect other parts of the program, such as data structures used by all the modules.
3. Testing and debugging modules separately is difficult, since other modules may produce the data used by the module being debugged and still other modules may use the results. You may have to write special programs (called “drivers”) just to produce sample data and test the programs. These drivers require extra programming effort that adds nothing to the system.
4. Programs may be very difficult to modularize. If you modularize the program poorly, integration will be very difficult, since almost all errors and changes will involve several modules.
5. Modular programs often require extra time and memory, since the separate modules may repeat functions.

Therefore, while modular programming is certainly an improvement over trying to write the entire program from scratch, it does have some disadvantages as well.

Important considerations include restricting the amount of information shared by modules, limiting design decisions that are subject to change to a single module, and restricting the access of one module to another.¹

PRINCIPLES OF MODULARIZATION

An obvious problem is that there are no proven, systematic methods for modularizing programs. We should mention the following principles:²

1. Modules that reference common data should be parts of the same overall module.
2. Two modules in which the first uses or depends on the second, but not the reverse, should be separate.
3. A module that is used by more than one other module should be part of a different overall module than the others.
4. Two modules in which the first is used by many other modules and the second is used by only a few other modules should be separate.
5. Two modules whose frequencies of usage are significantly different should be part of different modules.
6. The structure or organization of related data should be hidden within a single module.

If a program is difficult to modularize, you may need to redefine the tasks that are involved. Too many special cases or too many variables that require special handling are typical signs of inadequate problem definition.

EXAMPLES

Modularizing the Switch and Light System

This simple program can be divided into two modules:

Module 1 waits for the switch to be turned on and turns the light on in response.

Module 2 provides the one-second delay.

Module 1 is likely to be specific to the system, since it will depend on how the switch and light are attached. Module 2 will be generally useful, since many tasks require delays. Clearly, it would be advantageous to have a standard delay module that could provide delays of varying lengths. The module will require careful documentation so that you will know how to specify the length of the delay, how to call the module, and what registers and memory locations the module affects.

A general version of Module 1 would be far less useful, since it would have to deal with different types and connections of switches and lights.

You would probably find it simpler to write a module for a particular configuration of switches and lights rather than try to use a standard routine. Note the difference between this situation and Module 2.

Modularizing the Switch-Based Memory Loader

The switch-based memory loader is difficult to modularize, since all the programming tasks depend on the hardware configuration and the tasks are so simple that modules hardly seem worthwhile. The flowchart in Figure 17-3 suggests that one module might be the one that waits for the operator to press one of the three pushbuttons.

Some other modules might be:

- A delay module that provides the delay required to debounce the switches
- A switch and display module that reads the data from the switches and sends it to the displays
- A Lamp Test module

Highly system-dependent modules such as the last two are unlikely to be generally useful. This example is not one in which modular programming offers great advantages.

Modularizing the Verification Terminal

The verification terminal, on the other hand, lends itself very well to modular programming. The entire system can easily be divided into three main modules:

- **Keyboard and display module**
- **Data transmission module**
- **Data reception module**

A general keyboard and display module could handle many keyboard- and display-based systems. The sub-modules would perform such tasks as:

- Recognizing a new keyboard entry and fetching the data
- Clearing the array in response to a Clear Key
- Entering digits into storage
- Looking for the terminator or Send key
- Displaying the digits

Although the key interpretations and the number of digits will vary, the basic entry, data storage, and data display processes will be the same for many programs. Such function keys as Clear would also be standard. Clearly, **the designer must consider which modules will be useful in other applications, and pay careful attention to those modules.**

The data transmission module could also be divided into such sub-modules as:

1. Adding the header character.
2. Transmitting characters as the output line can handle them.
3. Generating delay times between bits or characters.
4. Adding the trailer character.
5. Checking for transmission failures; i.e., no acknowledgement, or inability to transmit without errors.

The data reception module could include sub-modules which:

1. Look for the header character.
2. Check the message destination address against the terminal address.
3. Store and interpret the message.
4. Look for the trailer character.
5. Generate bit or character delays.

INFORMATION HIDING PRINCIPLE

Note here how important it is that each design decision (such as the bit rate, message format, or error-checking procedure) be implemented in only one module. A change in any of these decisions will then require changes only to that single module. The other modules should be written so that they are totally unaware of the values chosen or the methods used in the implementing module. **An important concept here is the "information-hiding principle,"³ whereby modules share only information that is absolutely essential to getting the task done. Other information is hidden within a single module.**

Error handling is a typical situation in which information should be hidden. When a module detects a lethal error, it should not try to recover; instead, it should inform the calling module of the error status and allow that module to decide how to proceed. The reason is that the lower level module often lacks sufficient information to establish recovery procedures. For example, suppose that the lower level module is one that accepts numeric input from a user. This module expects a string of numeric digits terminated by a carriage return. Entry of a non-numeric character causes the module to terminate abnormally. Since the module does not know the context (i.e. is the numeric string an operand, a lone number, an I/O unit number, or the length of a file?), it cannot decide how to handle an error. If the module always followed a single error recovery procedure, it would lose its generality and only be usable in those situations where that procedure was required.

REVIEW OF MODULAR PROGRAMMING

Modular programming can be very helpful if you abide by the following rules:

1. **Use modules of 20 to 50 lines.** Shorter modules are usually a waste of time, while longer modules are seldom general and may be difficult to integrate.
2. **Make modules reasonably general.** Differentiate between common features like ASCII code or asynchronous transmission formats, which will be the same for many applications, and key identifications, number of displays, or number of characters in a message, which are likely to be unique to a particular application. Make the changing of the latter parameters simple. Major changes like different character codes should be handled by separate modules.
3. **Take extra time on modules like delays, display handlers, keyboard handlers, etc. that will be useful in other projects or in many different places in the present program.**
4. **Make modules independent of each other.** Restrict the flow of information between modules and implement each design in a single module.
5. **Do not modularize simple tasks** that are already easy to implement.

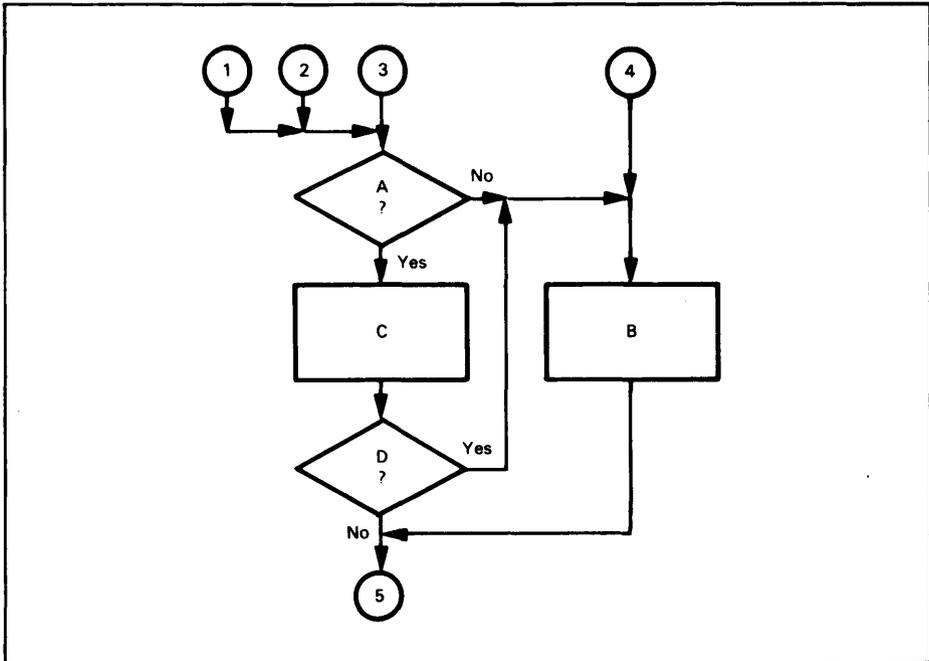


Figure 17-8. Flowchart of an Unstructured Program

STRUCTURED PROGRAMMING

How do you keep modules distinct and stop them from interacting? How do you write a program that has a clear sequence of operations so that you can isolate and correct errors? One answer is to use the methods known as “structured programming,” whereby each part of the program consists of elements from a limited set of structures and each structure has a single entry and a single exit.

Figure 17-8 shows a flowchart of an unstructured program. If an error occurs in Module B, we have five possible sources for that error. Not only must we check each sequence, but we also have to make sure that corrections do not affect any sequences. The usual result is that debugging becomes like wrestling an octopus. Every time you think the situation is under control, there is another loose tentacle somewhere.

BASIC STRUCTURES

The solution is to establish a clear sequence of operations so that you can isolate errors. Such a sequence uses single-entry, single-exit structures. A program consists of a sequence of structures; it may be a single statement or it may consist of structures that are nested within each other to any level of complexity. The required structures are listed below.

1. **An ordinary sequence;** that is, a linear structure in which programs are executed consecutively. If the sequence is:

P1
P2
P3

the computer executes P1 first, P2 second, and P3 third. P1, P2, and P3 may be single statements or complex programs.

2. **A conditional structure in which the execution of a program depends on a condition.**

There are many possible conditional structures, but a common one is “if C then P1 else P2” where C is a condition and P1 and P2 are programs. The computer executes P1 if C is true, and P2 if C is false. Figure 17-9 shows the logic of this structure. Note that it has a single entry and a single exit; the computer cannot enter or leave P1 or P2 other than through the structure.

3. **A loop structure in which a program is repeated until (or as long as) a condition holds.**

There are many possible loop structures. A common one (called a “do-while” structure) is “while C do P,” where C is a condition and P is a program. The computer continually checks C and then executes P as long as C is true.

An obvious alternative is “until C do P” in which the computer continually checks C and then executes P as long as C is false. Figures 17-10 and 17-11 show the logic of these alternatives. Both have a single entry and a single exit. The computer will not execute P at all if C is originally in the exit state; thus P is not executed at least once automatically as it is in a FORTRAN DO loop. Alternative structures like “do P while C” or “repeat P until C” produce the FORTRAN implementation in which the computer checks the condition after executing the program (remember Figures 5-1 and 5-2). This approach is often more efficient, but we will use only the form in Figure 17-10 to simplify

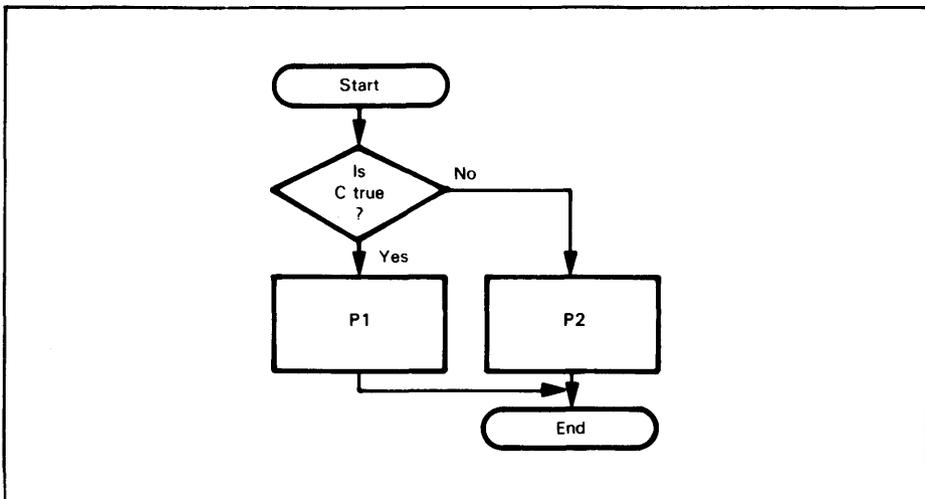


Figure 17-9. Flowchart of the If-Then-Else Structure

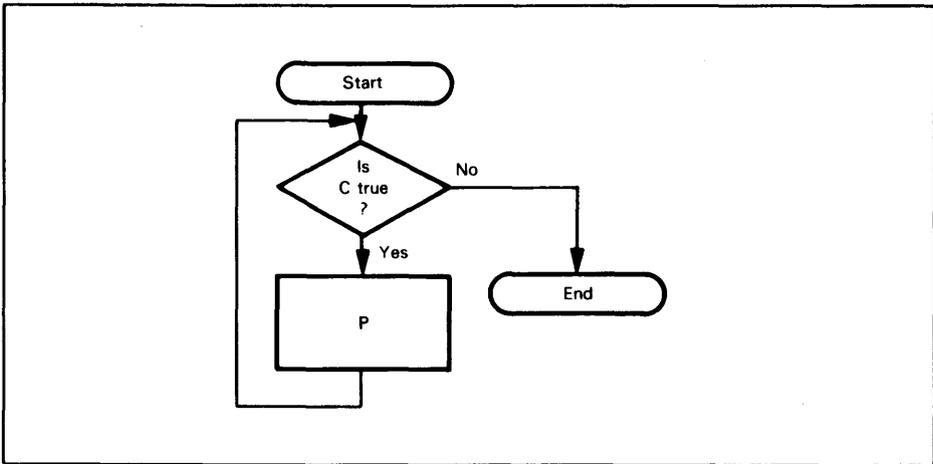


Figure 17-10. Flowchart of the Do-While Structure

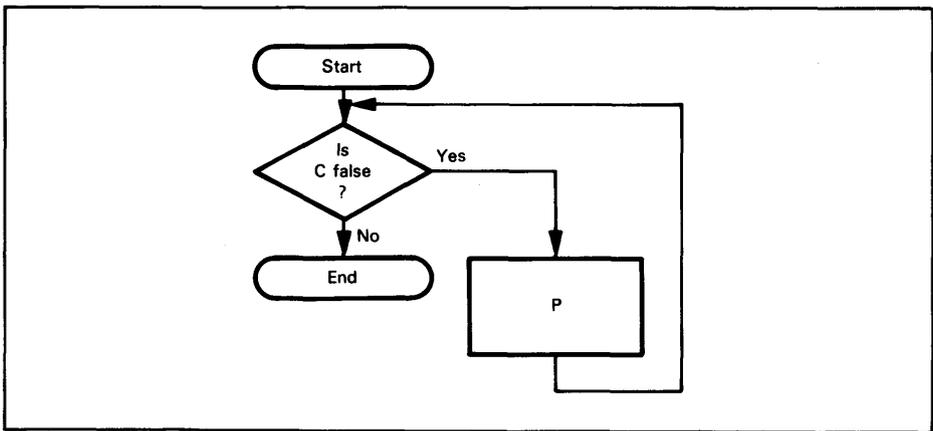


Figure 17-11. Flowchart of the Do-Until Structure

the discussion. Most high-level structured languages allow all four alternatives to provide flexibility. In most cases, the program P must eventually force C into the exit state; if it does not, the computer will execute P endlessly (the so-called DO FOREVER structure) as it must if P is the overall control program for an instrument, computer peripheral, test system, or electronic game.

4. **A case structure.** Although it is not a primitive structure like our first three, the case structure is so common that it merits a special description. The case structure is "case I of P₀, P₁, ..., P_n," where I is an index and P₀, P₁, ..., P_n are programs. The computer executes program P₀ if I is 0, P₁ if I is 1, and so on; it executes only one of the n programs. If I is greater than n (the number of programs in the case statement) or after execution of one of the programs,

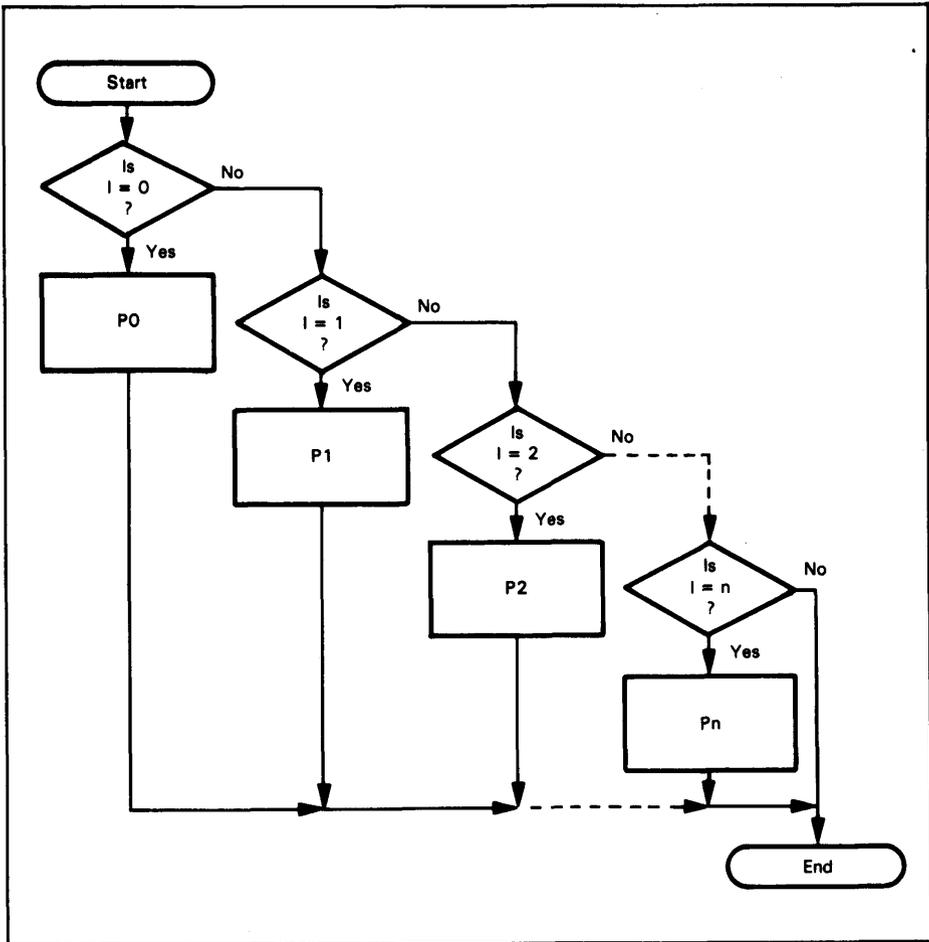


Figure 17-12. Flowchart of the Case Structure

the computer then executes the next sequential statement as shown in Figure 17-12. Obviously, we could implement a case structure as a series of conditional structures, much as we could implement a jump table as a series of conditional branches. However, the alternative implementations are long, awkward, and difficult to expand.

FEATURES AND EXAMPLES OF STRUCTURES

Note the following features of structured programming:

1. Only the three basic structures, and possibly a small number of auxiliary structures, are permitted. Variations of the conditional and loop structures may be allowed.

2. Structures may be nested to any level of complexity since any structure can, in turn, contain any of the structures.
3. Each structure has a single entry and a single exit.

Some examples of the conditional structure illustrated in Figure 17-9 are:

1. P2 included:

```
if X > 0 then NPOS = NPOS + 1
else NNEG = NNEG + 1
```

Both P1 and P2 are single statements.

2. P2 omitted:

```
if X ≠ 0 then Y = 1/X
```

Here no action is taken if C ($X \neq 0$) is false. P2 and "else" can be omitted in this case. Some examples of the loop structure illustrated in Figure 17-10 are:

1. Form the sum of integers from 1 to N.

```
I = 0
SUM = 0
do while I < N
  I = I + 1
  SUM = SUM + I
end
```

The computer executes the loop as long as $I < N$. If $N=0$, the program within the "do-while" is not executed at all.

2. Count characters in an array SENTENCE until you find an ASCII period.

```
NCHAR = 0
do while SENTENCE(NCHAR) ≠ PERIOD
  NCHAR = NCHAR + 1
end
```

The computer executes the loop as long as the character in SENTENCE is not an ASCII period. The count is zero if the first character is a period.

ADVANTAGES OF STRUCTURED PROGRAMMING

The advantages of structured programming are:

1. The sequence of operations is simple to trace. This allows you to test and debug programs easily.
2. The number of structures is limited and the terminology is standardized.
3. The structures can easily be made into modules.
4. Theoreticians have proved that the given set of structures is complete; that is, all programs can be written in terms of the three structures.
5. The structured version of a program is partly self-documenting and fairly easy to read.
6. Structured programs are easy to describe with program outlines.
7. Structured programming has been shown in practice to increase programmer productivity.

Structured programming basically forces much more discipline on the programmer than does modular programming. The result is more systematic and better organized programs.

DISADVANTAGES OF STRUCTURED PROGRAMMING

The disadvantages of structured programming are:

1. Only a few high-level languages (e.g., PL/M, PASCAL) will directly accept the structures. The programmer therefore has to go through an extra translation stage to convert the structures to assembly language code. The structured version of the program, however, is often useful as documentation.
2. Structured programs often execute more slowly and use more memory than unstructured programs.
3. Limiting the structures to the three basic forms makes some tasks very awkward to perform. The completeness of the structures only means that all programs can be implemented with them; it does not mean that a given program can be implemented efficiently or conveniently.
4. The standard structures are often quite confusing; e.g., nested “if-then-else” structures may be very difficult to read, since there may be no clear indication of where the inner structures end. A series of nested “do-while” loops can also be difficult to read.
5. Structured programs consider only the sequence of program operations, not the flow of data. Therefore, the structures may handle data awkwardly.
6. Few programmers are accustomed to structured programming. Many find the standard structures awkward and restrictive.

WHEN TO USE STRUCTURED PROGRAMMING

We are neither advocating nor discouraging the use of structured programming. It is one way of systematizing program design. In general, structured programming is most useful in the following situations:

- Larger programs, perhaps exceeding 1000 instructions.
- Applications in which memory usage is not critical.
- Low-volume applications where software development costs, particularly testing and debugging, are important factors.
- Applications involving string manipulation, process control, or other algorithms rather than simple bit manipulations.

In the future, we expect the cost of memory to decrease, the average size of microprocessor programs to increase, and the cost of software development to increase. Therefore, methods like structured programming, which decrease software development costs for larger programs but use more memory, will become more valuable.

Just because structured programming concepts are usually expressed in high-level languages does not mean that structured programming is not applicable to assembly language programming. On the contrary, **the assembly language programmer, with the total freedom of expression that assembly level programming allows, needs the structuring concept provided by structured programming. Creating modules with single entry and exit points, using simple control structures and keeping the complexity of each module minimal increases the productivity of the assembly language programmer.**

EXAMPLES

Structured Program for the Switch and Light System

The structured version of this example is:

```

SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
end
LIGHT = ON
DELAY 1
LIGHT = OFF

```

ON and OFF must have the proper definitions for the switch and light. We assume that DELAY is a module that provides a delay given by its parameter in seconds.

A statement in a structured program may actually be a subroutine. However, in order to conform to the rules of structured programming, the subroutine cannot have any exits other than the one that returns control to the main program.

Since "do-while" checks the condition before executing the loop, we set the variable SWITCH to OFF before starting. The structured program is straightforward, readable, and easy to check by hand. However, it would probably require somewhat more memory than an unstructured program, which would not have to initialize SWITCH and could combine the reading and checking procedures.

Structured Program for the Switch-Based Memory Loader

The switch-based memory loader is a more complex structured programming problem. We may implement the flowchart of Figure 17-3 as follows (a * indicates a comment, and we use "begin" and "end" around a conditionally executed program that consists of more than one line):

```

*
* CLEAR ADDRESS INITIALLY SO ITS STARTING VALUE IS ZERO
*
HIADDRESS = 0
LOADADDRESS = 0
*
* CONTINUOUSLY EXAMINE THE SWITCHES AND LOAD DATA INTO MEMORY
* NOTE THAT "DO FOREVER" IS JUST "DO WHILE" WITH NO CONDITION
*
do forever
*
* TEST HIGH ADDRESS BUTTON. IF IT IS BEING PRESSED, DEBOUNCE IT
* AND WAIT FOR THE OPERATOR TO RELEASE IT. THEN ENTER HIGH
* ADDRESS FROM THE SWITCHES AND SHOW IT ON THE LIGHTS
*
  if HIADDRBUTTON = 0 then
    begin
      do while HIADDRBUTTON = 0
        DELAY (DEBOUNCE TIME)
      end
      HIADDRESS = SWITCHES
      LIGHTS = SWITCHES
    end
*
* TEST LOW ADDRESS BUTTON. IF IT IS BEING PRESSED, DEBOUNCE IT AND
* WAIT FOR THE OPERATOR TO RELEASE IT. THEN ENTER LOW ADDRESS
* FROM THE SWITCHES AND SHOW IT ON THE LIGHTS
*
  if LOADDRBUTTON = 0 then
    begin
      do while LOADDRBUTTON = 0

```

17-22 6809 Assembly Language Programming

```
        DELAY (DEBOUNCE TIME)
      end
      LOADADDRESS = SWITCHES
      LIGHTS = SWITCHES
    end
*
*TEST DATA BUTTON. IF IT IS BEING PRESSED, DEBOUNCE IT AND WAIT
* FOR THE OPERATOR TO RELEASE IT. THEN ENTER DATA FROM THE
* SWITCHES, SHOW IT ON THE LIGHTS, AND STORE IT IN MEMORY AT
* (HIGH ADDRESS, LOW ADDRESS)
*
  if DATABUTTON = 0 then
    begin
      do while DATABUTTON = 0
        DELAY (DEBOUNCE TIME)
      end
      DATA = SWITCHES
      LIGHTS = SWITCHES
      (HIADDRESS, LOADADDRESS) = DATA
    end
*
*WAIT THE DEBOUNCING TIME BEFORE EXAMINING THE BUTTONS AGAIN.
* THIS DELAY DEBOUNCES THE RELEASE FOR SURE
*
    DELAY (DEBOUNCE TIME)
  end
*
*THE LAST END ABOVE TERMINATES THE
* DO FOREVER LOOP
*
```

Structured programs are not easy to write, but they can give a great deal of insight into the overall program logic. You can check the logic of the structured program by hand before writing any actual code.

Structured Program for the Verification Terminal

Let us look at the keyboard entry for the transaction terminal. We will assume that the display array is ENTRY, the keyboard strobe is KEYSTROBE, and the keyboard data is KEYIN. The structured program without the function keys is:

```
NKEYS = 10
*
*CLEAR ENTRY TO START
*
  do while NKEYS > 0
    NKEYS = NKEYS - 1
    ENTRY(NKEYS) = 0
  end
*
*FETCH A COMPLETE ENTRY FROM KEYBOARD
*
  do while NKEYS < 10
    if KEYSTROBE = ACTIVE then
      begin
        KEYSTROBE = INACTIVE
        ENTRY(NKEYS) = KEYIN
        NKEYS = NKEYS + 1
      end
    end
  end
```

Adding the SEND key means that the program must ignore extra digits after it has a complete entry, and must ignore the SEND key until it has a complete entry. The structured program is:

```
NKEYS = 10
*
*CLEAR ENTRY TO START
*
  do while NKEYS > 0
    NKEYS = NKEYS - 1
```

```

        ENTRY(NKEYS) = 0
    end
    *
    *WAIT FOR COMPLETE ENTRY FOLLOWED BY SEND KEY
    *
    do while KEY ≠ SEND or NKEYS ≠ 10
        if KEYSTROBE = ACTIVE then
            begin
                KEYSTROBE = INACTIVE
                KEY = KEYIN
                if NKEYS ≠ 10 and KEY ≠ SEND then
                    begin
                        ENTRY(NKEYS) = KEY
                        NKEYS = NKEYS + 1
                    end
                end
            end
        end
    end
end

```

Note the following features of this structured program.

1. The second if-then is nested within the first one, since the keys are only entered after a strobe is recognized. If the second if-then were on the same level as the first, a single key could fill the entry, since its value would be entered into the array during each iteration of the do-while loop.
2. KEY need not be defined initially, since NKEYS is set to zero as part of the clearing of the entry.

Adding the CLEAR key allows the program to clear the entry originally by simulating the pressing of CLEAR; i.e., by setting NKEYS to 10 and KEY to CLEAR before starting. The structured program must also only clear digits that have previously been filled. The new structured program is:

```

    *
    *SIMULATE COMPLETE CLEARING
    *
    NKEYS = 10
    KEY = CLEAR
    *
    *WAIT FOR COMPLETE ENTRY AND SEND KEY
    *
    do while KEY ≠ SEND or NKEYS ≠ 10
    *
    *CLEAR WHOLE ENTRY IF CLEAR KEY STRUCK
    *
        if KEY = CLEAR then
            begin
                KEY = 0
                do while NKEYS > 0
                    NKEYS = NKEYS - 1
                    ENTRY(NKEYS) = 0
                end
            end
        end
    *
    *GET DIGIT IF ENTRY INCOMPLETE
    *
        if KEYSTROBE = ACTIVE then
            begin
                KEYSTROBE = INACTIVE
                KEY = KEYIN
                if KEY < 10 and NKEYS ≠ 10 then
                    begin
                        ENTRY(NKEYS) = KEY
                        NKEYS = NKEYS + 1
                    end
                end
            end
        end
    end
end

```

Note that the program resets KEY to zero after clearing the array, so that the operation is not repeated.

We can similarly build a structured program for the receive routine. An initial program could just look for the header and trailer characters. We will assume that RSTB is the indicator that a character is ready. The structured program is:

```

*
* CLEAR HEADER FLAG TO START
*
HFLAG = 0
*
* WAIT FOR HEADER AND TRAILER
*
do while HFLAG = 0 or CHAR ≠ TRAILER
*
* GET CHARACTER IF READY. LOOK FOR HEADER
*
  if RSTB = ACTIVE then
    begin
      RSTB = INACTIVE
      CHAR = INPUT
      if CHAR = HEADER then HFLAG = 1
    end
  end
end

```

Now we can add the section that checks the message address against the three digits in TERMINAL ADDRESS (TERMADDR). If any of the corresponding digits are not equal, the ADDRESS MATCH flag (ADDRMATCH) is set to 1.

```

*
* CLEAR HEADER FLAG, ADDRESS MATCH FLAG, ADDRESS COUNTER TO START
*
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
*
* WAIT FOR HEADER, DESTINATION ADDRESS, AND TRAILER
*
do while HFLAG = 0 or CHAR ≠ TRAILER or ADDRCTR ≠ 3
*
* GET CHARACTER IF READY
*
  if RSTB = ACTIVE then
    begin
      RSTB = INACTIVE
      CHAR = INPUT
    end
  end
*
* CHECK FOR TERMINAL ADDRESS AND HEADER
*
  if HFLAG = 1 and ADDRCTR ≠ 3 then
    begin
      if CHAR ≠ TERMADDR(ADDRCTR) then ADDRMATCH = 1
      ADDRCTR = ADDRCTR + 1
    end
  end
  if CHAR = HEADER then HFLAG = 1
end

```

The program must now wait for a header, a three-digit identification code, and a trailer. You must be careful of what happens during the iteration when the program finds the header, and of what happens if an erroneous identification code character is the same as the trailer.

A further addition can store the message in MESSG. NMESS is the number of characters in the message; if it is not zero at the end, the program knows that the terminal has received a valid message. We have not tried to minimize the logic expressions in this program.

```

*
* CLEAR FLAGS, COUNTERS TO START
*
HFLAG = 0
ADDRMATCH = 0

```

```

ADDRCTR = 0
NMESS = 0
*
*WAIT FOR HEADER, DESTINATION ADDRESS, AND TRAILER
*
do while HFLAG = 0 or CHAR ≠ TRAILER or ADDRCTR ≠ 3
*
*GET CHARACTER IF READY
*
  if RSTB = ACTIVE then
    begin
      RSTB = INACTIVE
      CHAR = INPUT
    end
  *
*READ MESSAGE IF DESTINATION ADDRESS = TERMINAL ADDRESS
*
  if HFLAG = 1 and ADDRCTR = 3 then
    if ADDRMATCH = 0 and CHAR ≠ TRAILER then
      begin
        MESSG(NMESS) = CHAR
        NMESS = NMESS + 1
      end
    *
*CHECK FOR TERMINAL ADDRESS
*
    if HFLAG = 1 and ADDRCTR ≠ 3 then
      begin
        if CHAR ≠ TERMADDR(ADDRCTR) then ADDRMATCH = 1
        ADDRCTR = ADDRCTR + 1
      end
    *
*LOOK FOR HEADER
*
    if CHAR = HEADER then HFLAG = 1
  end

```

The program checks for the identification code only if it found a header during a previous iteration. It accepts the message only if it has previously found a header and a complete, matching destination address. The program must work properly during the iterations when it finds the header, the trailer and the last digit of the destination address. It must not try to match the header with the terminal address or place the trailer or the final digit of the destination address in the message. **You might try adding the rest of the logic from the flowchart (Figure 17-7) to the structured program. Note that the order of operations is often critical. You must be sure that the program does not complete one phase and start the next one during the same iteration.**

REVIEW OF STRUCTURED PROGRAMMING

Structured programming brings discipline to program design. It forces you to limit the types of structures you use and the sequence of operations. It provides single-entry, single-exit structures, which you can check for logical accuracy. Structured programming often makes the designer aware of inconsistencies or possible combinations of inputs. Structured programming is not a cure-all, but it does bring some order into a process that can be chaotic. The structured program should also aid in debugging, testing, and documentation.

Structured programming is not simple. The programmer must not only define the problem adequately, but must also work through the logic carefully. This is tedious and difficult, but it results in a clearly written, working program.

Terminators

The particular structures we have presented are not ideal and are often awkward. In addition, it can be difficult to determine where one structure ends and another begins, particularly if they are nested. Theorists may provide better structures in the future, or designers may wish to add some of their own. A terminator for each structure seems necessary, since indenting does not always clarify the situation. "End" is a logical terminator for the "do-while" loop. There is no obvious terminator, however, for the "if-then-else" statement; some theorists have suggested "endif" or "fi" ("if" backwards), but these are both awkward and detract from the readability of the program.

RULES FOR STRUCTURED PROGRAMMING

We suggest the following rules for applying structured programming:

1. **Begin by writing a basic flowchart** to help define the logic of the program.
2. **Start with the "sequential," "if-then-else," and "do-while" structures.** They are known to be a complete set, i.e., any program can be written in terms of these structures.
3. **Indent each level** a few spaces from the previous level, so that you will know which statements belong where.
4. **Use terminators for each structure:** e.g., "end" for the "do-while" and "endif" or "fi" for the "if-then-else." The terminators plus the indentation should make the program reasonably clear.
5. **Emphasize simplicity and readability.** Leave lots of spaces, use meaningful names, and make expressions as clear as possible. Do not try to minimize the logic at the cost of clarity.
6. **Comment the program** in an organized manner.
7. **Check the logic.** Try all the extreme cases or special conditions and a few sample cases. Any logical errors you find at this level will not plague you later.

TOP-DOWN DESIGN

The remaining problem is how to check and integrate modules or structures. Certainly we want to divide a large task into sub-tasks. But how do we check the sub-tasks in isolation and put them together? The standard procedure, called "bottom-up design," requires extra work in testing and debugging and leaves the entire integration task to the end. What we need is a method that allows testing and debugging in the actual program environment and modularizes system integration.

This method is "top-down design." Here we start by writing the overall supervisor program. We replace the undefined sub-programs by program "stubs," temporary programs that may either record the entry, provide the answer to a selected test problem, or do nothing. We then test the supervisor program to see that its logic is correct.

We proceed by expanding the stubs. Each stub will often contain sub-tasks, which we will temporarily represent as stubs. This process of expansion, debugging, and testing continues until all the stubs are replaced by working programs. Note that testing and integration occur at each level, rather than all at the end. No special driver or data generation programs are necessary. We get a clear idea of exactly where we are in the design. **Top-down design assumes modular programming, and is compatible with structured programming as well.**

DISADVANTAGES OF TOP-DOWN DESIGN

The disadvantages of top-down design are:

1. The overall design may not mesh well with system hardware.
2. It may not take good advantage of existing software.
3. Stubs may be difficult to write, particularly if they must work correctly in several different places.
4. Top-down design may not result in generally useful modules.
5. Errors at the top level can have catastrophic effects, whereas errors in bottom-up design are usually limited to a particular module.

In large programming projects, top-down design has been shown to greatly improve programmer productivity. However, almost all of these projects have used some bottom-up design in cases where the top-down method would have resulted in a large amount of extra work.

Top-down design is a useful tool that should not be followed to extremes. It provides the same discipline for system testing and integration that structured programming provides for module design. The method, however, has more general applicability, since it does not assume the use of programmed logic. However, top-down design may not result in the most efficient implementation.

EXAMPLES

Top-Down Design of Switch and Light System

The first structured programming example actually demonstrates top-down design as well. The program was:

```

SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
end
LIGHT = ON
DELAY 1
LIGHT = OFF

```

These statements are really stubs, since none of them is fully defined. For example, what does READ SWITCH mean? If the switch were one bit of input port SPORT, it really means:

```
SWITCH = SPORT and SMASK
```

where SMASK has a '1' bit in the appropriate position. The masking may, of course, be implemented with a Bit Test instruction.

Similarly, DELAY 1 actually means (if the processor itself provides the delay):

```
REG = COUNT
do while REG ≠ 0
  REG = REG - 1
end
```

COUNT is the appropriate number to provide a one-second delay. **The expanded version of the program is:**

```
SWITCH = 0
do while SWITCH = 0
  SWITCH = SPORT and MASK
end
LIGHT = ON
REF = COUNT
do while REG ≠ 0
  REG = REG - 1
end
LIGHT = not (LIGHT)
```

Certainly this program is more explicit, and could more easily be translated into actual instructions or statements.

Top-Down Design of the Switch-Based Memory Loader

This example is more complex than the first example, so we must proceed systematically. Here again, **the structured program contains stubs.**

For example, if the HIGH ADDRESS button is one bit of input port CPORT, “if HIADDRBUTTON=0” really means:

1. Input from CPORT
2. Logical AND with HAMASK

where HAMASK has a ‘1’ in the appropriate bit position and ‘0’ elsewhere. Similarly the condition “if DATABUTTON=0” really means:

1. Input from CPORT
2. Logical AND with DAMASK

So, the initial stubs could just assume that no buttons are being pressed:

```
HIADDRBUTTON = 1
LOADRBUTTON = 1
DATABUTTON = 1
```

A run of the supervisor program should show that it takes the implied “else” path through the “if-then-else” structures, and never reads the switches. Similarly, if the stub were:

```
HIADDRBUTTON = 0
```

the supervisor program should stay in the “do while HIADDRBUTTON=0” loop waiting for the button to be released. These simple runs check the overall logic.

Now we can expand each stub and see if the expansion produces a reasonable overall result. Note how debugging and testing proceed in a straightforward and modular manner. We expand the HIADDRBUTTON=0 stub to:

```
READ CPORT
HIADDRBUTTON = (CPORT) and HAMASK
```

The program should wait for the HIGH ADDRESS button to be released. The program should then display the values of the switches on the lights. This run checks for the proper response to the HIGH ADDRESS button.

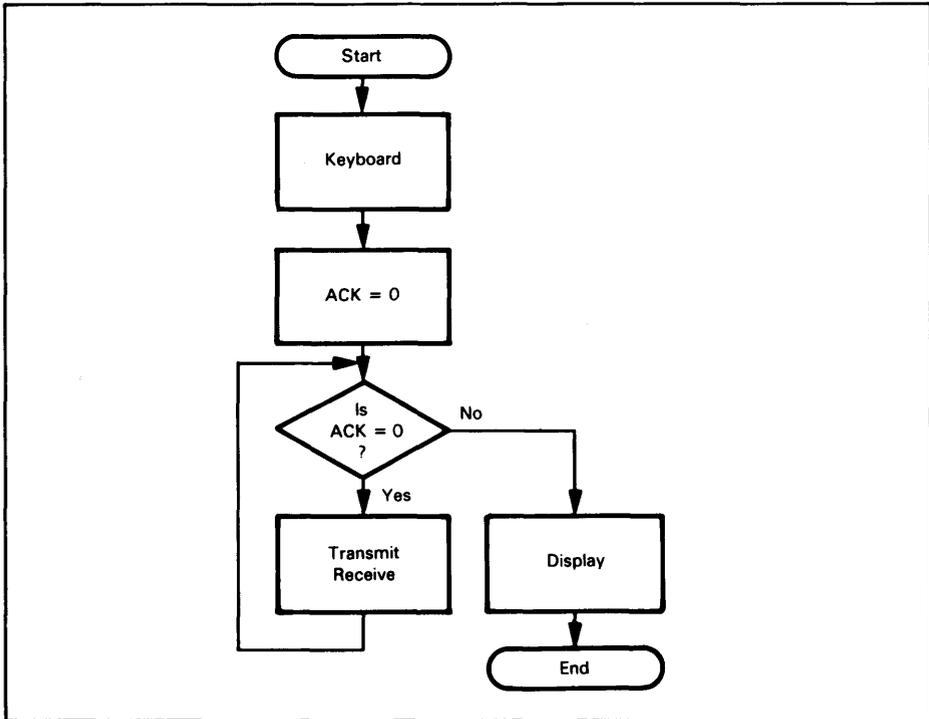


Figure 17-13. Initial Flowchart of Transaction Terminal

We then expand the LOW ADDRESS button module to:

```

    READ CPORT
    LOADDRBUTTON = (CPORT) and LAMASK
  
```

When the LOW ADDRESS button is released, the program should display the values of the switches on the lights. This run checks for the proper response to the LOW ADDRESS button.

Similarly, we can expand the DATA button module and check for the proper response to that button. The entire program will then have been tested.

When all the stubs have been expanded, the coding, debugging, and testing stages will all be complete. Of course, we must know exactly what results each stub should produce. However, many logical errors will become obvious at each level without any further expansion.

Top-Down Design of Verification Terminal

This example, of course, will have more levels of detail. We could start with the following program (see Figure 17-13 for a flowchart):

```

    KEYBOARD
    ACK = 0
    do while ACK = 0
      TRANSMIT
      RECEIVE
    end
    DISPLAY
  
```

Here, **KEYBOARD**, **TRANSMIT**, **RECEIVE**, and **DISPLAY** are program stubs that will be expanded later. **KEYBOARD**, for example, could simply place a ten-digit verified number into the appropriate buffer.

The next stage of expansion could produce the following program for **KEYBOARD** (see Figure 17-14):

```

VER = 0
do while VER = 0
  COMPLETE = 0
  do while COMPLETE = 0
    KEYIN
    KEYDS
  end
  VERIFY
end
end
    
```

Here **VER=0** means that an entry has not been verified; **COMPLETE=0** means that the entry is incomplete. **KEYIN** and **KEYDS** are the keyboard input and display routines respectively. **VERIFY** checks the entry. A stub for **KEYIN** would simply place a random entry (from a random number table or generator) into the buffer and set **COMPLETE** to 1.

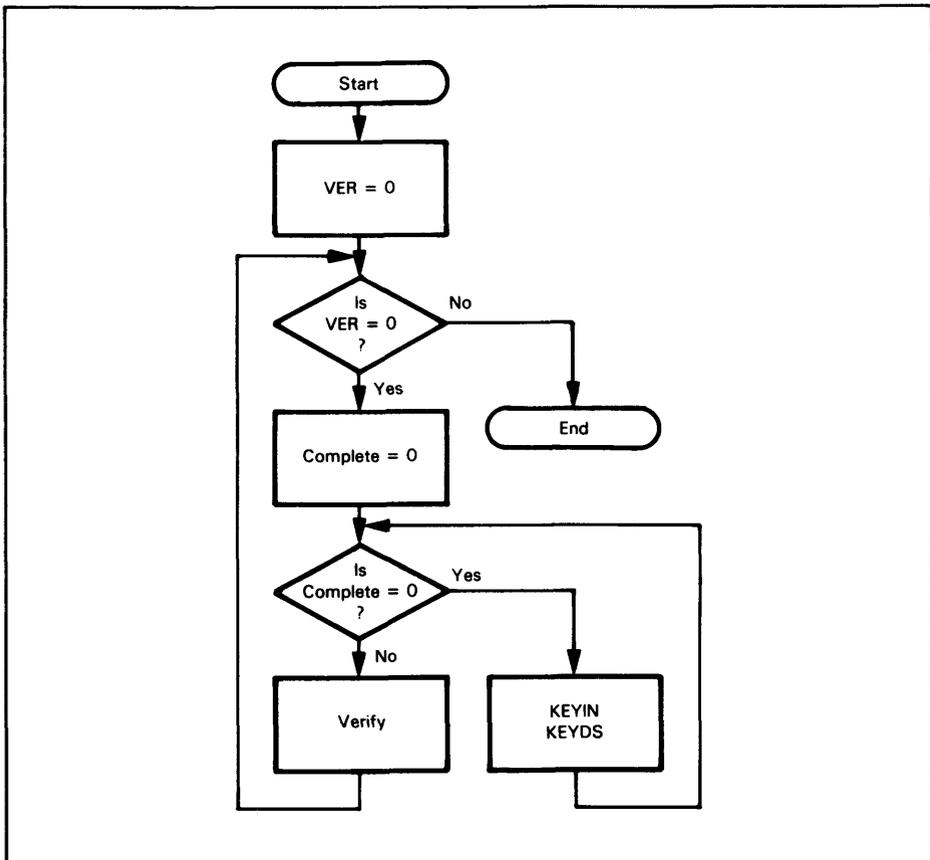


Figure 17-14. Flowchart for Expanded **KEYBOARD** Routine

We would continue by similarly expanding, debugging, and testing TRANSMIT, RECEIVE, and DISPLAY. Note that you should expand each program by one level so that you do not perform the integration of an entire program at any one time. You must use your judgment in defining levels. Too small a step wastes time, while too large a step gets you back to the problems of system integration that top-down design is supposed to solve.

REVIEW OF TOP-DOWN DESIGN

Top-down design brings discipline to the testing and integration stages of program design. It provides a systematic method for expanding a flowchart or problem definition to the level required to actually write a program. Together with structured programming, it forms a complete set of design techniques.

Like structured programming, top-down design is not simple. The designer must have defined the problem carefully and must work systematically through each level. Here again, the methodology may seem tedious, but the payoff can be substantial if you follow the rules.

We recommend the following approach to top-down design:

1. Start with a basic flowchart.
2. Make the stubs as complete and as separate as possible.
3. Define precisely all the possible outcomes from each stub and select a test set.
4. Check each level carefully and systematically.
5. Use the structures from structured programming.
6. Expand each stub by one level. Do not try to do too much in one step.
7. Watch carefully for common tasks and data structures.
8. Test and debug after each stub expansion. Do not try to do an entire level at a time.
9. Be aware of what the hardware can do. Do not hesitate to stop and do a little bottom-up design where that seems necessary.

DESIGNING DATA STRUCTURES

Beginning programmers seldom think about data structures. They generally assume that the data will be stored somewhere in the computer's memory, much as records are piled into a cabinet or books into a bookcase. Designing data structures seems as far-fetched as establishing a complete card catalog for one's books or records; few people take organization to such lengths.

But the fact is that **most computer-based systems involve a surprisingly large amount of data processing.** Numerical algorithms assume that the processor can easily find the element in the next row or next column of an array. Editor programs assume that the processor can easily find the next character, the previous line, a particular string of characters, or the starting point of an entire paragraph or page. An operator interface for a piece of test equipment may assume that the processor can easily find a particular command or data entry and move it from one place to another. **Imagine how difficult**

the following tasks would be to implement if the data is simply scattered through memory or organized in a long, linear array:

1. The operator of a machine tool wants to insert two extra cutting steps between steps 14 and 15 of a 40-step pattern.
2. The operator of a chemical processing plant wants to see the last ten values of the temperature at the inlet to tank #5.
3. An accounting clerk wants to enter a new account into an alphabetical list.

The processor may spend most of its time finding the data, moving from one data item to the next, and organizing the data.

SELECTING DATA STRUCTURES

Obviously, we cannot provide a complete description of data structures here.^{4,5} Just as clearly, **the design of data structures has great influence on the design of programs if the data is complex. We will briefly mention the following considerations in selecting data structures:**

1. **How are the data items related?** Closely related items should be accessible from each other, since such accesses will be frequent.
2. **What kind of operations will be performed on the data?** Simple linear structures are adequate if the data is always handled in a single, fixed order. However, more complex structures are essential if the tasks involve operations such as searching, editing, or sorting.
3. **Can standard structures be used?** Methods are readily available for handling structures such as queues, stacks, and linked lists. Other arrangements will require special programming.
4. **What kind of access is necessary?** Clearly we need more structure if we must find elements that are identified by a number or a relative position, rather than just the first or last entries. We must organize the data to make the accesses as rapid as possible.

REVIEW OF PROBLEM DEFINITION AND PROGRAM DESIGN

You should note that we have spent two entire chapters without mentioning any specific microprocessor or assembly language, and without writing a single line of actual code. However, you should now know a lot more about the examples than you would if we had just asked you to write the programs at the start. **Although we often think of the writing of computer instructions as a key part of software development, it is actually one of the easiest stages.**

Once you have written a few programs, coding will become simple. You will soon learn the instruction set, recognize which instructions are really useful, and remember the common sequences that make up the largest part of most programs. You

will then find that many of the other stages of software development remain difficult and have few clear rules.

We have suggested some ways to systematize the important early stages. In the problem definition stage, you must define all the characteristics of the system — its inputs, outputs, processing, time and memory constraints, and error handling. You must particularly consider how the system will interact with the larger system of which it is a part, and whether that larger system includes electrical equipment, mechanical equipment, or a human operator. You must start at this stage to make the system easy to use and maintain.

In the program design stage, several techniques can help you to systematically specify and document the logic of your program. Modular programming forces you to divide the total program into small, distinct modules. Structured programming provides a systematic way of defining the logic of those modules, while top-down design is a systematic method for integrating and testing them. Of course, no one can compel you to follow all of these techniques; they are, in fact, guidelines more than anything else. But they do provide a unified approach to design, and you should consider them a basis on which to develop your own approach.

REFERENCES

1. D. L. Parnas (see the references below) has been a leader in the area of modular programming.
2. Collected by B. W. Unger (see reference below).
3. Formulated by D. L. Parnas.
4. K. J. Thurber. and P. C. Patton. *Data Structures and Computer Architecture*, Lexington Books, Lexington, Mass., 1977.
5. K. S. Shankar. "Data Structures, Types, and Abstractions," *Computer*, April 1980, pp. 67-77.

The following references provide additional information on problem definition and program design:

Chapin, N. *Flowcharts*, Auerbach, Princeton, N. J., 1971.

Dalton, W. F. "Design Microcomputer Software like Other Systems — Systematically," *Electronics*, January 19, 1978, pp. 97-101.

Dijkstra, E. W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Halstead, M. H. *Elements of Software Science*, American Elsevier, New York, 1977.

Hughes, J. K. and J. I. Michtom. *A Structured Approach to Programming*, Prentice-Hall, Englewood Cliffs, N. J., 1977.

Morgan, D. E. and D. J. Taylor. "A Survey of Methods for Achieving Reliable Software," *Computer*, February 1977, pp. 44-52.

Myers, W. "The Need for Software Engineering," *Computer*, February 1978, pp. 12-25.

Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-58.

- Parnas, D. L. "A Technique for the Specification of Software Modules with Examples," *Communications of the ACM*, May 1973, pp. 330-336.
- Phister, M. Jr. *Data Processing Technology and Economics*, Santa Monica Publishing Co., Santa Monica, Ca., 1976.
- Schneider, V. "Prediction of Software Effort and Project Duration — Four New Formulas," *SIGPLAN Notices*, June 1978, pp. 49-59.
- Schneiderman, B. et al. "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *Communications of the ACM*, June 1977, pp. 373-381.
- Tausworthe, R. C. *Standardized Development of Computer Software*, Prentice-Hall, Englewood Cliffs, N. J., 1977 (Part 1); 1979 (Part 2).
- Unger, B. W. "Programming Languages for Computer System Simulation," *Simulation*, April 1978, pp. 101-10.
- Wirth, N. *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N. J., 1976.
- Wirth, N. *Systematic Programming: an Introduction*, Prentice-Hall, Englewood Cliffs, N. J., 1973.
- Yourdon, E. U. *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N. J., 1975.

18

Documentation

Software development must yield more than just a working program. A software product must also include the documentation that allows it to be used, maintained, and extended. Adequate documentation is helpful during program debugging and testing, and essential in the later stages of the program's life cycle.

SELF-DOCUMENTING PROGRAMS

Although no program is ever completely self-documenting, some of the rules that we mentioned earlier can help. These include:

- Clear, simple, structure with as few transfers of control (jumps) as possible
- Use of meaningful names and labels
- Use of names for I/O devices, parameters, numerical factors, subroutines, branch destinations, etc.
- Emphasis on simplicity rather than on minor savings in memory usage, execution time, or typing

For example, the following program sends a string of characters to a teletypewriter:

```
          LDB      $40
          LDX      #$1000
W         LDA      ,X+
          STA      $8008
          JSR      XXX
          DECB
          BNE     W
          SWI
```

CHOOSING USEFUL NAMES

Even without comments we can improve the program as follows:

```

COUNT EQU $40
MESSG EQU $1000
TTYPIA EQU $8008
        LDB COUNT
        LDX #MESSG
OUTCH LDA ,X+
        STA TTYPIA
        JSR BITDLY
        DECB
        BNE OUTCH
        SWI

```

This program is undoubtedly easier to understand than the earlier version. **Even without further documentation, you could probably guess at the function of the program and the meanings of most of the variables.** Other documentation techniques cannot substitute for self-documentation.

Some further notes on choosing names:

1. **Use the obvious name** when it is available, like TTY or CRT for output devices, START or RESET for addresses, DELAY or SORT for subroutines, COUNT or LENGTH for data.
2. **Avoid acronyms** like S16BA for SORT 16-BIT ARRAY. These seldom mean anything to anybody.
3. **Use full words** or close to full words when possible, like DONE, PRINT, SEND, etc.
4. **Keep the names as distinct** as possible. Avoid names that look alike, such as TEMPI and TEMP1, or resemble operation codes or other built-in names.

COMMENTS

Comments are a simple form in which to provide additional documentation. However, few programs (even those used as examples in books) have effective comments. **You should consider the following guidelines for good comments:**

1. **Don't explain the internal effects of the instruction.** Instead, explain the purpose of the instruction in the program. Comments like

```
DECB          B = B - 1
```

do not help the reader understand the program. A more useful comment is

```
DECB          LINE NUMBER = LINE NUMBER - 1
```

Remember that the standard manuals contain descriptions of how the processor executes its instructions. The comments should explain what tasks the program is performing and what methods it is using.

2. **Make the comments as clear as possible.** Do not use abbreviations or acronyms unless they are well-known (like ASCII, PIA, or UART) or standard (like no for number, ms for millisecond, etc.) Avoid comments like

```
DECB          LN = LN - 1
```

or

```
DECB          DEC. LN BY 1
```

The extra typing is certainly worthwhile.

3. **Comment every important or obscure point.** Be particularly careful to mark operations that may not have obvious functions, such as

```

ANDA  #00100000  TURN OFF TAPE READER
or
LDA   A,X        GET SEVEN-SEGMENT CODE FROM TABLE

```

Clearly, I/O operations often require extensive comments. If you're not exactly sure what an instruction does, or if you have to think about it, add a clarifying comment. The comment will save you time later and will be helpful in documentation.

4. **Don't comment the obvious.** A comment on each line makes it difficult to find the important points. Standard sequences like

```

DECB
BNE          SEARCH

```

need not be marked unless you're doing something special. One comment will often suffice for several lines, as in

```

LSRA          GET MOST SIGNIFICANT DIGIT
LSRA
LSRA
LSRA

LDA  $40      EXCHANGE MOST SIGNIFICANT, LEAST
LDB  $41      SIGNIFICANT BYTES
STA  $41
STB  $40

```

5. **Place comments on the lines to which they refer or at the start of a sequence.**
6. **Keep your comments up-to-date.** If you change the program, change the comments.
7. **Use standard forms and terms** in commenting. Don't worry about repetitiveness. Varied names for the same things are confusing, even if the variations are just COUNT and COUNTER, START and BEGIN, DISPLAY and LEDS, or PANEL and SWITCHES. You gain nothing from inconsistency. Minor variations may be obvious to you now, but may not be clear later; others will get confused immediately.
8. **Make comments mingled with instructions brief.** Leave a complete explanation to header comments and other documentation. Otherwise the program gets lost in the comments and you may have a hard time even finding the actual instructions.
9. **Keep improving your comments.** If you come to one that you cannot read or understand, take the time to change it. If you find that the listing is getting crowded, add some blank lines. The comments won't improve themselves; in fact, they will just become worse as you leave the task behind and forget exactly what you did.
10. **Use comments to place a heading in front of every major section, subsection, or subroutine.** The heading should describe the functions of the code that follows it; it should include information about the algorithm employed, the inputs and outputs, and any incidental effects that may be produced.
11. **If you modify a working program, use comments to describe the modifications that you made and identify the date and author of the revision.** This

information should go both at the front of the program (so a user can easily tell one version from another) and at the points where changes were actually made.

Remember, **comments are important. Good ones will save you time and effort.** Put some work into comments and try to make them effective.

EXAMPLES

18-1. COMMENTING A MULTIPLE-PRECISION ADDITION ROUTINE

The basic program is:

```

LDB      $40
LDX      #$41
LDY      #$51
ANDCC    %%11111110
ADBYTE   LDA      ,X
          ADCA     ,Y+
          STA      ,X+
          DECB
          BNE      ADBYTE
          SWI

```

Important Points

First, comment the important points. These are typically initializations, data fetches, and processing operations. Don't bother with standard sequences like updating pointers and counters. Remember that **names are clearer than numbers, so use them freely.**

The new version of the program is:

```

*
*THE FOLLOWING PROGRAM PERFORMS MULTI-BYTE BINARY ADDITION
*
* INPUTS: LOCATION 0040 CONTAINS LENGTH OF NUMBERS IN BYTES
*         LOCATIONS 0041 ON CONTAIN ONE OPERAND STARTING WITH LSB'S
*         LOCATIONS 0051 ON CONTAIN ONE OPERAND STARTING WITH LSB'S
*
* OUTPUTS: LOCATIONS 0041 ON CONTAIN SUM STARTING WITH LSB'S
*
LENGTH EQU  $40
OPER1   EQU  $41
OPER2   EQU  $51
LDB     LENGTH      COUNT = LENGTH OF NUMBERS IN BYTES
LDX     #OPER1      POINT TO LSB'S OF FIRST OPERAND, SUM
LDY     #OPER2      POINT TO LSB'S OF SECOND OPERAND
ANDCC   %%11111110
ADBYTE  LDA      ,X      GET A BYTE FROM FIRST OPERAND
          ADCA     ,Y+     ADD A BYTE FROM SECOND OPERAND
          STA      ,X+     STORE SUM OVER FIRST OPERAND
          DECB
          BNE      ADBYTE
          SWI

```

Obscure Functions

Second, look for instructions that may not have obvious functions and explain their purposes with comments. Here, the purpose of ANDCC#%11111110 (the 6800 operation code CLC is surely easier to understand) is to clear the Carry flag before adding the least significant bytes.

Questions for Commenting

Third, ask yourself whether the comments tell you what you would need to know to use the program; for example:

1. Where is the program entered? Are there alternative entry points?
2. What parameters are necessary? How and in what form must they be supplied?
3. What operations does the program perform?
4. From where does it get the data?
5. Where does it store the results?
6. What special cases does it consider?
7. What does the program do about errors?
8. How does it exit?

Some questions may be irrelevant and some answers may be obvious. **Make sure, however, that you wouldn't have to dissect the program to answer the important questions. Remember also that too much explanation may be an obstacle to using the program.** Are there any changes you would like to see in the listing? If so, make them — you are the one who has to decide if the commenting is adequate and reasonable.

```

*
*THE FOLLOWING PROGRAM PERFORMS MULTI-BYTE BINARY ADDITION
*
* INPUTS: LOCATION 0040 CONTAINS LENGTH OF NUMBERS IN BYTES
*         LOCATIONS 0041 ON CONTAIN ONE OPERAND STARTING WITH LSB'S
*         LOCATIONS 0051 ON CONTAIN ONE OPERAND STARTING WITH LSB'S
* OUTPUTS: LOCATIONS 0041 ON CONTAIN SUM STARTING WITH LSB'S
*
LENGTH EQU    $40          LENGTH OF NUMBERS IN BYTES
OPER1 EQU     $41          LSB'S OF ONE OPERAND AND SUM
OPER2 EQU     $51          LSB'S OF OTHER OPERAND
LDB LENGTH    COUNT = LENGTH OF NUMBERS IN BYTES
LDX #OPER1    POINT TO LSB'S OF FIRST OPERAND, SUM
LDY #OPER2    POINT TO LSB'S OF SECOND OPERAND
ANDCC #11111110 CLEAR CARRY FOR ADDITION OF LSB'S
ADDBYTE LDA   ,X          GET A BYTE FROM FIRST OPERAND
ADCA ,Y+      ADD A BYTE FROM SECOND OPERAND
STA ,X+      STORE SUM OVER FIRST OPERAND
DECB
BNE ADDBYTE  CONTINUE UNTIL ALL BYTES ADDED
SWI

```

18-2. COMMENTING A TELETYPEWRITER OUTPUT ROUTINE

The basic program is:

```

LDA $60
ASLA
LDB #11
TBIT STA $8008
JSR BITDLY
RORA
ORCC #00000001
DECB
BNE TBIT
SWI

```

Commenting the important points and adding names gives:

```

*
*TELETYPEWRITER OUTPUT PROGRAM
*
*THIS PROGRAM SENDS THE CONTENTS OF MEMORY LOCATION 0060 TO THE
  TELETYPEWRITER
*
* INPUTS: CHARACTER TO BE TRANSMITTED IN MEMORY LOCATION 0060
* OUTPUTS: NONE
*
NBITS EQU 11          NUMBER OF BITS PER CHARACTER
TDATA EQU $60        ADDRESS OF CHARACTER TO BE TRANSMITTED
TTYPIA EQU $8008     TELETYPEWRITER OUTPUT DATA PORT
                LDA TDATA          GET DATA
                ASLA             SHIFT DATA LEFT AND FORM START BIT
                LDB #NBITS        COUNT = NUMBER OF BITS IN CHARACTER
TBIT     STA TTYPIA    SEND A BIT TO TELETYPEWRITER
                JSR BITDLY        WAIT 1 BIT TIME
                RORA             GET NEXT BIT
                ORCC #%00000001   SET CARRY TO FORM STOP BITS
                DECB
                BNE TBIT         COUNT BITS
                SWI

```

Changing the Program

Note how easily we could change this program so that it would transfer a whole string of data, starting at the address in locations BUFPTR and BUFPTR + 1 and ending with an "03" character (ASCII ETX). Furthermore, let us make the terminal a 30 character per second device with one stop bit (we will have to change subroutine BITDLY). Try making the changes before looking at the listing.

```

*
*STRING OUTPUT PROGRAM
*
*TERMINAL
*TRANSMISSION CEASES WHEN AN ASCII ETX IS ENCOUNTERED
*
*INPUTS: MEMORY LOCATIONS 0060 AND 0061 CONTAIN STARTING ADDRESS
* OF STRING TO BE TRANSMITTED
*
* OUTPUTS: NONE
*
BUFPTR EQU $60        STARTING ADDRESS OF OUTPUT DATA BUFFER
ENDCH EQU $03        ENDING CHARACTER = ASCII ETX
NBITS EQU 10         NUMBER OF BITS PER CHARACTER
TRMPIA EQU $8008     TERMINAL OUTPUT DATA PORT
                LDX BUFPTR        GET STARTING ADDRESS OF OUTPUT BUFFER
TCHAR     LDA ,X+     GET A CHARACTER FROM THE BUFFER
                CMPA #ENDCH      IS IT THE ENDING CHARACTER?
                BEQ DONE        YES, DONE
                ASLA             NO, SHIFT IT LEFT TO FORM A START BIT
                LDB #NBITS        COUNT = NUMBER OF BITS IN CHARACTER
TBIT     STA TRMPIA   SEND A BIT TO THE TERMINAL
                JSR BITDLY
                RORA             GET NEXT BIT
                ORCC #%00000001   SET CARRY TO FORM STOP BIT
                DECB
                BNE TBIT         COUNT BITS
                BRA TCHAR
DONE     SWI

```

Good comments will help you change a program to meet new requirements. For example, try changing the last program so that it:

- Starts each message with ASCII STX (02) followed by a three-digit identification code stored in memory locations IDCODE through IDCODE + 2.

- Adds no start or stop bits
- Waits 1 ms between bits
- Transmits 40 characters, starting with the one located at the address in DPTR and DPTR + 1.
- Ends each message with two consecutive ASCII ETXs (03)

FLOWCHARTS AS DOCUMENTATION

We have already described the use of flowcharts as a design tool in Chapter 17. Flowcharts are also useful in documentation, particularly if:

- They are not cluttered or too detailed.
- Their decision points are explained and marked clearly.
- They include all branches.
- They correspond to the actual program listings.

Flowcharts are helpful if they give you an overall picture of the program. They are not helpful if they are just as difficult to read as the program listing.

STRUCTURED PROGRAMS AS DOCUMENTATION

A structured program can serve as documentation for an assembly language program if:

- You describe the purpose of each section in the comments.
- You make it clear which statements are included in each conditional or loop structure by using indentation and ending markers.
- You make the total structure as simple as possible.
- You use a consistent, well-defined language.

The structured program can help you check the logic or improve it. Furthermore, since the structured program is machine-independent, it can also help you implement the same task on another computer.

MEMORY MAPS

A memory map is simply a list of all the memory assignments in a program. The map allows you to determine the amount of memory needed, the locations of data or subroutines, and the parts of memory not allocated. The map is a handy reference for finding storage locations and entry points and for dividing memory between different routines or programmers. The map will also give you easy access to data and subroutines if you need them in later extensions or in maintenance. **Sometimes a graphical map is more helpful than a listing.**

A typical map is:

Program Memory		
Address	Routine	Purpose
E000 - E1FF	RDKBD	Interrupt Service Routine for Keyboard
E200 - E240	BRKPT	Breakpoint Routine Entered Via Software Interrupt
E241 - E250	DELAY	Generalized Delay Program
E251 - E270	DSPLY	Control Program for Operator Displays
E271 - E3EF	SUPER	Main Supervisor Program
E3F0 - E3FF		Interrupt and Reset Vectors
Data Memory		
Address	Name	Purpose
0000	NKEYS	Number of Keys Pressed by Operator
0001 - 0002	KBPTR	Keyboard Buffer Pointer
0003 - 0041	KBUFFR	Keyboard Buffer
0042 - 0050	DBUFFR	Display Buffer
0051 - 006F	TEMP	Miscellaneous Temporary Storage
0070 - 00FF	STACK	Hardware Stack

The map may also list additional entry points and include a specific description of the unused parts of memory.

PARAMETER AND DEFINITION LISTS

Parameter and definition lists at the start of the main program and each subroutine make understanding and changing the program far simpler. The following rules can help.

1. Separate RAM locations, I/O units, parameters, definitions, and fixed memory addresses.
2. Arrange lists alphabetically when possible, with a description of each entry.
3. Give each parameter that might change a name and include it in the lists. Such parameters may include time constants, inputs or codes corresponding to particular keys or functions, control or masking patterns, starting or ending characters, thresholds, etc.
4. List fixed memory addresses separately. These may include Reset and interrupt service addresses, the starting address of the program, RAM areas, Stack areas, etc.
5. Give each port used by an I/O device a name, even though devices may share ports in the current system. The separation will make it easier for you to expand or change the I/O section.

A typical list of definitions is:

```

*
*MEMORY SYSTEM CONSTANTS
*
FRQSRV EQU $E100    SERVICE ADDRESS FOR FAST INTERRUPT
IRQSRV EQU $E200    SERVICE ADDRESS FOR REGULAR INTERRUPT
RAMST  EQU 0        STARTING ADDRESS FOR TEMPORARY STORAGE
RESET  EQU $E300    RESET ADDRESS
STKPTR EQU $0180    STARTING ADDRESS FOR HARDWARE STACK
*
*I/O UNITS
*
DSPLAY EQU $8006    OUTPUT PIA FOR DISPLAYS
KBDIN  EQU $8004    INPUT PIA FOR KEYBOARD
KBDOUT EQU $8006    OUTPUT PIA FOR KEYBOARD
TTYPIA EQU $8008    DATA PORT FOR TTY
*
*RAM STORAGE
*
      ORG RAMST      TEMPORARY DATA STORAGE AREA
NKEYS RMB 1          NUMBER OF KEYS
KPTR  RMB 2          KEYBOARD BUFFER POINTER
KBFR  RMB $40        KEYBOARD INPUT BUFFER
DISBFR RMB $10       DISPLAY OUTPUT BUFFER
TEMP  RMB $14        TEMPORARY DATA STORAGE
*
*PARAMETERS
*
BOUNCE EQU 2         DEBOUNCING TIME IN MS
GOKEY  EQU 10        IDENTIFICATION NUMBER FOR 'GO' KEY
MSCNT  EQU $7A       COUNT FOR 1 MS DELAY
OPEN   EQU $0F       INPUT PATTERN WHEN NO KEYS ARE PRESSED
TPULS  EQU 1         PULSE LENGTH FOR DISPLAYS IN MS
*
*DEFINITIONS
*
ALLHI  EQU $FF       ALL ONES INPUT
STCON  EQU $80       OUTPUT FOR START OF CONVERSION PULSE

```

Of course, the RAM entries will usually not be in alphabetical order, since the designer must order these to minimize the number of address changes required in the program.

LIBRARY ROUTINES

Standard documentation of subroutines helps you build a library of programs that are easy to use. If you describe each subroutine with a standard form, anyone can see at a glance what the routines do and how to use them. You should organize the forms carefully, dividing them, for example, by processor, language, and type of program. Remember, without proper documentation and organization, using the library may be more difficult than writing programs from scratch. If you are going to use subroutines from a library or other outside source, you must know all their effects in order to debug your overall program.

STANDARD PROGRAM LIBRARY FORMS

Among the information that you will need in the standard form is:

- Purpose of the program

- Processor used
- Language used
- Parameters required and how they are passed to the subroutine
- Results produced and how they are passed to the main program
- Number of bytes of memory used
- Number of clock cycles required. This number may be an average or a typical figure, or it may vary widely. Actual execution time will, of course, depend on the processor clock rate and the memory cycle time.
- Registers affected
- Flags affected
- A typical example
- Error handling
- Special cases
- Documented program listing

If the program is complex, the standard library form should also include a general flowchart or a structured outline of the program. As we have mentioned before, a library program is most likely to be useful if it performs a single function in a general manner.

18-3. SUM OF DATA/LIBRARY ROUTINE

Purpose: The program SUM8 computes the sum of a set of 8-bit unsigned binary numbers.

Language: 6809 Assembler

Initial Conditions: Starting address of set of numbers in Index Register X, length of set in Accumulator B.

Final Conditions: Sum in Accumulator A.

Requirements:

Memory — 7 bytes

Time — $7 + 11N$ clock cycles, where N is the length of the set of numbers.

Registers — A,B,X

All flags affected

Typical Case: (all data in hexadecimal)

```
Start:
      (X)  = 0050  Starting address
      (B)  = 03   Length of set
      (0050) = 27  Data items
      (0051) = 3E
      (0052) = 26
End:
      (A)  = 8B  Sum
```

Error Handling: Program ignores all carries. Carry flag reflects only the result of the last operation. Initial contents of Accumulator B must be 1 or more.

Listing:

```

*
*SUM OF A SET OF 8-BIT DATA ITEMS
*
SUM8  CLRA          CLEAR SUM TO START
ADBYTE ADDA  ,X+    ADD AN ELEMENT TO THE SUM
      DECB
      BNE  ADBYTE
      RTS

```

18-4: DECIMAL TO SEVEN SEGMENT CONVERSION/LIBRARY ROUTINE

Purpose: The program SEVEN converts a binary-coded decimal number to a seven-segment display code.

Language: 6809 Assembler

Initial Conditions: Data in Accumulator A.

Final Conditions: Seven-segment code in Accumulator B.

Requirements:

Memory — 21 bytes, including the seven-segment code table (10 entries).

Time — 20 clock cycles if the data is valid, 12 if it is not.

Registers — A,B,X

All flags affected

Input data in Accumulator A is not changed.

Typical Case: (data in hexadecimal)

Start:

(A) = 05 Decimal data

End:

(B) = 6D Seven-segment representation of input data

Error Handling: Program returns zero in Accumulator B if the data is not a decimal digit.

Listing:

```

*
*DECIMAL TO SEVEN-SEGMENT CODE CONVERSION
*
SEVEN CLRB          GET ERROR CODE
      CMPA  #9      IS DATA A DECIMAL DIGIT?
      BHI  DONE     NO, KEEP ERROR CODE AS RESULT
      LDX  #SSEG    YES, GET SEVEN-SEGMENT CODE FROM TABLE
      LDB  A,X
DONE   RTS
SSEG  FCB  $3F,$06,$5B,$4F,$66
      FCB  $6D,$7D,$07,$7F,$6F

```

18-5. DECIMAL SUM/LIBRARY ROUTINE

Purpose: The program DECSUM adds two multi-digit decimal (BCD) numbers with digits packed two to a byte.

Language: 6809 Assembler

Initial Conditions: Address of LSD's of one operand (and sum) in Index Register X, address of LSD's of other operand in Index Register Y. Length of numbers (in bytes) in Accumulator B. Numbers arranged starting with LSD's at lowest address.

Final Conditions: Sum replaces number with starting address in Index Register X.

Requirements:

Memory — 13 bytes

Time — $8 + 23N$ clock cycles, where N is the number of bytes.

Registers — A,B,X,Y

All flags affected — Carry shows if sum produced a carry.

Typical Case: (all data in hexadecimal)

Start:

(X)	=	0060	Address of LSD's of one operand and sum
(Y)	=	0050	Address of LSD's of other operand
(B)	=	02	Length of operands in bytes
(0060)	=	34	} 5534 is first operand
(0061)	=	55	
(0050)	=	88	} 1588 is second operand
(0051)	=	15	

End:

(0060)	=	22	} 7122 is decimal sum
(0061)	=	71	
Carry	=	0	

Error Handling: Program does not check the validity of decimal inputs. The contents of Accumulator B must be 1 or more.

Listing:

```
*
* MULTI-DIGIT DECIMAL (BCD) ADDITION
*
DECSUM  ANDCC  ##11111110  CLEAR CARRY TO START
ADDIGS  LDA    ,X          GET TWO DIGITS OF FIRST OPERAND
        ADCA  ,Y+         ADD TWO DIGITS OF SECOND OPERAND
        DAA                    DECIMAL CORRECTION
        STA  ,X+         STORE SUM OVER FIRST OPERAND
        DECB
        BNE  ADDIGS
        RTS
```

TOTAL DOCUMENTATION

Complete documentation of microprocessor software will include all or most of the elements that we have mentioned.

DOCUMENTATION PACKAGE

The total documentation package may involve:

- General flowcharts

- A written description of the program
- A list of all parameters and definitions
- A memory map
- A documented listing of the program
- A description of the test plan and test results

The documentation may also include:

- Programmer's flowcharts
- Data flowcharts
- Structured programs

Even this package is sufficient only for non-production software. **Production software also requires the following documents:**

- Program Logic Manual
- User's Guide
- Maintenance Manual

Program Logic Manual

The program logic manual expands the written explanation provided with the software. It should explain the system's design goals, algorithms, and tradeoffs, assuming a reader who is competent technically but lacks detailed knowledge of the program. It should provide a step-by-step guide to the operations of the program and it should explain the data structures and their manipulation.

User's Guide

The User's Guide is the most important single piece of documentation. No matter how well-designed the system may be, it will not be useful if no one can understand its operations or take advantage of its features. The User's Guide should explain system features and their use, provide frequent examples that clarify the text, and give tested step-by-step directions. The writing of User's Guides requires care and objectivity, since the writer must be able to take an outsider's point of view.

One problem in writing User's Guides is the need to avoid overwhelming the beginner or taxing the patience of the experienced user. Two separate versions can help overcome this problem. A guide for the beginner can explain the most common features of the program with the aid of simple examples and detailed discussions. A guide for the experienced user can provide more extensive descriptions of features and fewer details. Remember that the beginner needs help getting started, whereas the experienced user wants organized reference material.

Maintenance Manual

The maintenance manual is designed for the programmer who has to modify the system. It should explain the procedures for any changes or expansion that have been designed into the program.

IMPORTANCE OF DOCUMENTATION

Documentation should not be taken lightly or left to the last minute. Good documentation, combined with proper programming practices, is not only an important part of the final product but can also make development simpler, faster, and more productive. **The designer should make consistent and thorough documentation part of every stage of software development.**

19

Debugging

As we noted at the beginning of this section, debugging and testing are among the most time-consuming stages of software development. **Even though such methods as modular programming, structured programming, and top-down design can simplify programs and reduce the frequency of errors, debugging and testing are still difficult** because they are so poorly defined. The selection of an adequate set of test data is seldom a clear or scientific process. Finding errors sometimes seems like a game of “pin the tail on the donkey,” except that the donkey is moving and the programmer must position the tail by remote control. Surely, few tasks are as frustrating as debugging programs.

This chapter will first describe the tools available to aid in debugging. It will then discuss basic debugging procedures, describe the common types of errors, and present some examples of program debugging. The next chapter will describe how to select test data and test programs.

We will describe only the purposes of most debugging tools. There is little standardization in this area and we cannot discuss all the available products. The examples show the uses, advantages, and limitations of some common tools.

Debugging tools have two major functions. One is to pin the error down to a short section of the program; the other is to provide more detailed information about what the computer is doing than is provided by normal runs and so make the source of the error obvious. Current debugging tools do not find and correct errors by themselves; you must know enough about what is happening to recognize and correct the error when the debugging tools zero in on it and show its effects in detail.

SIMPLE DEBUGGING TOOLS

The most common simple debugging tools are:

- A single-step facility
- A breakpoint facility
- A trace facility
- A Register Dump Program (or utility)
- A Memory Dump Program

SINGLE STEP

The single-step facility allows you to execute the program one instruction or one memory cycle at a time. Only some 6809-based microcomputers have this facility, since the circuitry is fairly complex. Of course, all that you can see when the computer executes a single-step are the states of the output lines that you are monitoring. The most important lines are:

- Data Bus
- Address Bus
- Control Lines
- BUSY and READ/WRITE

If you monitor these lines either in hardware or in software, you can see the progression of addresses, instructions, and data as the program is executed. You can determine what kinds of operations the CPU is performing. This information will be sufficient for you to identify such errors as Jump or Branch instructions with incorrect conditions or destinations, omitted or incorrect addresses, incorrect operation codes, and incorrect data values. However, you cannot see the contents of registers, flags, or memory locations without some additional debugging tool.

Furthermore, a single-step mode obviously slows the processor way below its normal speed. You cannot check delay loops or I/O operations in real time. Nor can a single-step mode help you find timing errors or errors in the interrupt or DMA systems. In fact, the single-step mode typically operates at less than one millionth of normal processor speed. To single-step through one second of real processor time would require more than ten days. The single-step mode, therefore, is useful only to check the logic of a short sequence of instructions.

BREAKPOINT

A breakpoint is a place at which the program will automatically halt or wait so that the user can examine the current status of the system. The program will not continue until the user orders its resumption. Breakpoints allow you to check or pass through an entire section of a program. Thus, to see if an initialization routine is correct, you can place a breakpoint at the end of it and run the program. You can then check memory locations and registers to see if the entire section is correct. However, note that if the section is not correct, you still must pinpoint the error, either with earlier breakpoints or with a single-step mode.

Breakpoints complement the single-step mode. You can use breakpoints either to localize the error or to pass through sections that you know are correct. You can then do the detailed debugging in the single-step mode. In some cases, breakpoints do not affect program timing. They can then be used to check input/output and interrupts.

Software and Hardware Interrupts

Breakpoints often use the microprocessor's interrupt system (see Chapter 15). **The 6809 has 3 Software Interrupt instructions (SWI, SWI2, and SWI3) that can act as breakpoints. If you are not already using the interrupt inputs (IRQ, FIRQ, and NMI), you can use those vectors as externally controlled breakpoints.** Table 15-1 lists the addresses used by the various instructions and inputs. The breakpoint routine can print the contents of registers and memory locations or just wait (by executing a conditional jump dependent on a switch input) until the user allows the computer to proceed. But remember that the interrupts and SWI instructions use the Hardware Stack to store the return address and the contents of the registers. Figure 19-1 shows a service routine in which SWI results in an endless loop. The program would have to clear this breakpoint with a RESET or non-maskable interrupt (SWI disables the maskable interrupts).

Inserting Breakpoints

The simplest method for inserting breakpoints is to replace the first byte or bytes of an instruction with a Software Interrupt instruction. Any Software Interrupt instruction will automatically direct the processor to the breakpoint routine and save the current values of all the registers (except the Hardware Stack Pointer) in the Hardware Stack. Figure 15-2 shows the order in which the processor saves its registers. The breakpoint routine can print all the register contents by starting at the address in the Hardware Stack Pointer. The only problem is that the return program counter value will be the address following the Software Interrupt. You may want to reduce that value by 1 or 2, either to display the actual breakpoint address or to resume the program correctly after restoring the original instruction. Typical programs to reduce the value are (using the methods discussed in Chapter 15):

1. Decrement the return address by 1 (if you are using SWI)

```
LDX    $0A,S      GET RETURN ADDRESS
LEAX   -1,X       MOVE IT BACK 1
STX    $0A,S      PUT ADJUSTED RETURN ADDRESS IN STACK
```

2. Decrement the return address by 2 (if you are using SWI2 or SWI3)

```
LDX    $0A,S      GET RETURN ADDRESS
LEAX   -2,X       MOVE IT BACK 2 (SWI2 AND SWI3 USE
                  TWO BYTES EACH)
STX    $0A,S      PUT ADJUSTED RETURN ADDRESS IN STACK
```

BRKPT	ORG	BRKPT	BREAKPOINT ROUTINE
	BRA	BRKPT	WAIT IN PLACE
	-		
	ORG	\$FFFA	SOFTWARE INTERRUPT BREAKPOINT
	FDB	BRKPT	ADDRESS OF BREAKPOINT ROUTINE
	-		

Figure 19-1. A Simple Breakpoint Routine

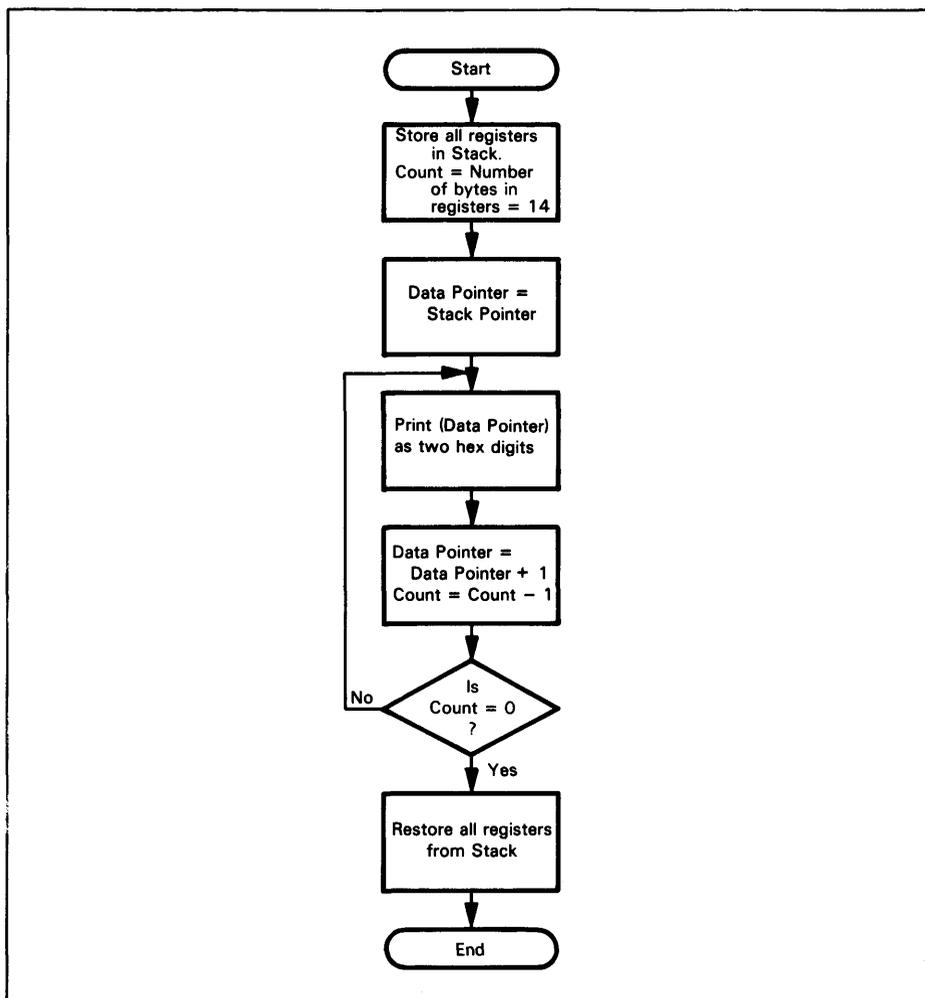


Figure 19-2. Flowchart of Register Dump Program

Setting and Clearing Breakpoints

Many monitors have facilities for automatically inserting (*setting*) and removing (*clearing*) breakpoints based on one of the Software Interrupt instructions. Such breakpoints do not affect the timing of the program until one of them is executed. However, you obviously cannot replace instructions that are in ROM or PROM. Other monitors implement breakpoints by actually checking the address lines or the Program Counter in hardware or in software. This method allows the user to set breakpoints on addresses in ROM or PROM, but it may affect system timing if the address must be checked in software. A more powerful facility would allow the user to enter an address to which the processor would transfer control. Another possibility would be a return dependent on a switch as in the following example.

```
BRKPT  TST  PIADRA  WAIT FOR SWITCH IN BIT 7 TO CLOSE
        BMI  BRKPT
        RTI
```

Of course, other PIA data or control lines could also be used. Remember that RTI reenables the interrupts automatically. If a PIA interrupt is used, the service routine must read the PIA data register to clear the interrupt status bit.

Precautions in Using Breakpoints

When you use breakpoints (whether manually or through monitor facilities), remember the following precautions:

1. **Only set breakpoints at addresses that contain operation codes.** Replacing data or parts of addresses with SWI instructions can result in chaos.
2. **Interpret the results carefully.** Remember that the computer has not yet executed the instruction that was replaced.
3. **Check all conditions before resuming the program.** You may have to change the program counter, correct the contents of registers or memory locations, clear breakpoints that are no longer necessary, and set new breakpoints. Methods for resuming programs vary greatly, so consult your microcomputer's User's Guide. Be particularly careful never to resume a program in the middle of an instruction (that is, at an address that does not contain an operation code) or in the middle of an I/O or timing operation (e.g., sending data to a teletypewriter) that cannot logically be resumed after a delay.

TRACE

The trace facility allows you to see intermediate results. **A simple trace prints the contents of all registers and other variables after each instruction is executed.** This obviously produces a large amount of information, most of which is irrelevant or repetitive. **Better trace facilities allow you to specify what you want traced and how often you want the values printed.** This results in less information, but means that you must decide what you need before instituting the trace.

The following approach will help you use traces:

1. **Decide what you need before executing the trace.** Otherwise, you will not know what to do with the results.
2. **Start by tracing only one or two variables and printing the results infrequently.** This will give you less information to analyze at one time.
3. **Use breakpoints to limit the extent of the trace.**
4. **Use whatever facilities your computer has to mark the output.** Otherwise, you will end up with pages of unidentified numbers and you will spend most of your time just figuring out what they are.

REGISTER DUMP

A Register Dump utility is a program that lists the contents of all the CPU registers. This information is usually not directly obtainable. **The following routine will**

Value	Register
A0	SH
8B	SL
DC	CC
27	A
E5	B
01	DP
B6	XH
97	XL
BD	YH
14	YL
05	UH
F3	UL
D7	PCH
3C	PCL

Figure 19-3. Results of a Typical 6809 Register Dump

print the contents of all the registers on the system printer, if we assume that PRTHEX prints the contents of Accumulator A as two hexadecimal digits. Figure 19-2 is a flowchart of the program and Figure 19-3 shows a typical result. We assume that the routine is entered with a BSR or JSR instruction that stores the old Program Counter at the top of the Hardware Stack. An interrupt or Software Interrupt instruction will store all the registers (except the Hardware Stack Pointer) on the Hardware Stack.

```

*
*SAVE ALL CPU REGISTERS IN THE HARDWARE STACK (PC IS ALREADY
*THERE)
*
RDUMP  PSHS  U,Y,X,DP,B,A,CC  SAVE USER REGISTERS
        LEAU  12,S            CALCULATE ORIGINAL STACK POINTER
        PSHS  U              SAVE ORIGINAL STACK POINTER
*
*PRINT CONTENTS OF REGISTERS
*ORDER IS S(HIGH),S(LOW),CC,A,B,DP,X(HIGH),X(LOW),Y(HIGH),Y(LOW),
* U(HIGH),U(LOW),PC(HIGH),PC(LOW)
*
        TFR  S,U            POINT TO START OF REGISTER STORAGE
        LDB  #14           NUMBER OF BYTES = 14
PRNT1  LDA  ,U+           GET A BYTE FROM THE STACK
        JSR  PRTHEX       AND PRINT IT
        DECB
        BNE  PRNT1
*
*RESTORE REGISTERS FROM THE STACK, INCLUDING THE ORIGINAL STACK
* POINTER
*
        PULS  U            RESTORE AND DISCARD STACK POINTER
        PULS  PC,U,Y,X,DP,B,A,CC  RESTORE OTHER REGISTERS AND RETURN

```

MEMORY DUMP

A Memory Dump is a program that lists the contents of memory on an output device (such as a printer). This is a much more efficient way to examine data arrays or entire programs than just looking at single locations. However, large memory dumps are not useful (except to supply scrap paper) because of the sheer mass of information that

they produce. They may also take a long time to execute on a slow printer. **Small dumps may, however, provide the programmer with a reasonable amount of information that can be examined as a unit. Relationships such as regular repetitions of data patterns or offsets of entire arrays may become obvious.**

A general dump can be difficult to write. The programmer should be careful of the following situations:

1. The size of the memory area exceeds 256 bytes, so that an 8-bit counter will not suffice.
2. The ending address is below the starting address. This can be treated as an error, since the user would seldom want to print the contents of memory in an unusual order.

Since the speed of the Memory Dump depends on the speed of the output device, the efficiency of the routine seldom matters. **The following program will ignore cases where the ending address is below the starting address, and will handle areas of any size.** We assume that the starting address is in memory locations START and START + 1 and the ending address is in memory locations LAST and LAST + 1.

```

*
*PRINT CONTENTS OF MEMORY LOCATIONS BETWEEN START AND LAST
*
DUMP  LDX  START  GET STARTING ADDRESS
CHKEND CMPX  LAST  ARE WE BEYOND ENDING ADDRESS?
      BHI  DONE   YES, DUMP COMPLETED
      LDA  ,X+    NO, GET CONTENTS OF NEXT LOCATION
      JSR  PRNT1  PRINT CONTENTS AS 2 HEX DIGITS
      BRA  CHKEND
DONE  RTS

```

Figure 19-4 shows the output from a dump of memory locations 1000 through 101F.

This routine correctly handles the case in which the starting and ending addresses are the same (try it!). You must interpret the results carefully if the dump area includes the stack, since the dump subroutine itself uses the stack. PRNT1 may also change memory and stack locations.

A memory dump can display the data in many different formats. Common alternatives are ASCII characters or pairs of hexadecimal digits for 8-bit values and four hexadecimal digits for 16-bit values. You should select a format based on how you plan to use the dump. If the area of memory contains object code, a hexadecimal format will be best, since you can look up the meanings of the operation codes in Appendix D or on a standard summary card. The following example shows a common format for displaying the output of a dump; since this approach provides both the hexadecimal and the ASCII forms, you can use it to examine areas containing either object code or ASCII text.

1000 54 68 65 20 64 75 6D 70 The dump

23	1F	60	54	37	28	3E	00
6E	42	38	17	59	44	98	37
47	36	23	81	E1	FF	FF	5A
34	ED	BC	AF	FE	FF	27	02

Figure 19-4. Results of a Typical Memory Dump

Each line consists of three parts: a starting address (the address of the first byte shown on the line), the contents of that address and the following seven or fifteen bytes in hexadecimal form, and the ASCII representation of those contents. You might try revising the memory dump program so it produces output in this format.

ADVANCED DEBUGGING TOOLS

Popular advanced debugging tools include:

- **Simulator programs** that help in checking program logic.
- **Logic or microprocessor analyzers** that help in checking timing and other hardware-related factors.

Many variations of both these tools exist; we shall discuss only the standard features.

SOFTWARE SIMULATOR

The **simulator** is the computerized equivalent of the pencil-and-paper computer. It is a **computer program that goes through the operating cycle of another computer, keeping track of the contents of all the registers, flags, and memory locations.** We could, of course, do this by hand, but it would require a large amount of effort and close attention to the exact effects of each instruction. The simulator program never gets tired or confused, forgets an instruction or register, or runs out of paper.

Most simulators are large FORTRAN programs. They can be purchased or used on the time-sharing service. The 6809 simulator is available in several versions from different sources.

Typical Features

Typical simulator features are:

1. **A breakpoint facility.** Usually, breakpoints can be set after a particular number of cycles have been executed, when a memory location or one of a set of memory locations is referenced, when the contents of a location or one of a set of locations are altered, or on other conditions.
2. **Register and memory dump facilities** that can display the values of memory locations, registers, and I/O ports.
3. **A trace facility** that will print the contents of particular registers or memory locations whenever the program changes or uses them.
4. **A load facility** that allows you to set values initially or change them during the simulation.

Some simulators can also simulate input/output, interrupts, and even DMA.

Advantages

The simulator has many advantages:

1. It can provide a complete description of the status of the computer, since it is not restricted by pin limitations or other characteristics of the underlying circuitry.
2. It can provide breakpoints, dumps, traces, and other facilities, without using any of the processor's memory space or control system. These facilities will therefore not interfere with the user program.
3. Programs, starting points, and other conditions are easy to change.
4. All the facilities of a large computer, including peripherals and software, are available to the microprocessor designer.

Limitations

On the other hand, the simulator is limited by its software base and its separation from the real microcomputer. The major limitations are:

1. The simulator cannot help with timing problems, since it operates far more slowly than real time and does not model actual hardware or interfaces.
2. The simulator cannot fully model the input/output section.
3. The simulator is usually quite slow. Reproducing one second of actual processor time may require hours of computer time. Using the simulator can be quite expensive.

The simulator represents the software side of debugging; it has the typical advantages and limitations of a wholly software-based approach. The simulator can provide insight into program logic and other software problems, but cannot help with timing, I/O, and other hardware problems.

LOGIC ANALYZER

The logic or microprocessor analyzer is the hardware approach to debugging. Basically, the analyzer is the parallel digital version of the standard oscilloscope. The analyzer displays information in binary, hexadecimal, or mnemonic form on a CRT, and has a variety of triggering events, thresholds, and inputs. Most analyzers also have a memory so that they can display the past contents of the busses.

The standard procedure is to set a triggering event, such as the occurrence of a particular address on the Address Bus or instruction on the Data Bus. For example, one might trigger the analyzer if the microcomputer tries to store data in a particular area or execute an input or output instruction. One may then look at the sequence of events that preceded the breakpoint. **Common problems you can find in this way include short noise spikes (or glitches), incorrect signal sequences, overlapping waveforms, and other timing or signaling errors.** You could not diagnose those errors with a software simulator any more than you could conveniently find errors in program logic with a logic analyzer.

Important Features

Logic analyzers vary in many respects. Some of these are:

1. **Number of input lines.** At least 24 are necessary to monitor an 8-bit Data Bus and a 16-bit Address Bus. Still more are needed for control signals, clocks, and other important inputs.
2. **Amount of memory.** Each previous state that is saved will occupy several bytes.
3. **Maximum frequency.** It must be several MHz to handle the fastest processors.
4. **Minimum signal width** (important for catching glitches).
5. **Type and number of triggering events allowed.** Important features are pre- and post-trigger delays. These allow the user to display events occurring before or after the trigger event.
6. **Methods of connecting to the microcomputer.** This may require a complex interface.
7. **Number of display channels.**
8. **Binary, hexadecimal, or mnemonic displays.**
9. **Display formats.**
10. **Signal-hold time requirements.**
11. **Probe capacitance.**
12. **Single or dual thresholds.**

All these factors are important in comparing different logic and microprocessor analyzers, since these instruments are new and unstandardized. A tremendous variety of products is already available and this variety will become even greater in the future.¹

Logic analyzers, of course, are necessary only for systems with complex timing. Simple applications with low-speed peripherals have few hardware problems that a designer cannot handle with a standard oscilloscope.

DEBUGGING WITH CHECKLISTS

The designer cannot possibly check an entire program by hand; however, there are certain trouble spots that the designer can easily check. **You can use systematic hand checking to find many errors before you start using debugging tools. The question is where to place the effort. The answer is on points that can be handled with either a yes-no answer or with a simple arithmetic calculation.** Do not try to do complex arithmetic, follow all the flags, or try every conceivable case. Limit your hand checking to matters that can be settled easily. Leave the complex problems to be solved with the aid of debugging tools. But proceed systematically, build your checklist, and make sure that the program performs the basic operations correctly.

WHAT TO CHECK BY HAND

The first step is to compare the flowchart or other program documentation with the actual code. Make sure that everything that appears in one also appears in the other. A simple checklist will do the job. It is easy to completely omit a branch or processing section.

Next concentrate on the program loops. Make sure that all registers and memory locations used inside the loops are initialized correctly. This is a common source of errors; once again, a simple checklist will suffice.

Now look at each conditional branch. Select a sample case that should produce a branch and one that should not; try both of them. Is the branch correct or inverted? If the branch involves checking whether a number is above or below a threshold, try the equality case. Does the correct branch occur? Make sure that your choice is consistent with the problem definition.

Look at the loops as a whole. Try the first and last iterations by hand; these are often troublesome special cases. What happens if the number of iterations is zero, i.e., there is no data or the table has no elements? Does the program fall through correctly? Programs will often perform one iteration unnecessarily, or even worse, decrement counters past zero before checking them.

Check off everything down to the last statement. Don't hopefully assume that the first error is the only one in the program. Hand checking will allow you to get the maximum benefit from debugging runs, since you will get rid of many simple errors ahead of time.

A quick review of hand checking questions:

1. Is every element of the program design in the program (and vice versa for documentation purposes)?
2. Are all registers and memory locations used inside loops initialized before the loops are entered?
3. Are all conditional branches logically correct?
4. Do all loops start and end properly?
5. Are equality cases handled correctly?
6. Are trivial cases handled correctly?

LOOKING FOR ERRORS

Of course, despite all these precautions (or if you skip over some of them), programs often still don't work. The designer is left with the problem of how to find the mistakes. The hand checklist provides a starting place if you didn't use it earlier.

PROGRAMMER ERRORS

Here are some of the errors that you may not have eliminated using the checklist:

1. Failure to initialize variables such as counters, pointers, sums, indexes, etc. Do not assume that registers, memory locations, or flags necessarily contain zero before they are used.

2. **Inverting the logic of a conditional jump**, such as using Branch on Carry Set when you should use Branch on Carry Clear. Remember the effects of comparison (CMP) and subtraction (SBC or SUB) instructions, since these are the most common flag-setting operations. If A is the contents of Accumulator A and M the contents of the effective address, CMPA (or SUBA) sets the Carry and Zero flags as follows:

Zero flag = 1 if A = M
 Zero flag = 0 if A \neq M
 Carry flag = 1 if A < M } Assuming unsigned operands
 Carry flag = 0 if A \geq M }

Note that the Carry flag is cleared in the equality case (A = M). So Branch on Carry Set causes a branch if A < M and Branch on Carry Clear causes a branch if A \geq M. If you want to handle the equality case in the opposite way, use Branch if Lower or Same (causes a branch if A \leq M) or Branch if Higher (causes a branch if A > M). For example, if you want to force a branch when A is greater than or equal to 10, use

CMPA #10
 BCC ADDR

The mnemonic BHS (Branch if Higher or Same) would be clearer than BCC in this case; both mnemonics represent the same instruction. On the other hand, if you want to force a branch when A is strictly greater than 10, use

CMPA #10
 BHI ADDR

3. **Updating counters, pointers, and indexes in the wrong place or not at all.**
 Be sure that there are no paths through a loop that either skip or repeat the updating instructions. Note the difference between the 6809's autoincrementing and autodecrementing:
 In autoincrementing, the processor increments the index register or stack pointer *after* using its contents.
 In autodecrementing, the processor decrements the index register or stack pointer *before* using its contents.
4. **Failure to handle trivial cases correctly.**
 Such cases may involve no data in a buffer, no tests to be run, or no entries in an array or table. Do not assume that such cases will never occur unless the program eliminates them specifically. Trivial cases often cause problems if you use FORTRAN-like loop structures (see Figure 5-2) which execute a routine once before checking conditions.
5. **Reversing order of operands.**
 Remember that TFR R1,R2 moves the contents of R1 to R2, not the other way around.
6. **Changing condition flags before you use them.**
 Almost all instructions affect the Negative and Zero flags. Remember also that RTI and TFR R1,CC change all the flags, while LEAX and LEAY change the Zero flag.
7. **Confusing the index registers and the indexed memory address.**
 Remember that CLR ,X clears the memory location addressed by Index

Register X, not Index Register X itself. Note the difference between INC ,X and INX (or LEAX 1,X); the former adds 1 to the contents of an 8-bit memory location (addressed by Index Register X) while the latter adds 1 to the contents of a 16-bit index register.

8. Confusing data and addresses.

Remember that LDX # $\$1000$ loads Index Register X with the number 1000_{16} , whereas LDX $\$1000$ loads Index Register X with the contents of memory locations 1000_{16} and 1001_{16} . A similar distinction applies to LDA COUNT and LDA #COUNT. This problem becomes more serious if you are using the indirect addressing modes. Now you must remember that LDA ,X+ loads Accumulator A from the address in Index Register X and then adds 1 to Index Register X, whereas LDA [,X+ +] loads Accumulator A from the address contained in the two memory bytes starting at the address in Index Register X and then adds 2 to Index Register X. Mathematical descriptions of what is happening are shorter and more meaningful than word descriptions, but either may be difficult to understand.

9. Accidentally reinitializing a register or memory location.

Make sure that no branches transfer control back into the initialization routine. Calculating a result and then writing over it is a common error that is difficult to trace.

10. Confusing numbers and characters.

Remember that the ASCII representation of a digit is not the same as the binary or BCD representation. For example, the ASCII representation of 7 is 37_{16} ; 07_{16} is the ASCII BELL character which rings the bell on a teletypewriter.

11. Confusing binary and decimal numbers.

In the BCD representation, each decimal digit is coded separately into binary. This is not true in the binary representation, since ten is not an integral power of 2. For example, the decimal number 54 is equal to 36_{16} in the binary representation and 54_{16} in the standard BCD representation.

12. Reversing the order of the data in non-commutative operations like subtraction and division.

Remember that SUB and CMP both subtract the contents of the effective address *from* the contents of the specified register.

13. Ignoring the effects of subroutines and macros.

Subroutine calls and references to macros typically result in the execution of many instructions. Those instructions will almost always change the flags and may change registers or memory locations as well. Be sure that you know the effects of any subroutine or macro you use. Note also the importance of documenting subroutines and macros so users can determine their effect without examining a long listing.

14. Using the Shift instructions improperly.

Remember the precise effects of ASR, ASL, LSR, ROL, and ROR. They are 1-bit shifts that affect all the flags. ASL and LSR both clear the empty bit, whereas ASR preserves the sign bit. ROR and ROL are circular shifts that include the Carry. Remember that shift instructions affect all the flags, even if they are operating on the data in a memory location.

15. Counting the length of an array incorrectly.

Remember that addresses 0300 through 0304 include five (not four) memory locations.

16. Confusing 8- and 16-bit quantities.

The Accumulators, Condition Code register, and Direct Page register are all 8 bits long, whereas the Index Registers, Stack Pointers, and Program Counter are 16 bits long. You cannot move data between registers of different lengths using the transfer (TFR) or exchange (EXG) instructions.

17. Forgetting that addresses or 16-bit data occupy two bytes of memory.

Extended or indirect addresses or 16-bit data occupy two memory locations. The 16-bit registers also occupy two memory locations when they are stored in memory. For example, LDX \$40 loads Index Register X from memory locations 0040 and 0041. Similarly, STU \$50 stores the User Stack Pointer in locations 0050 and 0051. Note that CMPX, CMPU, CMPY, CMPS, LDX, LDU, LDY, LDS, etc. can all use 8-bit direct page addresses, even though they are 16-bit operations.

18. Confusing the Stacks and their pointers.

Instructions like LD, TFR, LEA, and EXG affect the Stack Pointers, not the contents of the Stacks. PSH and PUL transfer data to and from the Stacks. Remember that JSR, BSR, RTI, RTS, and SWI all use the Hardware Stack. Remember also that you must initialize the Hardware Stack Pointer before calling any subroutines or allowing any interrupts.

19. Changing a register or memory location before using it.

Remember that LD, ST, and TFR all change the contents of the destination, but not the source. EXG, on the other hand, changes both of its operands (assuming they are not the same).

20. Forgetting to transfer control past sections of the program that should not be executed.

Remember that the computer will proceed sequentially through memory unless specifically instructed to do otherwise. Thus you may need some unconditional branches to avoid routines that should not be executed.

21. Changing registers that you are using for addressing.

Be particularly careful of instructions like LDA A,X which loads Accumulator A using the index in Accumulator A. The index in Accumulator A is destroyed, so you had better not need it again. Instructions that use a register both for addressing and as a destination can be powerful, but may also be confusing.

22. Ignoring the effects of autoincrementing and autodecrementing. Instructions that use these modes change the specified index register or stack pointer.

23. Ignoring the physical limitations of I/O devices and interface chips.

While we may address interface chips as if they were memory locations, they may not behave like memories. Storing data in an input port seldom makes sense, nor does loading data from an output port unless the port is latched and buffered. In particular, be careful of instructions like shifts, clear, complement, and test which both read and write a memory location. They may have unpredictable effects on I/O ports and interface chips.

24. Ignoring the limitations of read-only memory.

Clearly instructions that both read and write a memory location make little sense when applied to a ROM address.

25. Forgetting that the Hardware Stack is used in subroutine linkages.

JSR or BSR saves the return address in the Hardware Stack on top of any parameters you may have placed there. RTS simply transfers control to the address at the top of the Hardware Stack; if you have not managed the Stack properly, the computer could end up anywhere.

26. Using the single accumulators and the double accumulator inconsistently.

The double accumulator D is physically the same as Accumulator A (MSB's) and Accumulator B (LSB's). Double accumulator instructions are convenient, but you must be sure that they mesh with the single accumulator instructions.

27. Forgetting that addressing modes operate differently on Jump instructions than on other instructions.

Jump instructions (JMP or JSR) are executed as if one level of indirection had been removed. For example, JMP \$A000 loads the address $A000_{16}$ into the Program Counter, whereas LDX \$A000 loads the contents of addresses $A000_{16}$ and $A001_{16}$ into Index Register X. The equivalent JMP instruction would be JMP [A000]. A similar distinction applies to all the indexed modes; JMP or JSR using an indirect mode has the same effect on the Program Counter that LDX using the corresponding non-indirect mode would have on Index Register X. This distinction makes instructions that use indirect addressing even more confusing than they would normally be.

28. Using the wrong register.

A and B are close together in the alphabet and easy to confuse. So are X and Y. Even D, DP, S, and U can get into the act if you forget what is where. One-letter typing errors are easy to make and difficult to notice when the result is legal.

ASSEMBLER-RELATED ERRORS

The use of an assembler is the only practical way to convert source programs into object code, but it does introduce a few annoying errors. In particular,

- 1. Be careful of what your assembler may use as defaults.** For example, the standard 6809 assembler assumes that unmarked numbers are decimal and that instructions without designated addressing modes use direct addressing (if on the direct page) or extended addressing (otherwise). You must specify hexadecimal or binary numbers, ASCII characters, immediate addresses, indexed addresses, or indirect addresses, if you want to use them.
- 2. Watch for simple typing errors that can produce legal instructions or that can confuse the assembler completely.** Many operation codes differ by a single letter (e.g., ADCA, ADDA, and ADDD); you can easily make a typing error and still have a legal program. Some assemblers get confused if you insert too many spaces, too much punctuation, or meaningless characters like 1/2 or ¢; in fact, the assembler may object to a minor error, but accept a totally illogical entry that its developer never considered.

Remember, the assembler can print a reassuring message like “**NO ASSEMBLY ERRORS**” even when the program is wrong. All the message means is that the assembler found no errors according to its interpretation of the rules of the language. This does not exclude errors that produce legal instructions or that are beyond the assembler’s comprehension. It certainly does not mean that the program does what you intended.

INTERRUPT-DRIVEN PROGRAMS

Interrupt-driven programs are particularly difficult to debug, since errors may show up only when an interrupt occurs at a particular time. If, for example, the program enables the interrupts a few instructions too early, an error will appear only if an interrupt occurs while the processor is executing those few instructions. In fact, you can usually assume that sporadic or random errors are caused by the interrupt system, since the rest of the system can be reproduced.² **Typical errors in interrupt-driven programs are:**

1. **Forgetting to reenable interrupts after accepting one and clearing it.**
The processor disables the interrupt system automatically on RESET or on accepting an interrupt. Be sure that no possible sequences fail to reenable the interrupt.
2. **Forgetting that RTI automatically reenables the interrupt unless you specifically set the Interrupt Masks in the Stack.**
3. **Enabling interrupts before initializing all system parameters, such as flags, vectors, and priority registers.**
A checklist can help here.
4. **Leaving results in registers and then destroying them by executing RTI.**
Remember that RTI restores all the registers from the stack. As we noted in Chapter 15, you should not use the registers to pass parameters and results between the main program and the interrupt service routine.
5. **Forgetting that the interrupts (including SWI) save the registers in the Stack whether you want them or not.**
You may have to reinitialize or update the Hardware Stack Pointer.
6. **Failing to clear the interrupt before exiting from the service routine.**
If the interrupt comes from a PIA, the service routine must read the data register in order to clear the interrupt flag. The reading is necessary even if the interrupt is from an output device or a real-time clock. Otherwise, the interrupt will remain active and will be recognized again as soon as the processor reenables it.
7. **Not disabling the interrupt during multi-byte transfers.**
Watch particularly for routines that update time, position, or other data that the interrupt service routine uses. You must avoid situations in which partial updating results in erroneous values.
8. **Failing to reenable the interrupt after executing a routine that requires the interrupts to be disabled.**
Be especially careful of such routines if they may be entered with the interrupts disabled or enabled. The routine must then save and restore the

condition code register, so that it exits with the interrupt system in its original state.

Other Approaches

These lists are far from complete, but they should suggest some places where you can look for errors. Unfortunately, debugging computer programs is not an exact science; even the most systematic approach can leave you with baffling problems.³ Sometimes, your best bet may be to let the problem sit overnight or have someone with a fresh viewpoint look at it.

PROGRAM EXAMPLES

19-1. DEBUGGING A CODE CONVERSION PROGRAM

The program converts a decimal number in memory location 0040 to a seven-segment code in memory location 0041. It blanks the display if memory location 0040 does not contain a decimal number.

Initial Program (from Flowchart in Figure 19-5):

```

LDA    $40          GET DATA
CMPA   #9           IS DATA A DECIMAL DIGIT?
BCS   DONE         NO, DONE
LDX   SSEG         YES, GET BASE ADDRESS OF CODE TABLE
LDA   ,X           GET ELEMENT FROM TABLE
DONE  STA   $41     SAVE SEVEN-SEGMENT CODE
      SWI
SSEG  FCB   $3F,$06,$5B,$4F,$66
      FCB   $6D,$7D,$07,$7D,$6F

```

Using the Checklist

Using a checklist as described earlier in this chapter, we were able to find the following errors:

1. We have omitted the section that clears Result if the data is not a decimal digit.
2. The conditional branch (BCS DONE) is incorrect.

For example, if the data is zero, CMPA #9 clears the Carry flag and causes a branch. The correct version is

```

CMPA   #9           IS DATA GREATER THAN 9?
BHI   DONE         YES, DONE

```

If we had used the mnemonic BLO instead of BCS, the mistake might have been more obvious (or perhaps never made in the first place). You can clarify code by using the mnemonics BLO and BHS after comparisons instead of BCS and BCC.

The 6809 has many conditional branches and you must be careful to choose the right one.

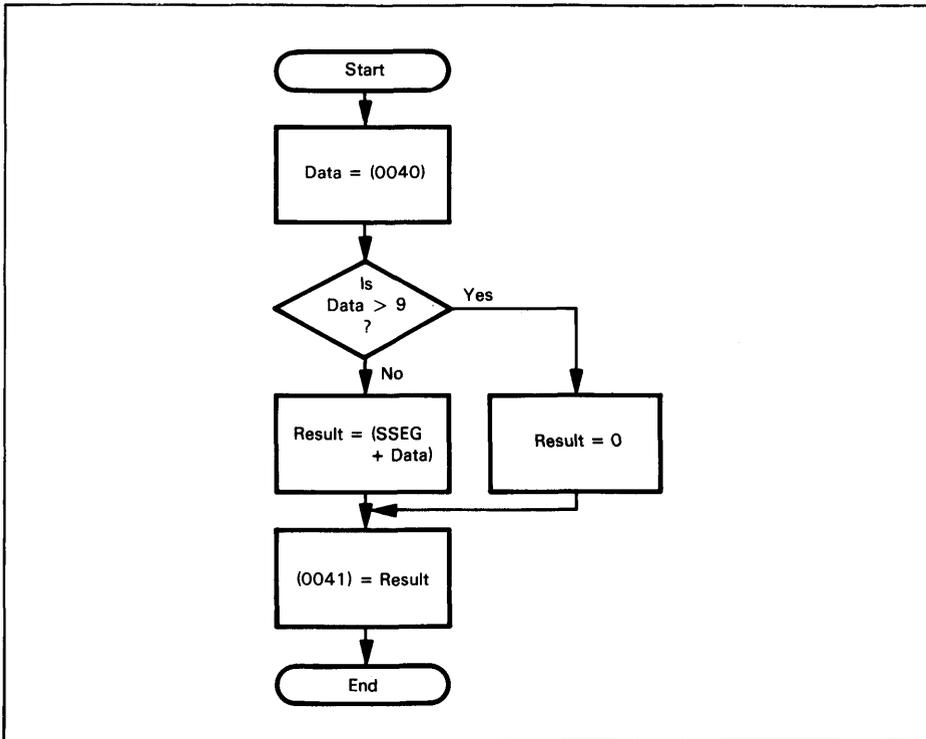


Figure 19-5. Flowchart of Decimal to Seven-Segment Conversion

Second Program:

```

CLR B          GET BLANK CODE FOR DISPLAY
LDA  $40       GET DATA
CMPA #9        IS DATA A DECIMAL DIGIT?
BHI  DONE      NO, KEEP ERROR CODE
LDX  SSEG      YES, GET BASE ADDRESS OF CODE TABLE
LDA  ,X        GET ELEMENT FROM TABLE
DONE STA  $41   SAVE SEVEN-SEGMENT CODE
      SWI
SSEG FCB  $3F,$06,$5B,$4F,$66
      FCB  $6D,$7D,$07,$7D,$6F
  
```

The hand check did not uncover any errors in this version.

Single Step

Since the program is simple, the next stage is to single-step through it with real data. We chose the following data for the trials:

- 0 The smallest decimal digit
- 9 The largest decimal digit
- 10 A boundary case
- 6B₁₆ A randomly selected case

For the first trial, we placed zero in memory location 0040. The program proceeded with no apparent errors until it reached the LDA ,X instruction. At that point, Index Register X contained 3F06, an address that did not even exist in our computer. Clearly, something had gone wrong.

Hand Check

It was now time for more hand-checking. Since we knew that BHI DONE was correct, the error had to be further along in the program. The hand check showed that LDX SSEG placed 3F06 in Index Register X, since it loaded the register with the contents of the two bytes starting at address SSEG. What we want to place in Index Register X is the address SSEG, not its contents; that is, we want immediate addressing, not direct addressing. This change creates an awkward patching problem in the object code, since LDX with immediate addressing occupies 3 bytes of memory, whereas LDX with direct addressing occupies only 2 bytes.

Run Test

With this correction (LDX #SSEG instead of LDX SSEG), the program worked correctly when the data was zero. However, when the data was 9, it produced the same result as for 0 (3F₁₆). A hand check of the LDA ,X instruction showed that the program was not performing any indexing; it was just loading Accumulator A from the address in Index Register X. What we want is the accumulator indexed mode in which the processor adds the index in Accumulator A to the base address in Index Register X. So we replaced LDA ,X with LDA A,X.

Third Program:

	CLRB		GET BLANK CODE FOR DISPLAY
	LDA	\$40	GET DATA
	CMPA	#9	IS DATA A DECIMAL DIGIT?
	BHI	DONE	NO, KEEP ERROR CODE
	LDX	#SSEG	YES, GET BASE ADDRESS OF CODE TABLE
	LDA	A,X	GET ELEMENT FROM TABLE
DONE	STA	\$41	SAVE SEVEN-SEGMENT CODE
	SWI		
SSEG	FCB	\$3F,\$06,\$5B,\$4F,\$66	
	FCB	\$6D,\$7D,\$07,\$7D,\$6F	

The results now were:

Data	Result
00	3F
09	6F
0A	0A
6B	6B

Another Run Test

The program was not clearing the result if the data was invalid (i.e., greater than 9). In fact, the program never used the blank code in Accumulator B at all. The required change is to load the seven-segment code into Accumulator B instead of Accumulator A (replace LDA A,X with LDB A,X) and store Accumulator B instead of Accumulator A (replace STA \$41 with STB \$41). After we made these corrections, the program produced the correct results for all the test cases.

Exhaustive Test

Since the program was simple, we could easily test it on each decimal digit. The results were:

Data	Result
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7D
9	6F

Note that the result for 8 is wrong — it should be 7F, not 7D. Since the program works for all other digits, the error is almost surely in the table. In fact, the eighth entry in the table had been typed incorrectly.

Final Program:

```

*
*DECIMAL TO SEVEN-SEGMENT CONVERSION
*
      CLRB          GET BLANK CODE FOR DISPLAY
      LDA   $40      GET DATA
      CMPA  #9       IS DATA A DECIMAL DIGIT?
      BHI  DONE      NO, KEEP ERROR CODE
      LDX  #SSEG     YES, GET BASE ADDRESS OF CODE TABLE
      LDB  A,X       GET ELEMENT FROM TABLE
DONE  STB  $41       SAVE SEVEN-SEGMENT CODE
      SWI
SSEG  FCB  $3F,$06,$5B,$4F,$66
      FCB  $6D,$7D,$07,$7F,$6F

```

Summary of Errors Discovered

The errors that we found in this example are typical of the ones that 6809 assembly language programmers should expect. They include:

1. Failing to initialize registers or memory locations.
2. Inverting the logic on conditional branches.
3. Branching incorrectly in boundary cases.
4. Confusing immediate and direct addressing (i.e., data and addresses).
5. Failing to keep track of the current contents of registers and therefore using the wrong accumulator, index register, or stack pointer.
6. Using the indexed addressing modes incorrectly. The 6809 terminology can be confusing, since the 16-bit index registers usually hold base addresses, not indexes.
7. Copying lists of numbers (or instructions) incorrectly.

Note that straightforward instructions (like AND, DEC, or INC) and simple addressing modes seldom cause any problems.

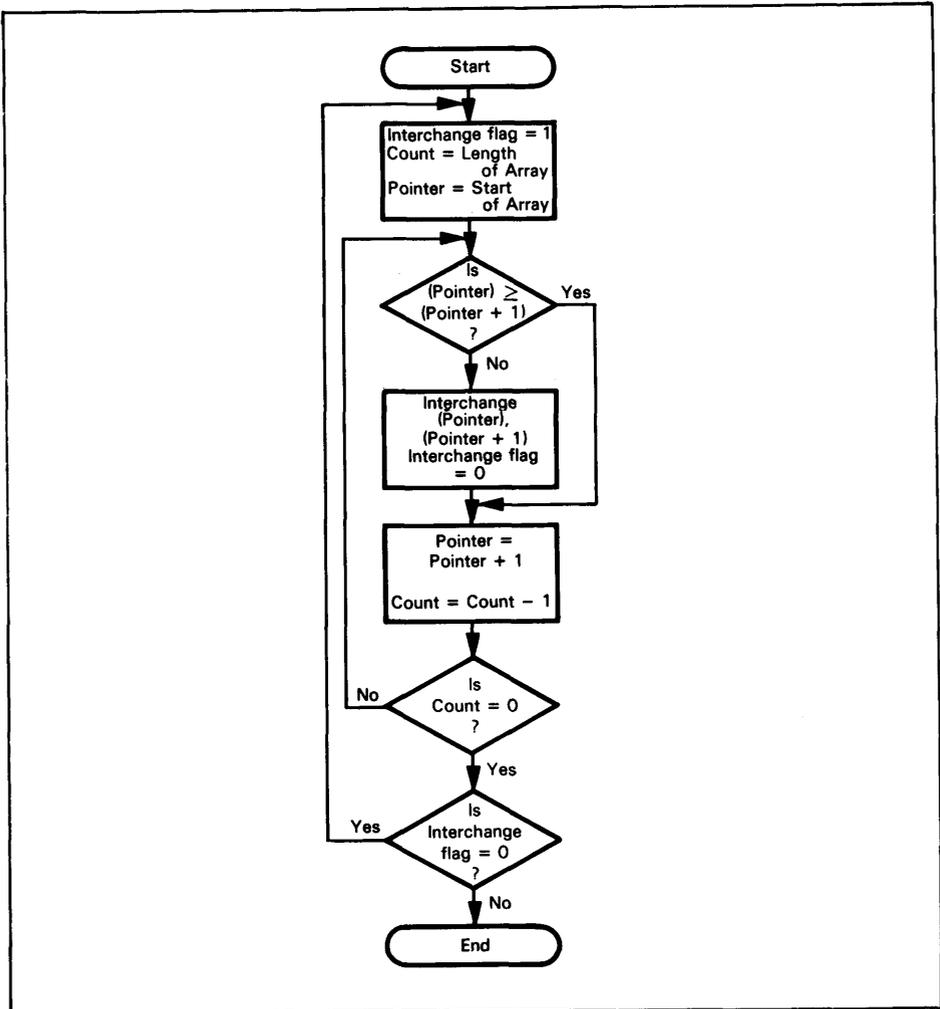


Figure 19-6. Flowchart of a Sort Program

19-2. DEBUGGING A SORT PROGRAM

The program sorts an array of unsigned 8-bit binary numbers into decreasing order. The array begins in memory location 0042 and its length is in memory location 0041.

Initial Program (from flowchart in Figure 19-6):

	LDA	#1	INTERCHANGE FLAG = 1
	STA	\$40	
	LDA	\$41	COUNT = LENGTH OF ARRAY
	LDX	#\$42	POINT TO START OF ARRAY
PASS	LDB	,X	GET AN ELEMENT
	CMPB	1,X	IS PAIR IN CORRECT ORDER?
	BLO	COUNT	YES, NO INTERCHANGE
	STB	1,X	NO, INTERCHANGE PAIR
COUNT	DECA		IS PASS THROUGH ARRAY COMPLETE?
	BNE	PASS	NO, GO ON TO NEXT PAIR
	TST	\$40	YES, WERE ANY INTERCHANGES PERFORMED?
	BNE	PASS	YES, MAKE ANOTHER PASS
	SWI		

Initial Hand Check

A hand check shows that we have implemented all the blocks in the flowchart and initialized all the registers and memory locations. We must examine the conditional branches carefully. The instruction BLO COUNT must force a branch if the pair is already in the correct order — that is, if the second element is less than or equal to the first element. The program must not interchange equal elements, since such an interchange would create an endless loop with each pass swapping elements.

Try an example:

(0042) = 30
(0043) = 37

The execution of CMPB 1,X causes the CPU to calculate 30 - 37. The Carry flag is set since the subtraction requires a borrow. This example should result in an interchange, but BLO COUNT branches around the interchange instructions. BHS COUNT produces the proper branch in this case. If the two numbers are equal, the comparison will clear the Carry flag so BHS COUNT is again correct.

How about BNE PASS at the end of the program? If there are any elements out of order, the program will clear the interchange flag and the contents of memory location 0040 will be zero. So the branch is inverted; it should be BEQ PASS.

Now let us check the first iteration by hand. The initialization (the first four instructions) produces the following values:

(A) = COUNT	Length of array
(X) = 0042	Starting address of array
(0040) = 1	Interchange flag

The effects of the instructions in the loop are:

LDB	,X	(B) = (0042)
CMPB	1,X	(0042) - (0043)
BHS	COUNT	
STB	1,X	(0043) = (0042)
DECA		(A) = COUNT - 1

Note that we have already checked the conditional branch instructions.

Clearly the logic is incorrect. If the first two numbers are out of order (as in our example), the results after the first iteration should be:

(0042) = old (0043)
 (0043) = old (0042)
 (X) = 0043
 (A) = COUNT - 1

Instead, they are:

(0042) = Unchanged
 (0043) = old (0042)
 (X) = 0042
 (A) = COUNT - 1

The error in Index Register X is easy to correct. We can use autoincrementing as long as we remember to adjust the later indexed offsets. Thus we need LDB ,X+; instead of LDB ,X; CMPB ,X instead of CMPB 1,X (the autoincrementing has increased Index Register X by 1); and STB ,X instead of STB 1,X. We must be careful to increment X in an instruction that will be executed regardless of the outcome of BHS COUNT. The interchange requires a bit more care and the use of both Accumulators (remembering to save the count in the Hardware Stack and restore it at the end):

```

PSHS A          SAVE COUNT IN HARDWARE STACK
LDA  ,X         GET SECOND ELEMENT OF PAIR
STB  ,X         REPLACE SECOND ELEMENT WITH FIRST ELEMENT
STA  -1,X      REPLACE FIRST ELEMENT WITH SECOND ELEMENT
PULS A         RESTORE COUNT FROM HARDWARE STACK
    
```

An interchange always requires a temporary storage place in which the program can save one element while it is transferring the other one.⁴

Second Program:

```

LDA  #1         SET INTERCHANGE FLAG
STA  $40
LDA  $41        COUNT = LENGTH OF ARRAY
LDX  #$42      POINT TO START OF ARRAY
PASS LDB ,X+    IS PAIR OF ELEMENTS IN ORDER?
      CMPB ,X
      BHS COUNT
      PSHS A          NO, INTERCHANGE ELEMENTS
      LDA  ,X
      STB  ,X
      STA  -1,X
      PULS A
COUNT DECA    IS PASS THROUGH ARRAY COMPLETE?
      BEQ  PASS      NO, GO ON TO NEXT PAIR
      TST  $40       WERE ANY INTERCHANGES PERFORMED?
      BNE  PASS      YES, MAKE ANOTHER PASS
      SWI
    
```

How about the last iteration? Let us assume that the array contains three elements:

(0041) = 03 Number of elements
 (0042) = 02 First element
 (0043) = 04 Second element
 (0044) = 06 Third element

Each time through the loop, the program increments Index Register X by 1. So, at the start of the third (last) iteration,

(X) = 0042 + 2 = 0044

The effects of the instructions in the loop are:

LDB ,X+ (B) = (0044), (X) = 0045
 CMPB ,X (0044) - (0045)

This is incorrect; the program is working on data beyond the end of the array. In fact, the previous iteration should have been the last one, since the number of pairs is one less than the number of elements. The last element in the array has no successor for comparison. The correction is to reduce the number of iterations by 1: i.e., place DECA after LDA \$41.

Checking Trivial Cases

What happens in the trivial cases — that is, if the array contains no elements or only one element? The answer is that the program does not work correctly and could change a large number of memory locations improperly and without any warning (try it!). The changes that handle the trivial cases are simple but essential; the cost is only a few bytes of memory to avoid problems that could be difficult to identify and correct.

Third Program:

```

LDA $41          GET LENGTH OF ARRAY
CMPA #1          IS THERE MORE THAN ONE ELEMENT?
BLS DONF        NO, NO ACTION NECESSARY
LDA #1          SET INTERCHANGE FLAG
STA $40
LDA $41          GET LENGTH OF ARRAY
DECA           NUMBER OF PAIRS = LENGTH - 1
LDX #$42        POINT TO START OF ARRAY
PASS LDB ,X+     IS PAIR OF ELEMENTS IN ORDER?
      CMPB ,X
      BHS COUNT
      PSHS A          NO, INTERCHANGE ELEMENTS
      LDA ,X
      STB ,X
      STA -1,X
      PULS A
COUNT DECA     IS PASS THROUGH ARRAY COMPLETE?
      BNE PASS      NO, GO ON TO NEXT PAIR
      TST $40       WERE ANY INTERCHANGES PERFORMED?
      BEQ PASS      YES, MAKE ANOTHER PASS
DONE  SWI

```

Run Test With Breakpoints

Now we must check the program on the computer or on the simulator. A simple set of data is:

```

(0041) = 02      Length of array
(0042) = 00 }   Array to be sorted
(0043) = 01 }

```

This set consists of two elements in the wrong order. The program should require two passes. The first pass should exchange the elements, producing:

```

(0042) = 01 }   Reordered array
(0043) = 00 }
(0040) = 00      Interchange flag

```

The second element should find the elements already in the proper (descending) order and produce:

```

(0040) = 01      Interchange flag

```

This program is too long for single-stepping, so we will use breakpoints

instead. Each breakpoint will halt the computer and print the contents of the key registers. The breakpoints will come:

1. After LDX #\$42 to check the initialization.
2. After CMPB ,X to check the comparison.
3. After PULS A to check the interchange.
4. After TST \$40 to check the completion of a pass through the array.

The contents of the registers at the first breakpoint are:

Register	Contents
CC	F0
B	00
A	01
X	0042

These are all correct, so the program is performing the initialization properly in this case.

The results at the second breakpoint are:

Register	Contents
CC	F9
B	00
A	01
X	0043

These results are also correct.

The results at the third breakpoint are:

Register	Contents
CC	F1
B	00
A	01
X	0043

Examining memory shows:

(0042) = 01
(0043) = 00

The program has interchanged the elements correctly.

The results at the fourth breakpoint are:

Register	Contents
CC	F0
B	00
A	00
X	0043

Examining memory shows:

(0040) = 01

The Zero flag (bit 2 of the Condition Code Register) is incorrect, since an interchange occurred and the program should branch and go back through the array again. Memory location 0040 (the interchange flag) should contain 0, rather than 1. Examining the program shows that it never clears the interchange flag; the correction is to insert the instruction CLR \$40 after BHS COUNT. The program now clears the interchange flag as soon as it determines that an interchange is necessary.

We can continue by clearing memory location 0040 and setting the Zero flag (CC = F4₁₆ instead of F0₁₆). The results at the second iteration of the second breakpoint are:

Register	Contents
CC	F9
B	00
A	01
X	0044

The program has not reinitialized the registers (particularly Index Register X). The condition branch that sends the program through the entire array again should transfer control to the initialization routine; note that we do not need to check the length of the array a second time to eliminate the trivial cases.

Final Program:

```

LDA $41          GET LENGTH OF ARRAY
CMPA #1         IS THERE MORE THAN ONE ELEMENT?
BLS DONE       NO, NO ACTION NECESSARY
ITER LDA #1     SET INTERCHANGE FLAG
STA $40
LDA $41          GET LENGTH OF ARRAY
DECA          NUMBER OF PAIRS = LENGTH - 1
LDX #$42       POINT TO START OF ARRAY
PASS LDB ,X+    IS NEXT PAIR OF ELEMENTS IN ORDER?
CMPB ,X
BHS COUNT     NO, CLEAR INTERCHANGE FLAG
CLR $40        AND EXCHANGE ELEMENTS
PSHS A
LDA ,X
STB ,X
STA -1,X
PULS A
COUNT DECA    IS PASS THROUGH ARRAY COMPLETE?
BNE PASS      NO, GO ON TO NEXT PAIR
TST $40       WERE ANY INTERCHANGES PERFORMED?
BEQ ITER      YES, MAKE ANOTHER PASS
DONE SWI
    
```

Other Test Cases

Clearly, we cannot check all possible cases for this program. Two other simple test cases that we could use for debugging are:

1. Two equal elements

- (0041) = 02 Number of elements
- (0042) = 00 } Array to be sorted
- (0043) = 00 }

2. Two elements already in descending order

- (0041) = 02 Number of elements
- (0042) = 01 } Array to be sorted
- (0043) = 00 }

REFERENCES

1. For more information about logic analyzers, see:
 - G. Brock. "Logic-State Analyzers Seek Out Microprocessor-System Faults," *EDN*, January 5, 1980, pp. 137-40.
 - R. Lorentzen. "Logic Analyzers Finish What Development Systems Start," *Electronic Design*, March 29, 1980, pp. 81-85.
 - J. Marshall. "Digital Analysis Instruments," *EDN*, January 20, 1980, pp. 141-143.
 - J. McLeod. "Special Report: Logic Analyzers," *Electronic Design*, March 29, 1980, pp. 48-56.
 - C. A. Ogdin. "Setting up a Microcomputer Design Laboratory," *Mini-Micro Systems*, May 1979, pp. 87-94.
 - I. H. Spector and R. Muething. "Logic Analyzer Deploys Its Full Strength," *Electronic Design*, March 29, 1980, pp. 177-214.
2. W. J. Weller. *Assembly Level Programming for Small Computers*, Lexington Books, Lexington, Mass., 1975, Chapter 23.
3. R. L. Baldrige. "Interrupts Add Power, Complexity to Microcomputer System Design," *EDN*, August 5, 1977, pp. 67-73.
4. One way to interchange A and B without using a temporary storage location is to use the formulas:

$$\begin{aligned} A &= A \oplus B \\ B &= A \oplus B \\ A &= A \oplus B \end{aligned}$$

You can verify this sequence if you are handy at Boolean algebra and the use of DeMorgan's theorem.



20

Testing

Program testing¹ is closely related to program debugging. We must test the program on the data that we used to debug it; for example,

- **Trivial cases** such as no data or a single statement
- **Special cases** that the program singles out for some reason
- **Simple cases** that exercise particular parts of the program

For the decimal to seven-segment conversion program in Chapter 19, these cases cover all possible situations. The test data consists of:

- The numbers 0 through 9
- The boundary case 10
- The random case $6B_{16}$

The program does not distinguish any other cases. **Here debugging and testing are virtually the same.**

In the sorting program, the problem is more difficult. The number of elements could range from 0 to 255, and each of the elements could lie anywhere in that range. The number of possible cases is therefore enormous. Furthermore, the program is moderately complex. How do we select test data that will give us a degree of confidence in that program? **Here testing requires some design decisions.** The testing problem is particularly difficult if the program depends on sequences of real-time data. How do we select the data, generate it, and present it to the microcomputer in a realistic manner?

TESTING AIDS

Most of the tools mentioned earlier for debugging are helpful in testing also. Logic or microprocessor analyzers can help check the hardware; simulators² can help check the software. Other tools can also be of assistance:

1. **I/O simulations** that can simulate many devices from a single input and a single output device.
2. **In-circuit emulators** that allow you to attach the prototype to a development system or control panel and test it.³
3. **ROM simulators** that can be changed like RAM but otherwise behave like the ROM or PROM that will be used in the final system.
4. **Real-time operating systems** that can provide inputs or interrupts at specific times (or perhaps randomly) and mark the occurrence of outputs. Real-time breakpoints and traces may also be included.
5. **Emulations** (often on microprogrammable computers) that may provide real-time execution speed and programmable I/O.⁴
6. **Interfaces** that allow another computer to control the I/O system and test the microcomputer program.
7. **Testing programs** that check each branch in a program for logical errors.
8. **Test generation programs** that can generate random data or other distributions.

Formal testing theorems exist, but are only practical for verifying short programs. You must be careful that the test equipment does not invalidate the test by modifying the environment. Often test equipment may buffer, latch, or condition input and output signals. The actual system may not do this and may therefore behave differently.

Furthermore, extra software in the test environment may use some of the memory space or part of the interrupt system. It may also provide error recovery and other features that will not exist in the final system. A software test bed must be just as realistic as a hardware test bed since software failure can be just as critical as hardware failure.

Emulations and simulations are, of course, never precise. They are usually adequate for checking logic, but can seldom help test interfaces or timing. On the other hand, real-time test equipment does not provide much of an overview of the program logic and may affect the interfacing and timing.

SELECTING TEST DATA⁵

Few real programs can be checked for all cases. The designer must choose a sample set that is in some sense representative.

Structured Testing

Testing should, of course, be part of the total development procedure. **Top-down design and structured programming provide for testing as part of the design. This is called structured testing.** Each module within a structured program should be checked separately. **Testing, as well as design, should be modular, structured, and top-down.**

Special Cases

But that leaves the question of selecting test data for a module. The designer must first list all special cases that a program recognizes. These may include:

- Trivial cases
- Equality cases
- Special situations

The test data should include all of these.

Forming Classes of Data

You must next identify each class of data that statements within the program may distinguish. These may include:

- Positive or negative numbers
- Numbers above or below a particular threshold
- Data that does or does not include a particular sequence or character
- Data that is or is not present at a particular time

Be careful; each two-way decision doubles the number of classes since you must test both paths. Thus three conditional branches will result in $2 \times 2 \times 2 = 8$ classes if the computer always executes each branch. Limiting the size of test sets is another important reason to keep modules short and general.

Selecting Data from Classes

You must now separate the classes according to whether the program produces a different result for each entry in the class (as in a table) or produces the same result for each entry (such as a warning that a parameter is above a threshold). In the discrete case, one may include each element if the total number is small or sample if the number is large. The sample should include all boundary cases and at least one case selected randomly. Random number tables are available in books, and random number generators are part of most computer facilities.⁶

You must be careful of distinctions that may not be obvious. For example, the 6809 microprocessor will regard an 8-bit unsigned number greater than 127 as negative; you must consider this when using the branch instructions that depend on the Negative (Sign) flag. You must also watch for instructions that do not affect flags, overflow in signed arithmetic, and the distinctions between address-length (16-bit) quantities and data-length (8-bit) quantities.

EXAMPLES

20-1. TESTING A SORT PROGRAM

The special cases here are obvious:

- No elements in the array
- One element, magnitude may be selected randomly

The other special case to be considered is one in which elements are equal.

There may be some problem here with signs and data length. Note that the array itself must contain fewer than 256 elements. Using the instruction CLR \$40 rather than DEC \$40 to modify the interchange flag means that multiple interchanges will create no special problems.

We could check to see if the sign of the number of elements has any effect by choosing half the test cases with elements between 128 and 255 and half with elements between 2 and 127. We should choose the magnitudes of the elements randomly to avoid unconscious bias which might favor small numbers, decimal (rather than hexadecimal) digits, or regular patterns.

20-2. TESTING AN ARITHMETIC PROGRAM

Here we will presume that a prior validity check has ensured that the number has the right length and consists of valid digits. Since the program makes no other distinctions, test data should be selected randomly. Here a random number table or random number generator will prove ideal; the range of the random numbers is 0 to 9.

RULES FOR TESTING

Sensible design simplifies testing. The following rules can help:

1. **Eliminate trivial cases early** without introducing unnecessary distinctions.
2. **Avoid special cases**, since they increase debugging and testing time.
3. **Perform validity or error checks on the data before it is processed.**
4. **Avoid inadvertent distinctions**, particularly in handling signed numbers or in using instructions that are intended to handle signed numbers.
5. **Check boundary cases by hand.** Be sure to define what should happen in these cases.
6. **Emphasize generality.** Each distinction and separate routine leads to more testing.
7. **Use top-down design and modular programming to modularize testing.**

CONCLUSIONS

Debugging and testing are the stepchildren of the software development process. Most projects leave far too little time for them and most textbooks neglect them. But designers and managers often find that these stages are the most expensive and time-consuming. Progress may be difficult to measure or produce. Debugging and testing microprocessor software is particularly difficult because the powerful hardware and software tools that can be used on larger computers are seldom available for microcomputers.

The designer should plan debugging and testing carefully. We recommend the

following guidelines:

1. **Try to write programs that are easy to debug and test.** Modular programming, structured programming, and top-down design are useful techniques.
2. **Prepare a debugging and testing plan as part of the problem definition.** Decide early what data you must generate and what equipment you will need.
3. **Debug and test each module using top-down design.**
4. **Debug each module's logic systematically.** Use checklists, breakpoints, and the single-step mode. If the program logic is complex, consider using the software simulator.
5. **Check each module's timing systematically if this timing is a problem.** An oscilloscope can solve many problems if you plan the test properly. If the timing is complex, consider using a logic or microprocessor analyzer.
6. **Be sure that the test data is representative.** Watch for any classes of data that the program may distinguish. Include all special and trivial cases.
7. **If the program handles each element differently or the number of cases is large, select the test data randomly.**
8. **Document all tests.** If errors are found later, you will not have to repeat tests you have already run.

REFERENCES

1. G. J. Myers. *The Art of Software Testing*, Wiley, New York, 1979.
R. C. Tausworthe. *Standardized Development of Computer Software*, Prentice-Hall, Englewood Cliffs, N.J., Vol. 1, 1977, Chapter 9; Vol. 2, 1979, Chapters 14 and 15.
E. Yourdon. *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1975, Chapter 7.
2. F. J. Langley. "Simulating Modular Microcomputers," *Simulation*, May 1979, pp. 141-54.
L. A. Leventhal. "Design Tools for Multiprocessor Systems," *Digital Design*, October 1979, pp. 24-26.
F. I. Parke et al. "An Introduction to the N.mPc Design Environment," *Proceedings of the 1979 Design Automation Conference*, San Diego, Ca., pp. 513-19.
3. R. Francis and R. Teitzel. "Realtime Analyzer Aids Hardware/Software Integration," *Computer Design*, January 1980, pp. 140-50.
4. H. R. Burris. "Time-Scaled Emulations of the 8080 Microprocessor," *Proceedings of the 1977 National Computer Conference*, pp. 937-46.
5. R. A. DeMillo et al. "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, April, 1978, pp. 34-41.
W. F. Dalton. "Design Microcomputer Software," *Electronics*, January 19, 1978, pp. 97-101.
6. R. D. Grappel and J. Hemenway. "EDN Software Tutorial: Pseudorandom Generators," *EDN*, May 20, 1980, pp. 119-23.

T. G. Lewis. *Distribution Sampling for Computer Simulation*, Lexington Books, Lexington, Mass., 1975.

R. A. Mueller et al. "A Random Number Generator for Microprocessors," *Simulation*, April 1977, pp. 123-27.

21

Maintenance and Redesign

Program maintenance always involves elements of redesign. A program may not work correctly in the field because of a flaw which was not discovered during the debugging and testing phases of development. Sometimes, however, a program works correctly but inefficiently — taking too long to respond, for example, or requiring an awkward sequence of actions by the operator. A manufacturer may decide to adapt a control program to run in a different hardware configuration. Inevitably, someone will find a use for a microcomputer that never occurred to the system designer; a user's needs often change in unanticipated ways. Thus it may become necessary to change a program or system even if it works correctly.

Sometimes the designer may have to squeeze the last microsecond of speed or the last byte of extra memory out of a program. As larger single-chip memories have become available, the memory problem has become less serious. The time problem, of course, is serious only if the application is time-critical. In many applications the microprocessor spends most of its time waiting for external devices and program speed is not a major factor.

COST OF REDESIGN

Squeezing the last bit of performance out of a program is seldom as important as some writers would have you believe. In the first place, the practice is **expensive for the following reasons:**

1. It requires extra programmer time, which is often the single largest cost in software development.
2. It sacrifices structure and simplicity with a resulting increase in debugging and testing time.
3. The programs require extra documentation.
4. The resulting programs will be difficult to extend, maintain, or re-use.

In the second place, the lower per-unit cost and higher performance may not really be important. Will the lower cost and higher performance really sell more units? Or would you do better with more user-oriented features? **The only applications that would seem to justify the extra effort and time are very high-volume, low-cost and low-performance applications, where the cost of an extra memory chip will far outweigh the cost of the extra software development.** For other applications, you will find that you are playing an expensive game for no reason.

MAJOR OR MINOR REORGANIZATION

However, if you must redesign a program, the following hints will help. First, determine how much more performance or how much less memory usage is necessary. If the required improvement is 25% or less, you may be able to achieve it by reorganizing the program. If it is more than 25% you have made a basic design error; you will need to consider drastic changes in hardware or software. We will deal first with reorganization and later with drastic changes. Reducing memory usage is particularly important if it results in a program that fits in the ROM and RAM provided by a simple one or two-chip microcomputer. The use of such stand-alone microcomputers can reduce hardware costs substantially in limited applications.

SAVING MEMORY

The following procedures will reduce memory usage for assembly language programs:

1. **Replace repetitious in-line code with subroutines.** Be sure, however, that the CALL and RETURN instructions do not offset most of the gain. Note that this replacement usually results in slower programs because of the time spent in transferring control back and forth.
2. **Place the most frequently used data on the direct page and access it with one-byte addresses.** You may even want to place a few I/O addresses there.
3. **Use the Stack when possible.** The Stack Pointer is automatically updated after each use so that no explicit updating instructions are necessary. PSH and PUL can move entire groups of registers to and from memory.
4. **Eliminate Jump instructions.** Try to reorganize the program instead.
5. **Take advantage of addresses that you can manipulate as 8-bit quantities.** These include page zero and addresses that are multiples of 100 hexadecimal. For example, you might try to place all ROM tables in one 100_{16} -byte section of memory, and all RAM variables in another 100_{16} -byte section.
6. **Organize data and tables so that you can address them without worrying about address calculation carries or without any actual indexing.** This will again allow you to manipulate 16-bit addresses as 8-bit quantities.
7. **Use the shift instructions to operate on bit positions at either end of a byte.**

8. **Take advantage of such instructions as ASL, DEC, INC, LSR, ROL, and ROR which operate directly on memory locations without using registers.**
9. **Use INC or DEC to set or reset flag bits.**
10. **Use relative branches rather than jumps with absolute or indexed addressing.**
11. **Use the Software Interrupt instructions RTS and RTI to perform jumps and reach subroutines** if they are not already being used. SWI2 should always be available for this purpose. This approach is particularly helpful if the program uses the Stack anyway for temporary storage of data and addresses.
12. **Watch for special short forms of instructions** that operate directly on the Accumulators or other registers.
13. **Use algorithms rather than tables** to calculate arithmetic or logical expressions and to perform code conversions. This replacement may make the program run slower.
14. **Reduce the size of mathematical tables by interpolating between entries.** Here again, we are saving memory at the cost of execution time.
15. Use instructions like CMPU, CMPX, and CMPY to **perform comparisons without involving the Accumulator.**
16. **Employ double accumulator instructions** such as ADDD, CMPD, LDD, STD, and SUBD rather than pairs of single accumulator instructions.
17. **Take advantage of the LEA instructions** to perform arithmetic as well as to calculate indirect, indexed, and relative addresses for repeated use later.
18. **Use indexed addressing rather than extended addressing to handle PIAs** and other situations involving **several addresses that are close together.**
19. Remember that operations on some of the registers take longer than on others. In particular, some address-length registers have more single-byte operation codes than others; the number is largest for Index Register X, next largest for the Double Accumulator and User Stack Pointer U, and smallest for Index Register Y and the Hardware Stack Pointer. For example, LDX, LDD, and LDU require one-byte operation codes, whereas LDY and LDS require two-byte codes. So, **when assigning address-length registers in your program, try to maximize the number of single-byte operation codes that are executed.** You can use TFR or EXG to move data from one register to another.
20. **Use the indexed addressing modes to perform address-length additions during an instruction cycle.** This approach is preferable to using LEA when you do not need the result later.
21. **Try to replace sequences of branch instructions with single branches.** You may be able to eliminate sequences by rearranging computations or by using the conditional branches that depend on combinations of flags. Examine the precise effects of branches like BGE, BGT, BHI, BLE, BLS, and BLT; they may be useful in situations that differ greatly from those suggested by their mnemonics.
22. **Use instructions such as BIT, CMP, and TST that affect the flags without changing any registers or memory locations.** You may be able to retain data for later use.

SAVING EXECUTION TIME

Although some of the methods that reduce memory usage also save time, you can generally save an appreciable amount of time only by concentrating on frequently executed loops. Even completely eliminating an instruction that is executed only once can save at most a few microseconds. But a savings in a loop that is executed frequently will be multiplied many times over.

So, if you must reduce execution time, proceed as follows:

1. **Determine how frequently each program loop is executed.** You can do this by hand or by using the software simulator or other testing methods.
2. **Examine the loops in the order determined by their frequency of execution,** starting with the most frequent. Continue through the list until you achieve the required reduction.
3. **First, see if there are any operations that can be moved outside the loop,** such as repetitive calculations, data that can be stored in a register or in the stack, data or addresses that can be stored on the direct page, special cases or errors that can be handled elsewhere, etc. Note that this may require extra initialization and memory but will save time.
4. **Try to eliminate Jump statements.** These are very time-consuming. Sometimes changing the initial conditions helps, particularly if the changes allow you to perform tests at the end of a loop rather than at the beginning.
5. **Replace subroutines with in-line code.** This will save at least a CALL and a RETURN instruction.
6. **Use the stack for temporary data storage** if you can take advantage of the automatic ordering it provides.
7. **Use any of the hints mentioned in saving memory that also decrease execution time.** These include the use of 8-bit addresses, SWI, RTI, special short forms of instructions, etc.
8. **Do not even look at instructions that are executed only once.** Any changes that you make in such instructions only invite errors for no appreciable gain.
9. **Avoid indexed and indirect addressing whenever possible** because they take extra time.
10. **Use tables rather than algorithms;** make the tables handle as much of the tasks as possible even if many entries must be repeated.

MAJOR REORGANIZATION

If you need more than a 25% increase in speed or decrease in memory usage do not try reorganizing the code. Your chances of getting that much of an improvement are small unless you call in an outside expert. You are generally better off making a major change.

BETTER ALGORITHMS

The most obvious change is a better algorithm. Particularly if you are doing sorts, searches, or mathematical calculations, you may be able to find a faster or shorter method in the literature. Libraries of algorithms are available in some journals and from professional groups. See the references at the end of this chapter for some important sources.

OTHER MAJOR CHANGES

Hardware can replace software. Counters, shift registers, arithmetic units, hardware multipliers, and other fast add-ons can save both time and memory. Calculators, UARTs, keyboards, encoders, and other slower add-ons may save memory even though they operate slowly. Compatible parallel and serial interfaces, and other devices specially designed for use with the 6809 or 6502 may save time by taking some of the burden off the CPU.

Other changes may help as well:

1. **A CPU with a longer word will be faster** if the data is long enough. Such a CPU will use less total memory. 16-bit processors, for example, use memory more efficiently than 8-bit processors, since more of their instructions are one word long.
2. **Versions of the CPU may exist that operate at higher clock rates.** But remember that you will need faster memory and I/O ports, and you will have to adjust any delay loops.
3. **Two CPUs may be able to do the job in parallel** or separately if you can divide the job and solve the communications problem.
4. **A specially microprogrammed processor may be able to execute the same program much faster.** The cost, however, will be much higher even if you use an off-the-shelf emulation.
5. **You can make tradeoffs between time and memory.** Lookup tables and function ROMs will be faster than algorithms, but will occupy more memory.

Deciding on a Major Change

This kind of problem, in which a large improvement is necessary, usually results from lack of adequate planning in the definition and design stages. In the problem definition stage you should determine which processor and methods will handle the problem. If you misjudge, the cost later will be high. A cheap solution may result in an unwarranted expenditure of expensive development time. Do not try to just get by; the best solution is usually to do the proper design and chalk a failure up to experience. **If you have followed such methods as flowcharting, modular programming, structured programming, top-down design, and proper documentation, you can salvage a lot of your effort even if you have to make a major change.**

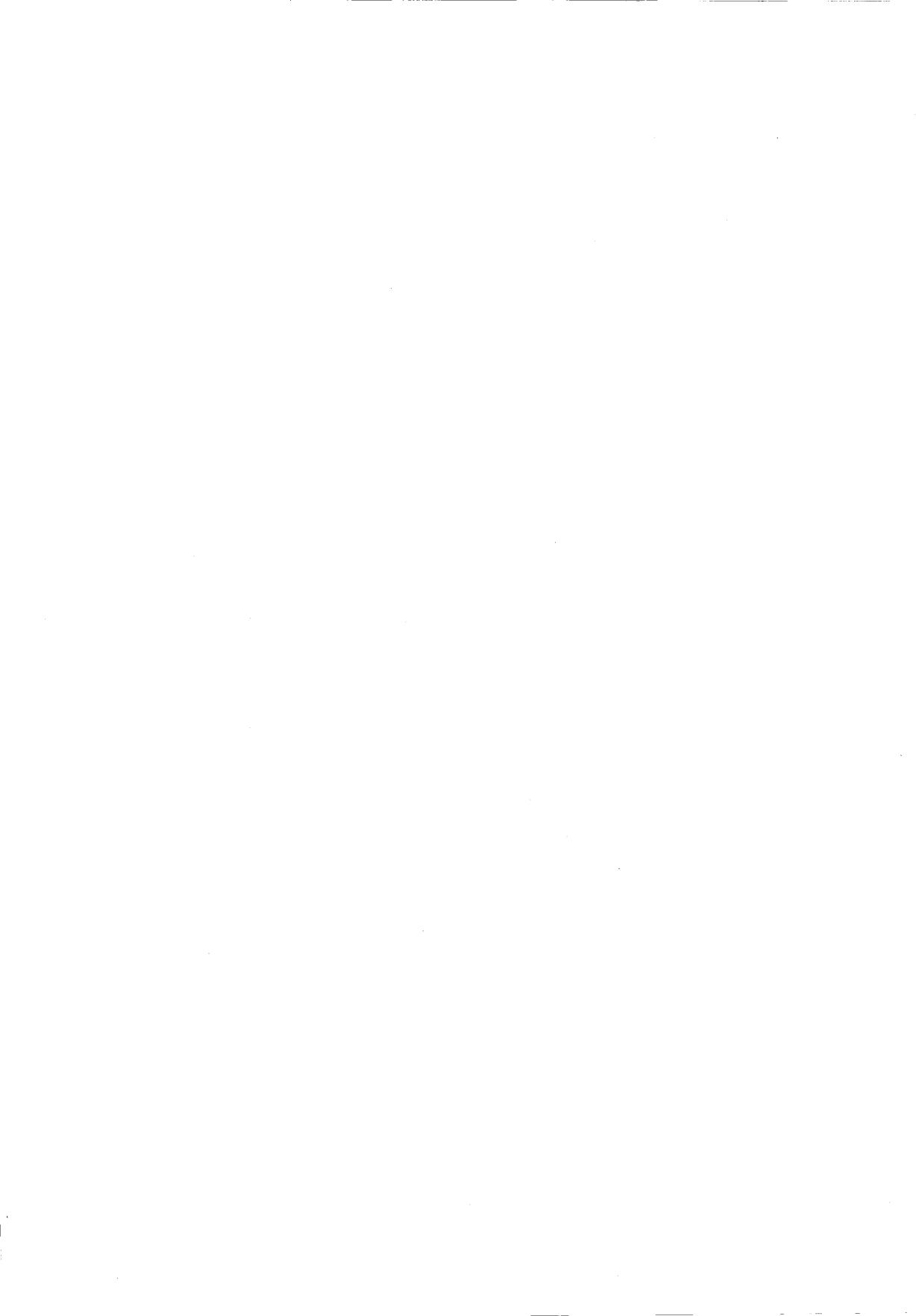
REFERENCES

- Carnahan, B., et al. *Applied Numerical Methods*, Wiley, New York, 1969.
- Chen, T. C. "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots," *IBM Journal of Research and Development*, Volume 18, pp. 380-388, July 1972.
- Collected Algorithms from ACM, ACM Inc., P.O. Box 12105, Church Street Station, New York, 10249.
- Despain, A. M. "Fourier Transform Computers Using CORDIC Iterations," *IEEE Transactions on Computers*, October 1974, pp. 993-1001.
- Edgar, A. D. and S. C. Lee. "FOCUS Microcomputer Number System," *Communications of the ACM*, March 1979, pp. 166-177.
- Hwang, K. *Computer Arithmetic*, Wiley, New York, 1978.
- Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms; The Art of Computer Programming, Volume 2: Seminumerical Algorithms; The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. 1967-1969.
- Luke, Y. L. *Algorithms for the Computation of Mathematical Functions*, Academic Press, New York, 1977.
- Schmid, H. *Decimal Computation*, Wiley-Interscience, New York, 1974.
- New methods for performing arithmetic operations on computers are often discussed in the triennial Symposium on Computer Arithmetic. The Proceedings (starting with 1969) are available from the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, Calif. 90720.

V

6809 Instruction Set

Chapter 22 and the appendices that follow it comprise a total reference for the 6809 instruction set. Chapter 22 describes each instruction in some detail; the appendices summarize that information and also provide material on indexed and indirect addressing modes.



22

Descriptions of Individual 6809 Instructions

In this chapter we present instructions in alphabetical order and describe them in great detail. The information contained here is summarized in Appendices A and C. **We have included several instruction mnemonics which 6809 assemblers may accept to maintain compatibility with 6800 source code.** These may be 6800 instructions which the 6809 does not have or 6800 mnemonics that are not part of the standard 6809 set. Table 3-10 shows the 6800 mnemonics and the equivalent 6809 instructions. In some cases, the instruction is a 6800-like mnemonic extended to the 6809's additional facilities. **The assembler turns each of these mnemonics into a 6809 instruction or sequence whose execution has results equivalent to those of the 6800 instruction.** **These instructions are predefined macro calls and may not be available on all 6809 assemblers.**

A description generally includes a diagram of the execution of the instruction. Since the 6809 microprocessor has so many addressing modes, **we have not attempted to describe all the modes for each instruction.**

ABA — Add Accumulator B to Accumulator A

The 6809 assembler translates this instruction into

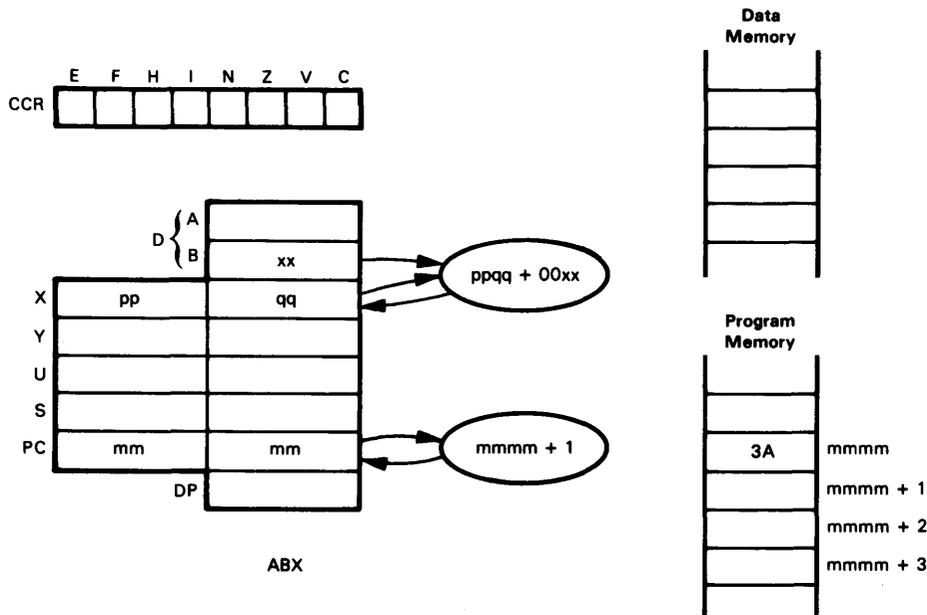
```
PSHS B  
ADDA ,S+
```

This instruction adds the two accumulators and stores the result in Accumulator A. It is included in the assembler to allow source compatibility between the 6800 and the 6809 microprocessors.

ABX — Add Accumulator B to Index Register X Unsigned

	Object Code	No. of Cycles	No. of Bytes
ABX	3A	3	1

Add the contents of Accumulator B to those of Index Register X. Store the result in Index Register X. ABX treats the contents of B as an unsigned number.



Suppose $xx = 84_{16}$ and $ppqq = 1097_{16}$. After the processor executes the ABX instruction, Index Register X will contain $111B_{16}$:

$$\begin{array}{r} 1097 = 0001\ 0000\ 1001\ 0111 \\ 0084 = 0000\ 0000\ 1000\ 0100 \\ \hline 0001\ 0001\ 0001\ 1011 \end{array}$$

This instruction calculates an indexed address and stores it in Register X for later use. For example, Accumulator B could contain a calculated selection code and Index Register X the base address of the table from which the selection would be made. ABX would then store the address of the selected item in Register X. Subsequent instructions could use that address without repeating the indexing process: for example, LDA ,X. The selected item could itself be the address of a set of parameters; these parameters could then be accessed via indirect addressing, as in LDA [,X++].

ABX performs almost the same address calculation as LEAX B,X. Whereas ABX has no effect on the flags, LEAX does affect the zero flag. Be careful of another difference between the two instructions; LEAX treats the contents of Accumulator B as a two's complement number, while ABX interprets the value in B as an unsigned number. For example, if B holds FF_{16} and X contains $27E1_{16}$, execution of ABX will leave $28E0_{16}$ in X, but LEAX B,X will place $27E0_{16}$ in X. LEAX B,X should be used in

most situations; however, when program space or execution time is at a premium, the shorter, faster ABX can replace it. Remember, though, that when you replace LEAX B,X with ABX, you may have to relocate the memory area being addressed by the index register.

ABX affects no flags; it is meant to manipulate addresses rather than data. This is one of the few 6809 instructions that lack generality; it applies to only two specific registers — B and X — and cannot be extended to any others. The instruction is included in the 6809 instruction set for compatibility with the 6801 processor.

ADC — Add Memory Plus Carry to Accumulator A or B

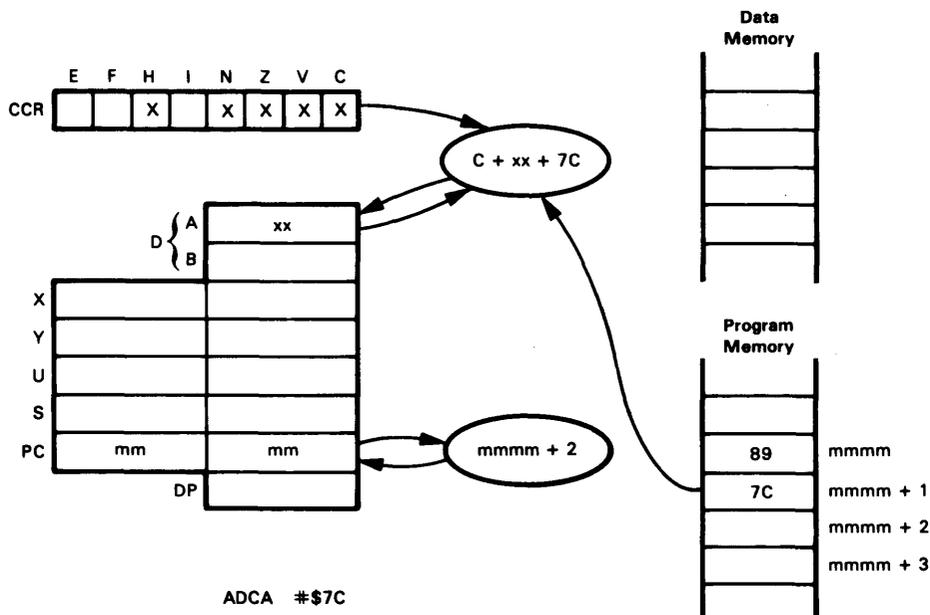
ADCA

ADCB

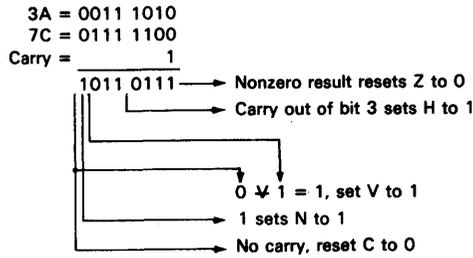
	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ADCA	89	2	2	99	4	2	B9	5	3	A9	4+	2+
ADCB	C9	2	2	D9	4	2	F9	5	3	E9	4+	2+

This instruction adds the contents of a memory location and the contents of the Carry flag to the contents of Accumulator A or B. The result is stored in the specified accumulator.

Consider performing an addition with Carry using immediate data and Accumulator A.



Suppose that $xx = 3A_{16}$ and $C = 1$. After the processor executes the instruction ADCA #\$7C, Accumulator A will contain $B7_{16}$.



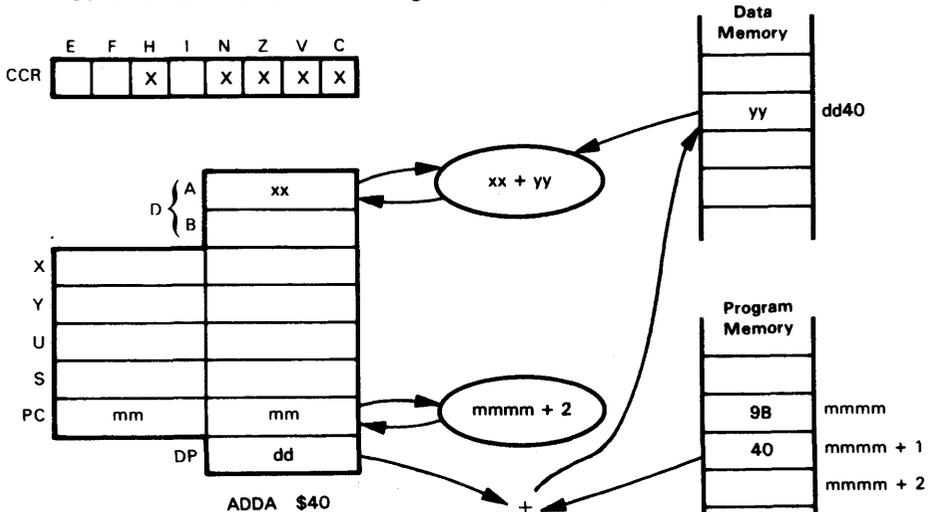
The ADC instruction is most frequently used in multibyte additions, to include the carry in the addition of the second and subsequent bytes. Note that for double byte addition, the ADDD instruction (described next) will perform the 16-bit addition in one instruction, and the ADC instruction for the high-order byte is not necessary.

ADD — Add Memory to Accumulator
ADDA
ADDB
ADD

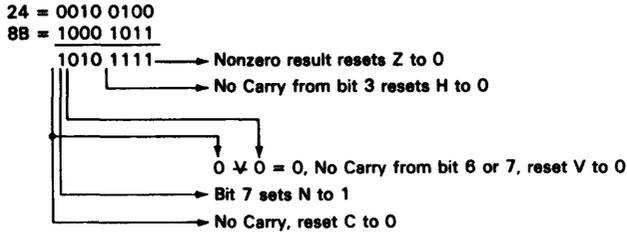
	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ADDA	8B	2	2	9B	4	2	BB	5	3	AB	4+	2+
ADDB	CB	2	2	DB	4	2	FB	5	3	EB	4+	2+
ADD	C3	4	3	D3	6	2	F3	7	3	E3	6+	2+

ADDA and ADDB add the contents of a memory location to the value in Accumulator A or Accumulator B, placing the sum in the designated accumulator. ADDD adds the contents of a memory word (two contiguous bytes) to the value in the double accumulator, placing the result in the double accumulator.

Consider the 8-bit addition using direct addressing and Accumulator A.



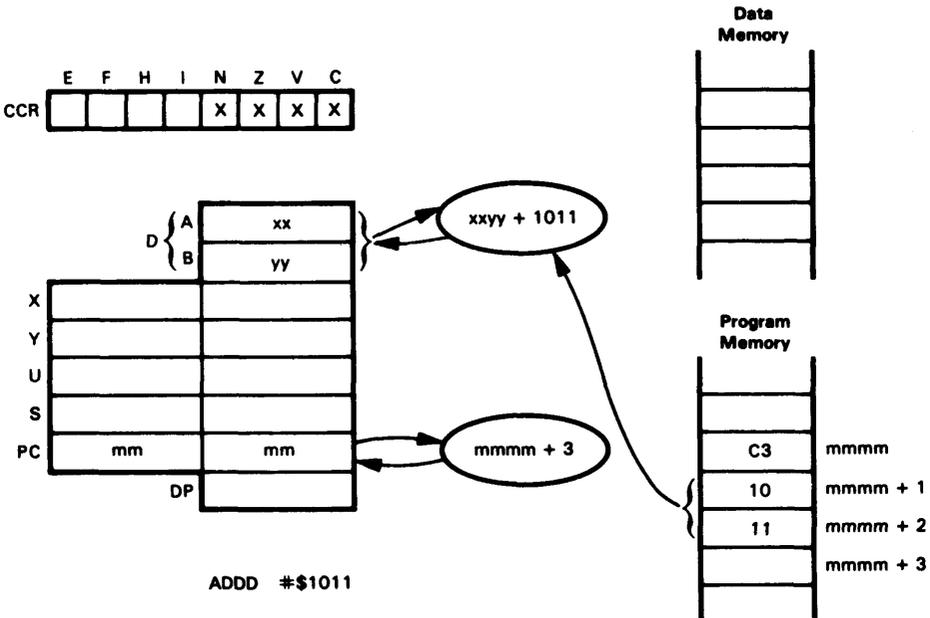
Suppose $xx = 24_{16}$, $dd = 00_{16}$, and the contents of memory byte 0040— (yy) are $8B_{16}$. After execution of the ADDA \$40 instruction, Accumulator A will contain AF_{16} :



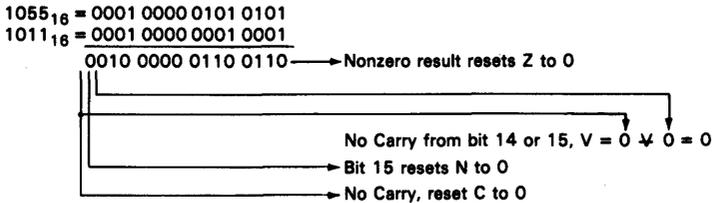
ADDA and ADDB are the usual single-byte addition instructions; they are also used to add the least significant bytes of multibyte addends greater than 16 bits. **ADDD**, which we will describe next, is available for 16-bit addition.

Now consider the **ADDD** instruction. This instruction adds the contents of two memory locations to the Double Accumulator D. The double accumulator's high-order byte is Accumulator A; its low-order byte is Accumulator B. The number to be added has its high-order byte in the first memory address and its low-order byte in the subsequent memory address.

We will illustrate the **ADDD** instruction using immediate addressing.



If $xx = 10_{16}$ and $yy = 55_{16}$, the instruction `ADDD # $\$1011$` yields 2066_{16} in Accumulator D: that is, 20_{16} in Accumulator A and 66_{16} in Accumulator B.



Note that `ADDD` does not affect the H flag.

The `ADDD` instruction can be used to perform 16-bit addition in preference to `ADDB`, `ADCA`, which together take longer and require more memory. However, it cannot readily be extended to handle longer data because of the lack of any way to add in carries; that is, there is no `ADCD` instruction. You must remember the order of the accumulators (A high-order, B low-order) and the fact that two memory locations are used for data: the one addressed and the one following that.

AND — Logical AND Accumulator or Condition Code Register

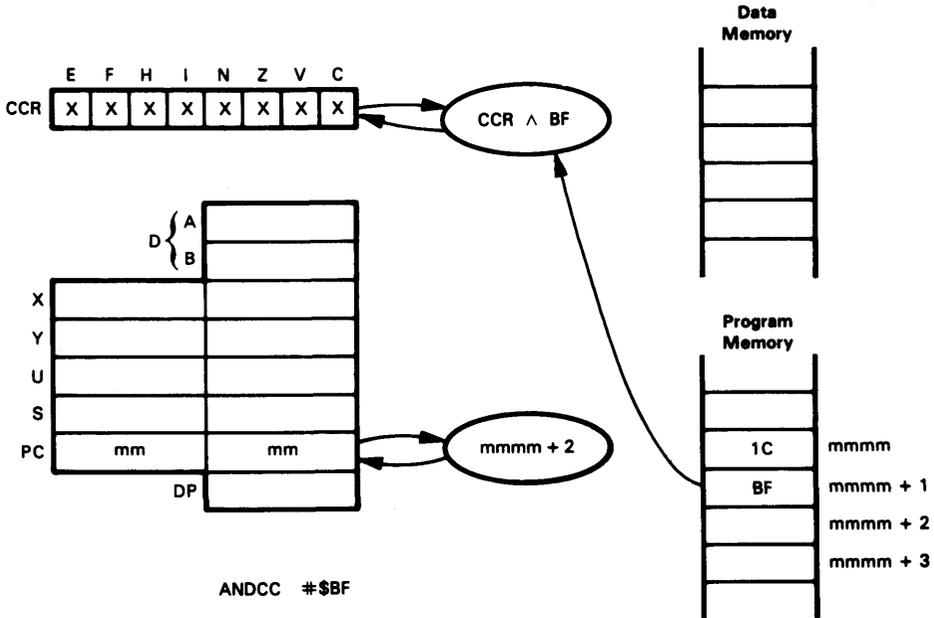
ANDA
ANDB
ANDCC

	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ANDA	84	2	2	94	4	2	B4	5	3	A4	4+	2+
ANDB	C4	2	2	D4	4	2	F4	5	3	E4	4+	2+
ANDCC	1C	3	2									

This instruction logically ANDs the contents of a memory location with Accumulator A or Accumulator B; the `ANDCC` instruction allows only immediate addressing and performs a logical AND of the condition code register with the immediate byte. The result of the AND operation is stored in the designated register.

Consider the `AND` instruction using Accumulator B and indexed addressing with zero offset.

Now consider the ANDCC instruction.



Let the CCR = $D4_{16}$. After the instruction ANDCC #\$BF, the CCR will contain 94_{16} .

$$\begin{array}{r} D4_{16} = 1101\ 0100 \\ BF_{16} = 1011\ 1111 \\ \hline 94_{16} = 1001\ 0100 \end{array}$$

All flags may be affected by the ANDCC operation. It clears all the flags that are logically ANDed with '0's, while leaving the other flags unchanged. The following masks can be used to clear individual flags:

Flag	Required Mask	
	Binary	Hexadecimal
E	0111 1111	7F
F	1011 1111	BF
H	1101 1111	DF
I	1110 1111	EF
N	1111 0111	F7
Z	1111 1011	FB
V	1111 1101	FD
C	1111 1110	FE

Of course, '0's in more than one bit position will clear more than one flag at a time. However, only a few possibilities are really useful. In particular, we should note:

ANDCC	#1011 1111	ENABLE FAST INTERRUPTS
ANDCC	#1110 1111	ENABLE REGULAR INTERRUPTS
ANDCC	#1111 1101	CLEAR OVERFLOW
ANDCC	#1111 1110	CLEAR CARRY

Remember that clearing an interrupt mask enables the interrupt from that source. This instruction is used to enable the regular or fast interrupt, to clear the Overflow flag for later use, and to clear the Carry flag for use as an indicator or to start multi-

ple-precision addition — there is, of course, no carry into the least significant bytes, so the Carry must be cleared originally. The Carry must also be cleared initially for multiple-precision binary subtraction, to signify that there is no borrow required from the least significant bytes. The CLR instruction also clears the Carry flag.

ASL — Shift Accumulator or Memory Byte Left

ASLA

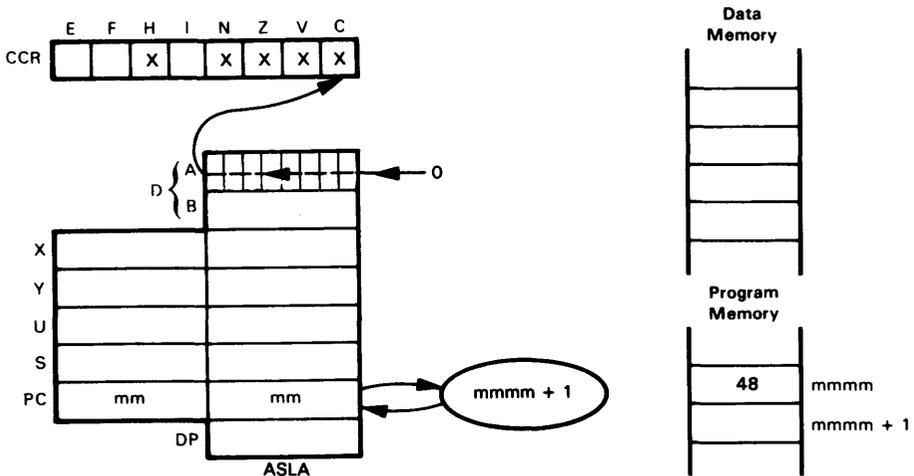
ASLB

ASL

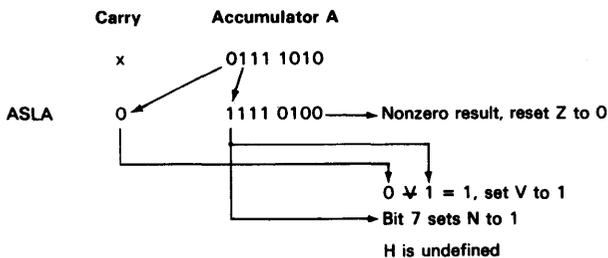
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ASL	48	2	1	08	8	2	78	7	3	68	6+	2+
ASLA	58	2	1									
ASLB												

Shift the contents of Accumulator A or B or the contents of the selected byte of memory left one bit arithmetically, clearing the least significant bit.

Consider shifting an accumulator (A):



Suppose that Accumulator A contains $7A_{16}$. Executing an ASLA instruction changes the contents of Accumulator A to $F4_{16}$.



The Overflow flag (V) is loaded with the Exclusive-OR of bits 7 and 6 of the original operand; these bit values are the same as those of the resulting Carry (C) and Sign (N) flags. **Common uses of ASL include simple multiplication (by small integers such as 2 or 4), serial-to-parallel conversion, and scaling.** Note that a single ASL instruction multiplies its operand by 2. This instruction is the same as Logical Shift Left (LSL).

An ASL operation on a memory location is exactly like the accumulator operation. There is, of course, some difference in object code size and execution time, depending on the addressing mode.

ASR — Shift Accumulator or Memory Byte Right

ASRA

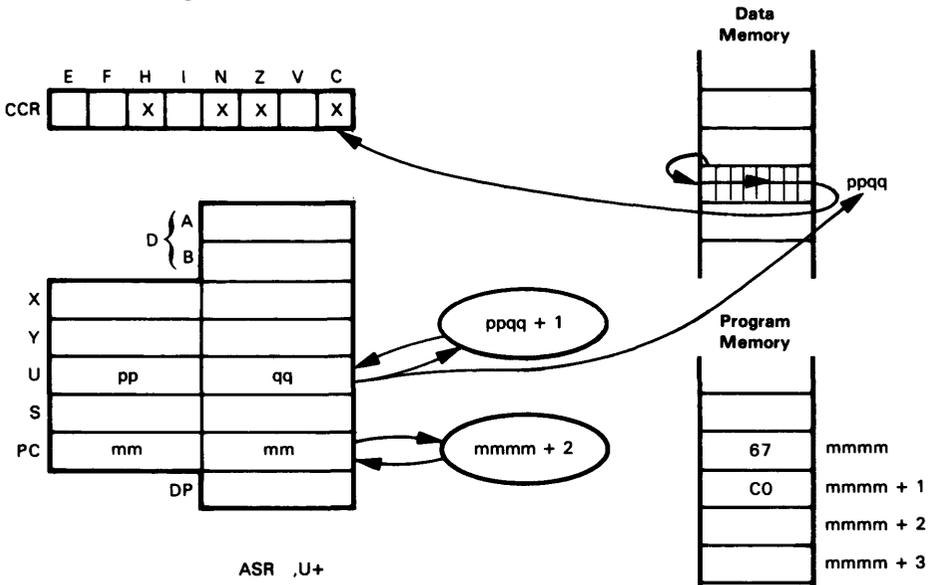
ASRB

ASR

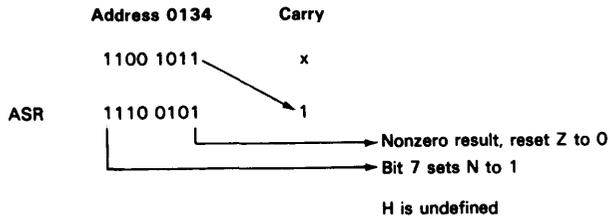
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ASR	47	2	1	07	6	2	77	7	3	67	6+	2+
ASRA	57	2	1									
ASRB												

Perform a one-bit arithmetic right shift of the contents of Accumulator A or B or the contents of a selected byte of memory.

Consider shifting a memory location right. The addressing mode is indexed, autoincrementing User Stack Pointer U.



Suppose $ppqq = 0134_{16}$ and the contents of location 0134_{16} are CB_{16} . Executing an ASR ,U+ instruction will change the contents of memory location 0134_{16} to $E5_{16}$. The final contents of the User Stack Pointer will be 0135_{16} .



While the 6809's ASR instruction does not affect the Overflow flag (V), those of the 6800/01/02/03/08 processors do.

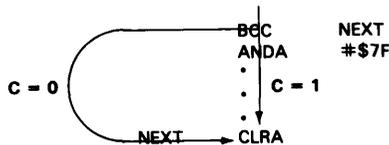
An arithmetic right shift preserves the value of the most significant bit (or sign bit); it can thus be used for scaling twos complement numbers, since it retains their signs. The ASR instruction is frequently used in division routines.

BCC — Branch if Carry Clear (C = 0)

	Object Code	No. of Cycles	No. of Bytes
BCC	24	3	2

This instruction is the same as BRA except that it causes a branch only if the Carry flag is 0. If the Carry flag is 1, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BCC, the processor next executes:

1. CLRA if the Carry flag is 0.
2. ANDA if the Carry flag is 1.

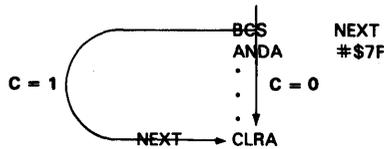
When used after a subtract or compare on unsigned binary values, this instruction could be called "branch if the register was higher or the same as the memory operand;" in fact, the 6809 assembler will accept the mnemonic BHS for this instruction.

BCS — Branch if Carry Set (C = 1)

	Object Code	No. of Cycles	No. of Bytes
BCS	25	3	2

This instruction is the same as BRA except that it causes a branch only if the Carry flag is 1. If the Carry flag is 0, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BCS, the processor next executes:

1. CLRA if the Carry flag is 1.
2. ANDA if the Carry flag is 0.

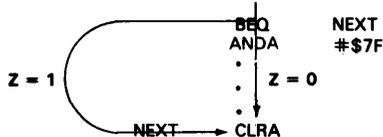
When used after a subtract or compare on unsigned binary values, this instruction could be called “branch if the register was higher or the same as the memory operand;” in fact, the 6809 assembler will accept the mnemonic BHS for this instruction.

BEQ — Branch if Equal to Zero (Z = 1)

	Object Code	No. of Cycles	No. of Bytes
BEQ	27	3	2

This instruction is the same as BRA except that it causes a branch only if the Zero flag is 1. If the Zero flag is 0, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BEQ, the processor next executes:

1. CLRA if the Zero flag is 1.
2. ANDA if the Zero flag is 0.

Remember that the Zero flag is set to 1 if the most recent result was zero. When BEQ is used after a subtract or compare, branching will occur only if the values compared were exactly the same.

BGE — Branch if Greater Than or Equal to Zero ($N \oplus V = 0$)

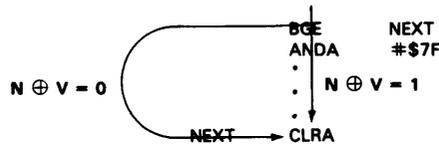
	Object Code	No. of Cycles	No. of Bytes
BGE	2C	3	2

This instruction is the same as BRA except that it causes a branch only if:

1. The Sign flag is 1 and the Overflow flag is 1, or
2. The Sign flag is 0 and the Overflow flag is 0.

If neither of these conditions is true, the processor continues to the next instruction in the normal sequence. The branch conditions can be simplified logically to the form $N \oplus V = 0$.

Consider the following section of a program:



After executing BGE, the processor next executes:

1. CLRA if $N \oplus V = 0$.
2. ANDA if $N \oplus V = 1$.

The conditions have the following significance if a CMP (compare) instruction immediately precedes the branch:

1. $N = 0$ and $V = 0$ if the result of CMP is positive ($N = 0$), and twos complement overflow did not occur ($V = 0$).
2. $N = 1$ and $V = 1$ if the result appears to be negative ($N = 1$), but the sign was changed by twos complement overflow ($V = 1$).

Thus the branch occurs if the result is a true positive (unaffected by overflow) or a false negative (affected by overflow). This analysis assumes that the numbers are all in twos complement form. BGE thus provides a twos complement Greater Than or Equal To branch; alternatives are:

1. BGT, a twos complement Greater Than branch.
2. BHS (BCC), an unsigned Greater Than or Equal To branch.

BGT — Branch if Greater Than Zero ($Z + (N \oplus V) = 0$)

	Object Code	No. of Cycles	No. of Bytes
BGT	2E	3	2

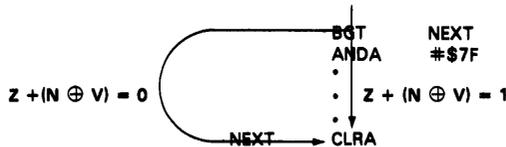
This instruction is the same as BRA except that it causes a branch only if the Zero

flag is 0 and:

1. The Sign flag is 1 and the Overflow flag is 1, or
2. The Sign flag is 0 and the Overflow flag is 0.

If this condition is not true, the processor continues to the next instruction in the normal sequence. The branch condition can be simplified logically to the form $Z + (N \oplus V) = 0$.

Consider the following section of a program:



After executing BGT, the processor next executes:

1. CLRA if $Z + (N \oplus V) = 0$.
2. ANDA if $Z + (N \oplus V) = 1$.

The condition has the following significance if a CMP (Compare) instruction immediately precedes the branch:

1. $Z = 0$, $N = 0$, and $V = 0$ if the result of CMP is positive but not zero ($Z = 0$ and $N = 0$) and twos complement overflow did not occur ($V = 0$).
2. $Z = 0$, $N = 1$, and $V = 1$ if the result of CMP is not zero and appears to be negative ($N = 1$), but its sign was changed by twos complement overflow ($V = 1$).

So the branch occurs if the result is not zero and is either a true positive number (unaffected by overflow) or a false negative number (affected by overflow). This analysis assumes that all numbers are in twos complement form. BGT thus provides a twos complement Greater Than branch; alternatives are

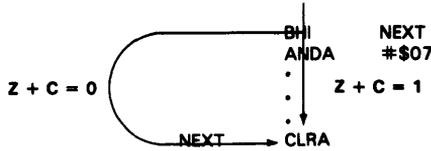
1. BGE, a twos complement Greater Than or Equal To branch.
2. BHI, an unsigned Greater Than branch.

BHI — Branch if Higher ($Z + C = 0$)

	Object Code	No. of Cycles	No. of Bytes
BHI	22	3	2

This instruction is the same as BRA except that it causes a branch only if the Zero flag and the Carry flag are both 0. If either flag is not zero, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BHI, the processor next executes:

1. CLRA if the Carry flag and the Zero flag are both 0.
2. ANDA if the Carry flag and/or the Zero flag is 1.

The condition has the following significance if a CMP (compare) instruction immediately precedes the branch:

1. $C = 0$ and $Z = 0$ if the result of CMP is not zero and the operation did not produce a borrow. Remember that the operation sets C to 1 if it requires a borrow (that is, if the contents of the register were smaller in the unsigned sense than the number to which they were compared).
2. $C = 1$ and/or $Z = 1$ if either the result of CMP is zero or the operation produced a borrow.

BHI differs from BHS (BCC) after a comparison only if the result is zero; BHS causes a branch in that case, while BHI does not. BHI thus provides an unsigned Greater Than branch; alternatives are:

1. BHS (BCC), an unsigned Greater Than or Equal To branch.
2. BGT, a twos complement Greater Than branch.

The instruction BHI is generally not useful after INC/DEC, LD/ST, or TST/CLR/COM: CLR always resets the Carry flag to 0; COM always sets the Carry flag to 1, and the other instructions listed do not affect the Carry flag.

BHS — Branch If Higher or Same (C = 0)

	Object Code	No. of Cycles	No. of Bytes
BHS	24	3	2

This instruction is exactly the same as BCC. The alternative mnemonic reflects the fact that the condition has the following significance if a CMP (compare) instruction immediately precedes the branch:

1. $C = 0$ if the operation did not require a borrow. That is, the unsigned number in the register was greater than or equal to the unsigned number to which it was compared.
2. $C = 1$ if the operation required (produced) a borrow. That is, the unsigned number in the register was less than the unsigned number to which it was compared.

BHS (BCC) causes a branch if the operation did not require a borrow. BHS (BCC) thus provides an unsigned Greater Than or Equal To branch; alternatives are:

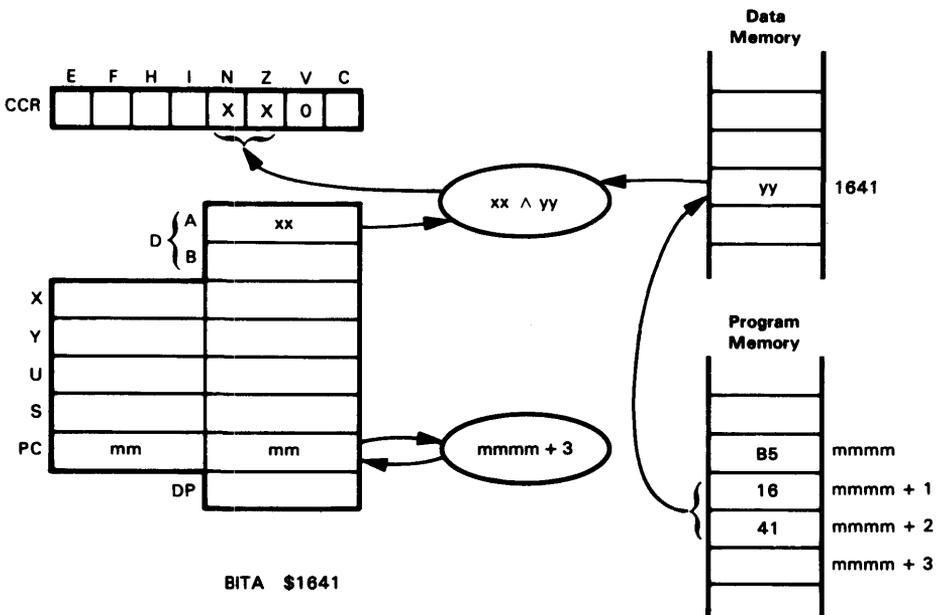
1. BGE, a twos complement Greater Than or Equal To branch.
2. BHI, an unsigned Greater Than branch.

This instruction is generally not useful after INC/DEC, LD/ST, or TST/CLR/COM: CLR always resets the Carry flag to 0, COM always sets the Carry flag to 1, and the other instructions listed do not affect the Carry flag.

BIT — Bit Test
BITA
BITB

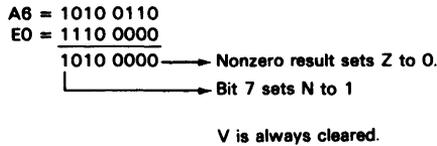
	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
BITA	85	2	2	95	4	2	B5	5	3	A5	4+	2+
BITB	C5	2	2	D5	4	2	F5	5	3	E5	4+	2+

This instruction ANDs the contents of accumulator A or B with the contents of a selected memory location and sets the flags accordingly, but does not alter the contents of the accumulator or memory byte. We illustrate this instruction with extended addressing and Accumulator A.



Suppose $xx = A6_{16}$ and $yy = E0_{16}$. After the processor executes BITA \$1641, Accumulator A will still contain $A6_{16}$, and memory location 1641_{16} will still contain $E0_{16}$

but the flags will be modified as follows:



BIT instructions frequently precede conditional branch instructions. BIT instructions are also used to perform masking functions on data. Note that BIT instructions differ from AND instructions only in that BIT instructions do not change the contents of the selected accumulator, thus allowing further tests or other operations without reloading.

BLE — Branch If Less Than or Equal to Zero ($Z + (N \oplus V) = 1$)

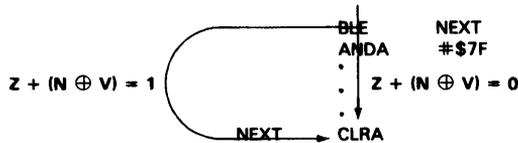
	Object Code	No. of Cycles	No. of Bytes
BLE	2F	3	2

This instruction is the same as BRA except that it causes a branch only if:

1. The Zero flag is 1 or
2. The Sign flag is 1 and the Overflow flag is 0 or
3. The Sign flag is 0 and the Overflow flag is 1.

If none of these conditions is true, the processor continues to the next instruction in the normal sequence. The branch conditions can be simplified logically to the form $Z + (N \oplus V) = 1$.

Consider the following section of a program:



After executing BLE, the processor next executes:

1. CLRA if $Z + (N \oplus V) = 1$.
2. ANDA if $Z + (N \oplus V) = 0$.

The condition has the following significance if a CMP (compare) instruction immediately precedes the branch:

1. $Z = 1$ if the result of CMP is zero.
2. $N = 1$ and $V = 0$ if the result of CMP is negative ($N = 1$), and twos complement overflow did not occur ($V = 0$).
3. $N = 0$ and $V = 1$ if the result appears to be positive ($N = 1$), but the sign was changed by twos complement overflow ($V = 1$).

So the branch occurs if the result is zero, a true negative (unaffected by overflow), or a false positive (affected by overflow). This analysis assumes that all numbers are in the twos complement form. BLE thus provides a twos complement Less Than or Equal To branch; alternatives are:

1. BLT, a twos complement Less Than branch.
2. BLS, an unsigned Less Than or Equal To branch.

BLO – Branch If Lower (C = 1)

	Object Code	No. of Cycles	No. of Bytes
BLO	25	3	2

This instruction is exactly the same as BCS. The alternative mnemonic reflects the fact that the condition has the following significance if a CMP (compare) instruction immediately precedes the branch:

1. C = 1 if the operation required (produced) a borrow. That is, the unsigned number in the register was less than the unsigned number to which it was compared.
2. C = 0 if the operation did not require a borrow. That is, the unsigned number in the register was greater than or equal to the unsigned number to which it was compared.

So BLO (BCS) causes a branch if the operation required a borrow. BLO (BCS) thus provides an unsigned Less Than branch; alternatives are:

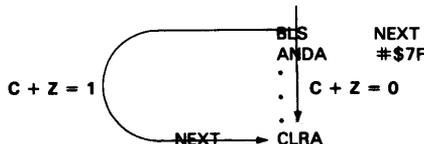
1. BLS, an unsigned Less Than or Equal To branch.
2. BLT, a twos complement Less Than branch.

BLS – Branch If Lower or Same (C + Z = 1)

	Object Code	No. of Cycles	No. of Bytes
BLS	23	3	2

This instruction is the same as BRA except that it causes a branch if either the Carry flag is 1 or the Zero flag is 1. If neither flag is 1, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BLS, the processor next executes:

1. CLRA if either the Carry flag or the Zero flag is 1.
2. ANDA if the Carry flag and the Zero flag are both 0.

The condition has the following significance if a CMP (compare) instruction immediately precedes the result:

1. C = 1 if the operation produced a borrow. Remember that the operation sets C to 1 if the contents of the register were smaller in the unsigned sense than the number to which they were compared.
2. Z = 1 if the result of CMP is zero.

BLS differs from BLO (BCS) after a comparison only if the result is zero; BLS causes a branch in that case, while BLO (BCS) does not, since no borrow is required if the result is zero. BLS thus provides an unsigned Less Than or Equal To branch; alternatives are:

1. BLO (BCS), an unsigned Less Than branch.
2. BLE, a twos complement Less Than or Equal To branch.

The BLS instruction is generally not useful after INC/DEC, LD/ST, TST/CLR/COM: CLR always resets the Carry flag to 0, COM always sets the Carry flag to 1, and the other instructions listed do not affect the carry flag.

BLT — Branch If Less Than Zero ($N \oplus V = 1$)

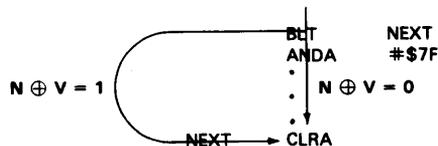
	Object Code	No. of Cycles	No. of Bytes
BLT	2D	3	2

This instruction is the same as BRA except that it causes a branch only if:

1. The Sign flag is 1 and the Overflow flag is 0 or
2. The Sign flag is 0 and the Overflow flag is 1.

If neither of these conditions is true, the processor continues to the next instruction in the normal sequence. The branch conditions can be simplified logically to the form $N \oplus V = 1$.

Consider the following section of a program:



After executing BLT, the processor next executes:

1. CLRA if $N \oplus V = 1$.
2. ANDA if $N \oplus V = 0$.

The conditions have the following significance if a CMP (compare) instruction immediately precedes the branch:

1. $N = 1$ and $V = 0$ if the result of CMP is negative ($N = 1$), and twos complement overflow did not occur ($V = 0$).
2. $N = 0$ and $V = 1$ if the result appears to be positive ($N = 0$), but the sign was changed by a twos complement overflow ($V = 1$).

So a branch occurs if the result is a true negative (unaffected by overflow) or a false positive (affected by overflow). This analysis assumes that the numbers are all in the twos complement form. BLT thus provides a twos complement Less Than branch; alternatives are:

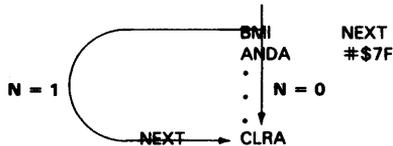
1. BLE, a twos complement Less Than or Equal To branch.
2. BLO (BCS), an unsigned Less Than branch.

BMI – Branch If Minus ($N = 1$)

	Object Code	No. of Cycles	No. of Bytes
BMI	2B	3	2

This instruction is the same as BRA except that it causes a branch only if the Sign flag is 1. If the Sign flag is 0, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BMI, the processor next executes:

1. CLRA if the Sign flag is 1.
2. ANDA if the Sign flag is 0.

BMI is used to test the value in bit position 7; that bit position is often used for parity, status indicators, or peripheral status bits. Used after an operation on twos complement binary values, this instruction will “branch if the result is minus,” but the sign may be invalid due to twos complement overflow.

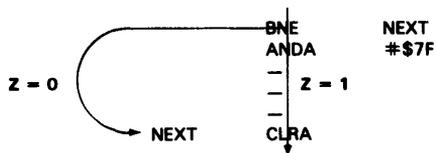
BNE – Branch If Not Equal to Zero ($Z = 0$)

	Object Code	No. of Cycles	No. of Bytes
BNE	26	3	2

This instruction is the same as BRA except that it causes a branch only if the zero

flag is 0. If the Zero flag is 1, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BNE, the processor next executes:

1. CLRA if the Zero flag is 0.
2. ANDA if the Zero flag is 1.

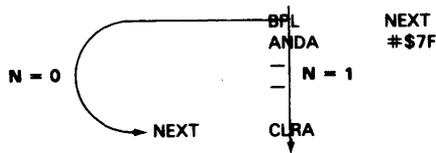
Remember that the Zero flag is set to 0 if the most recent result was not zero. Used after a subtract or compare operation on any binary values, this instruction will “branch if the register is not equal to the memory operand.”

BPL — Branch If Plus (N = 0)

	Object Code	No. of Cycles	No. of Bytes
BPL	2A	3	2

This instruction is the same as BRA except that it causes a branch only if the Sign flag is 0. If the Sign flag is 1, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BPL, the processor next executes:

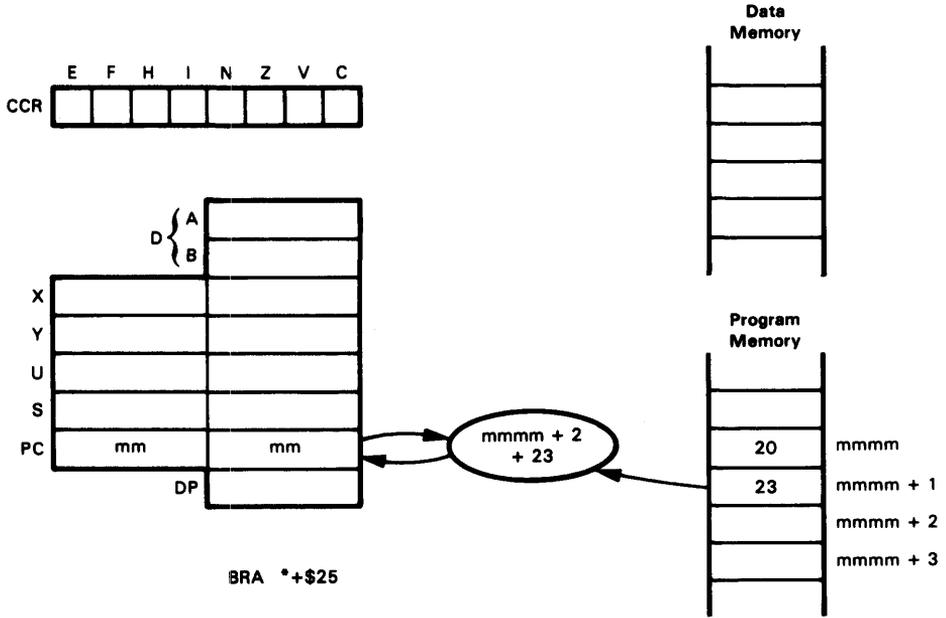
1. CLRA if the Sign flag is 0.
2. ANDA if the Sign flag is 1.

Used after an operation on twos complement binary values, this instruction will “branch if the result is positive,” but the sign may be invalid due to twos complement overflow.

BRA — Branch Always

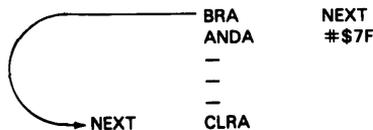
	Object Code	No. of Cycles	No. of Bytes
BRA	20	3	2

BRA always causes a branch to the specified address by placing that address in the Program Counter. The specified address is the sum of the current value of the program counter (after the processor has fetched the BRA instruction from memory) and the displacement. The displacement is an 8-bit twos complement number contained in the second byte of the instruction.



If $mmmm = 2042_{16}$, after $BRA *+$25$ is executed, the program counter will contain 2067_{16} , and the processor continues executing instructions from that point.

Consider the following section of a program:



After executing BRA, the processor always executes CLRA next. It will never execute the ANDA instruction unless a branch or jump instruction elsewhere in the program transfers control to that instruction.

The overall effect of a BRA instruction is:

$$PC = PC + 2 + disp$$

The extra factor of 2 is the result of the 2 bytes occupied by the BRA instruction itself. Since the displacement is an 8-bit twos complement number with the range:

$$-128 (10000000_2) \leq disp \leq +127 (01111111_2).$$

the range of a BRA instruction is:

$$* - 126 \leq destination \leq * + 129$$

where * refers to the value of the Program Counter at the start of the instruction.

BRA does not affect any flags or any registers except the program counter (its previous value is lost). Some typical example displacements are:

1. 05_{16}

The final value of the program counter is its original value plus 7 (5 more than its normal value at the end of a 2-byte instruction).

2. FE_{16}

The final value of the program counter is the same as its original value, since $FE_{16} = -2$ when considered as an 8-bit two's complement number. This displacement results in an endless loop.

3. FA_{16}

The final value of the program counter is its original value minus 4-6 less than its normal value at the end of a 2-byte instruction. $FA_{16} = -6$ when considered as an 8-bit two's complement number.

Note that a displacement of 00 produces a no-operation instruction (the processor continues its normal sequence) while a displacement of FF (or -1) makes no sense since it branches back into the middle of the BRA instruction itself.

BRN — Branch Never

	Object Code	No. of Cycles	No. of Bytes
BRN	21	3	2

BRN is the same as BRA except that it never causes a branch. Thus BRN is really a no-operation; that is, control always passes to the next instruction. Note that BRN is a 2-byte no-operation, since the second byte contains the displacement that will never be used. BRN makes the set of branches logically complete. Typical usage of BRN is as a byte filler, or it may be used to fine-tune delay routines. See the NOP instruction description for a discussion of uses for no-operations.

BSR — Branch to Subroutine

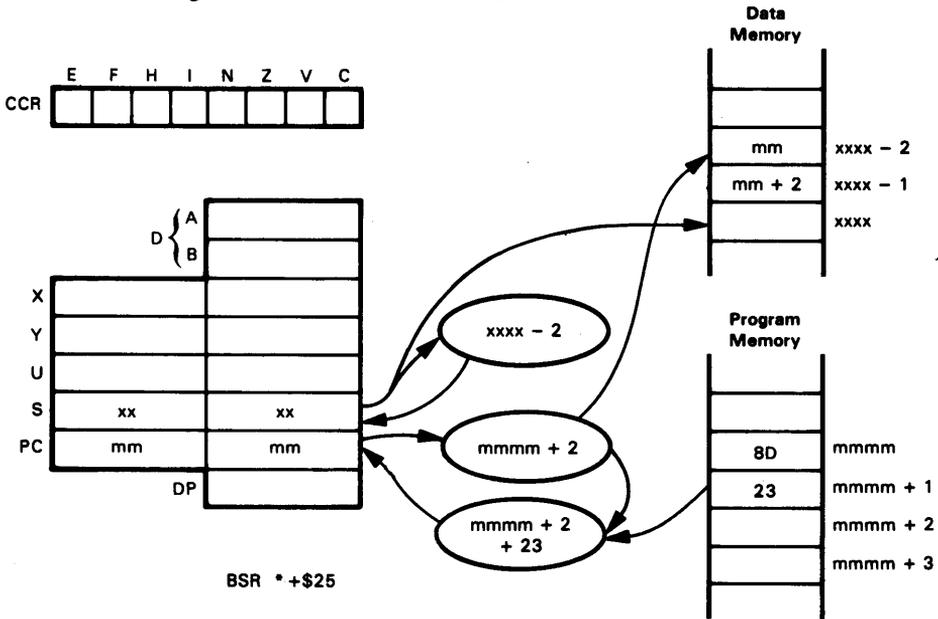
	Object Code	No. of Cycles	No. of Bytes
BSR	8D	7	2

This instruction is the same as BRA except that it saves the contents of the Program Counter (after the 2-byte BSR instruction has been fetched) in the Hardware Stack.

BSR saves the return address in the Hardware Stack as follows:

1. Decrement the Hardware Stack Pointer and store the low-order byte of the program counter at that address.

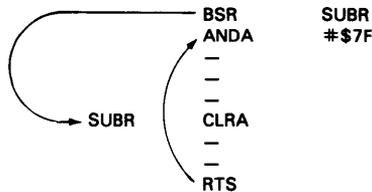
- Decrement the hardware Stack Pointer again and store the high-order byte of the Program Counter at that address.



Suppose $xxxx = DE30_{16}$ and $mmmm = 1024_{16}$. After the execution of $BSR *+$25$ the stack pointer S will contain $DE2E_{16}$, the Program Counter will contain 1049_{16} , location $DE2E_{16}$ will contain 10_{16} , and location $DE2F$ will contain 26_{16} .

A later instruction (such as RTS or $PULS PC$) can then restore that address to the program counter and thus resume execution of the calling program. BSR differs from BRA in that BSR "remembers" where it came from, thus allowing control to pass to a subroutine and back.

Consider the following section of a program:



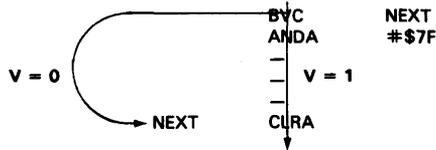
After executing BSR , the processor always executes $CLRA$ next just as after BRA . However, it also saves the address of the $ANDA$ instruction at the top of the hardware stack. Later an RTS instruction can conclude the subroutine and transfer control back to the return address at the top of the hardware Stack. Thus control passes from the BSR instruction to the subroutine and back to the $ANDA$ instruction.

BVC — Branch If Overflow Clear (V = 0)

	Object Code	No. of Cycles	No. of Bytes
BVC	28	3	2

BVC is the same as BRA except that it causes a branch only if the Overflow flag is 0. If the Overflow flag is 1, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BVC, the processor next executes:

1. CLRA if the Overflow flag is 0.
2. ANDA if the Overflow flag is 1.

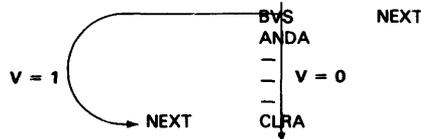
Used after an operation on two's complement binary values, this instruction will "branch if there was no overflow."

BVS — Branch If Overflow Set (V = 1)

	Object Code	No. of Cycles	No. of Bytes
BVS	29	3	2

BVS is the same as BRA except that it causes a branch only if the Overflow flag is 1. If the Overflow flag is 0, the processor continues to the next instruction in the normal sequence.

Consider the following section of a program:



After executing BVS, the processor next executes:

1. CLRA if the Overflow flag is 1.
2. ANDA if the Overflow flag is 0.

Used after an operation on two's complement binary values, this instruction will "branch if there was an overflow." This instruction is also used after ASL or LSL to detect binary floating-point normalization.

CBA — Compare Accumulators

The 6809 assembler translates this 6800 instruction into:

```
PSHS  B  
CMPA  ,S+
```

This instruction subtracts Accumulator B from Accumulator A and sets the flags accordingly. It is handled by the 6809 assembler to allow source compatibility with the 6800 processor. The contents of the accumulators do not change.

CLC — Clear Carry

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction `ANDCC #%11111110` — that is, into an instruction that clears the least significant bit of the Condition Code Register (the Carry flag). CLC does not affect any other flags or registers. Note that the CLR instruction also clears the Carry flag.

CLF — Clear Fast Interrupt Mask

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction `ANDCC #%10111111` — that is, into an instruction that clears bit 6 of the Condition Code Register (the fast interrupt mask). This instruction enables the fast interrupt — that is, the 6809 will respond to the fast interrupt request control line. No other registers or flags are affected. Note that you can also clear the fast interrupt mask as part of the execution of the CWAI instruction.

CLI — Clear Interrupt Mask

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction `ANDCC #%11101111` — that is, into an instruction that clears bit 4 of the Condition Code Register (the regular interrupt mask bit). This instruction enables the 6809's regular interrupt — that is, the 6809 will respond to the interrupt request control line. No other registers or flags are affected. Note that you can also clear the interrupt mask as part of the execution of the CWAI instruction.

CLIF — Clear Regular and Fast Interrupt Masks

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction `ANDCC #%10101111` — that is, into an instruction that clears bits 4 and 6 of the Condition Code Register (the regular and fast interrupt mask bits). This instruction enables both of the 6809's maskable interrupts — that is, the 6809 will respond to either the fast interrupt request control line or to the interrupt request control line. No other registers or flags are affected. Note that you can also clear the interrupt masks as part of the execution of the CWAI instruction.

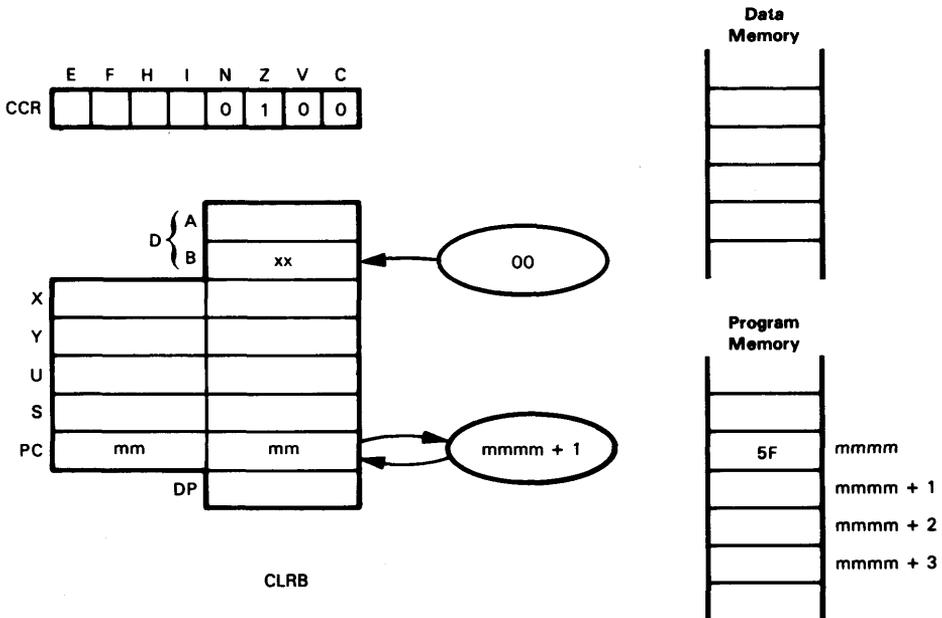
CLR — Clear Accumulator or Memory

CLRA
CLRB
CLR

	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
CLR	4F	2	1	0F	6	2	7F	7	3	6F	6+	2+
CLRA	5F	2	1									
CLRB												

This instruction clears a specified accumulator or a selected byte of memory — that is, it loads the accumulator or memory location with zero.

We will illustrate clearing Accumulator B:



Suppose that Accumulator B contains 43_{16} . After the processor executes the instruction CLRB, Accumulator B will contain 00_{16} . In addition, the Sign, Overflow, and Carry flags will all be 0 and the Zero flag will be 1.

CLV — Clear Overflow

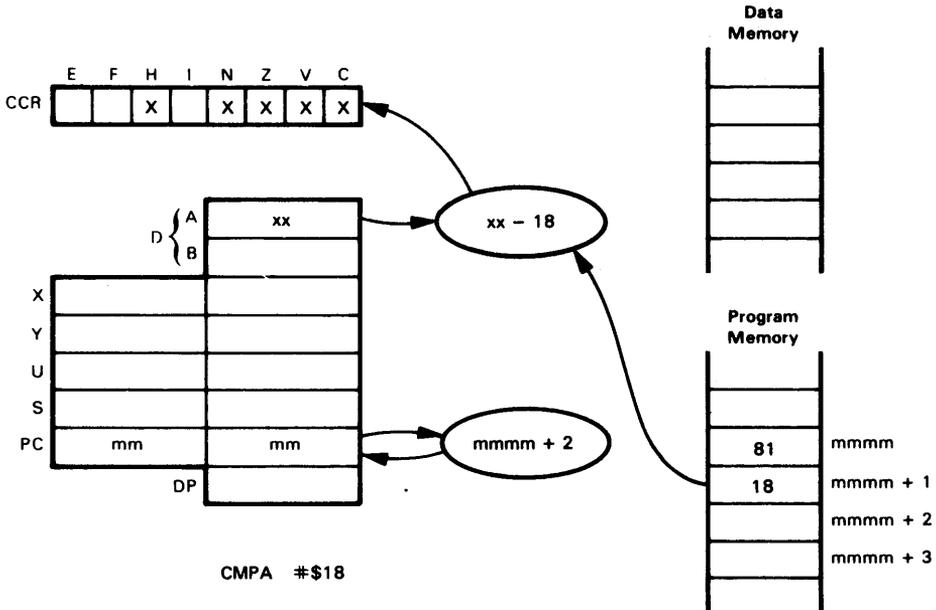
The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction ANDCC # $\%11111101$ — that is, into an instruction that clears bit 1 of the Condition Code register (the Overflow flag). No other flags or registers are affected. The Overflow flag is also cleared by many other instructions, including AND, BIT, CLR, COM, EOR, LD, OR, ST, and TST.

CMP — Compare Memory with a Register
CMPA
CMPB
CMPD
CMPS
CMPU
CMPX
CMPY

	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
CMPA	81	2	2	91	4	2	B1	5	3	A1	4+	2+
CMPB	C1	2	2	D1	4	2	F1	5	3	E1	4+	2+
CMPD	10 83	5	4	10 93	7	3	10 B3	8	4	10 A3	7+	3+
CMPS	11 8C	5	4	11 9C	7	3	11 BC	8	4	11 A3	7+	3+
CMPU	11 83	5	4	11 93	7	3	11 B3	8	4	11 A3	7+	3+
CMPX	8C	4	3	9C	6	2	BC	7	3	AC	6+	2+
CMPY	10 8C	5	4	10 9C	7	3	10 BC	8	4	10 AC	7+	3+

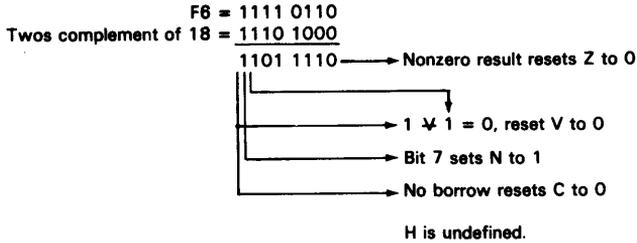
There are two forms of this instruction — an 8-bit form and a 16-bit form. The 8-bit form is associated with Accumulators A and B. The 16-bit form is associated with the 16-bit registers D, X, Y, U, and S. This instruction subtracts the contents of the selected memory location from the contents of the specified register and sets the Condition flags accordingly. Neither the contents of the memory location nor the contents of the register are changed. The Carry flag represents a borrow.

Let us begin with the 8-bit case using immediate addressing and Accumulator A.



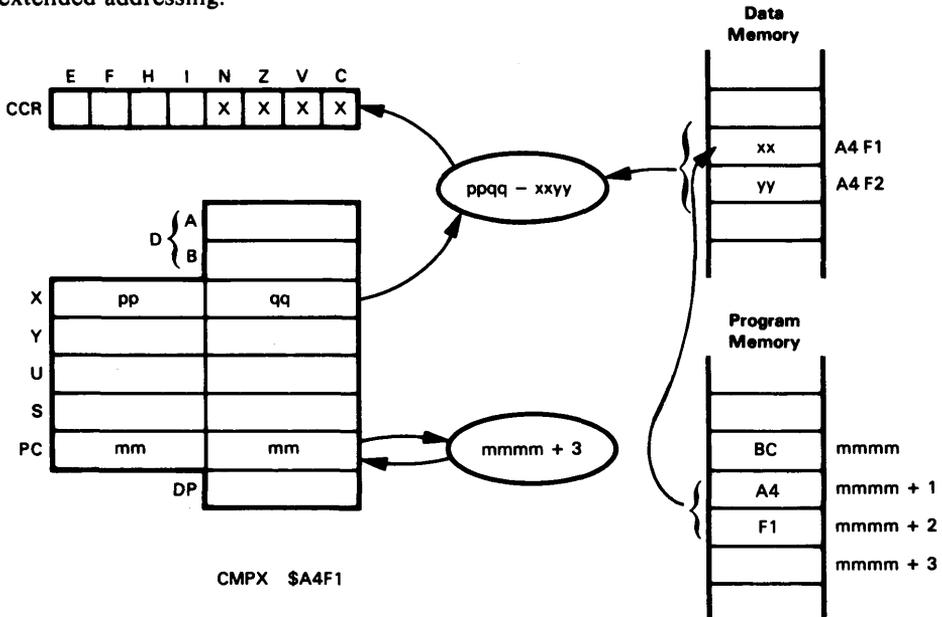
Suppose that $xx = F6_{16}$. After the processor executes the instruction `CMPA #18`,

Accumulator A will still contain $F6_{16}$, but the flags will be modified as follows:

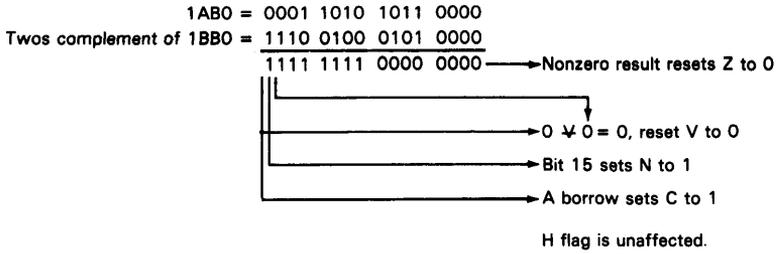


Note that C is the complement of the resulting carry since it represents a borrow. Compare instructions are most frequently used to set flags before the execution of branch instructions. Note that the half-carry flag (H) is undefined after the 8-bit CMP instruction.

Now consider the 16-bit case. The execution of the two-byte compare is the same as for the one-byte compare illustrated above with the exception that a 16-bit comparison takes place rather than an 8-bit comparison. We will illustrate CMPX using extended addressing.



Suppose that $ppq = 1AB0_{16}$, xx (the contents of memory location $A4F1$) = $1B_{16}$, and yy (the contents of memory location $A4F2$) = $B0_{16}$. After the processor executes **CMPX \$A4F1**, Index Register X and memory will be unchanged but the Sign, Zero, Overflow, and Carry flags will be modified as follows.

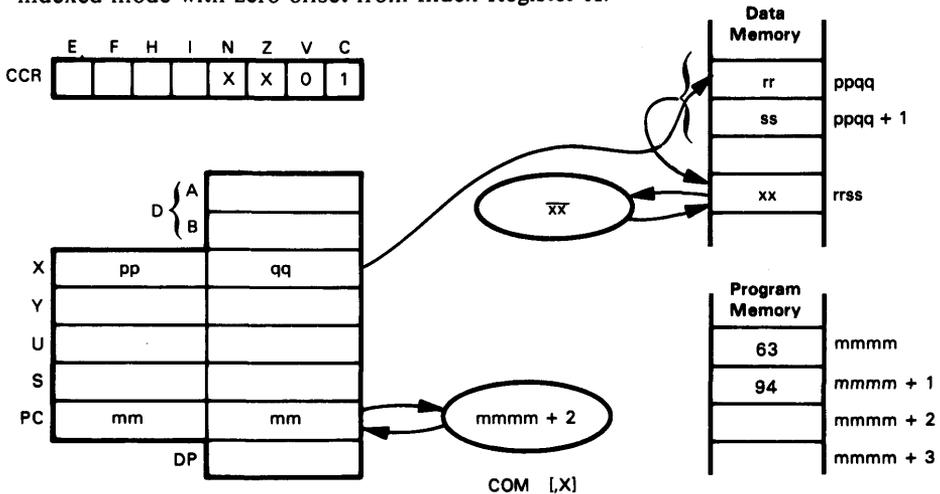


Notice that C is the complement of the resulting carry, just as in the CMPA instruction. The flags are affected by the complete 16-bit operation, not by the two 8-bit operations separately. The similar instructions (CMPD, CMPS, CMPU, and CMPY) all require 2-byte operation codes while CMPX requires only one. This means that programmers should prefer Index Register X over Index Register Y and the stack pointers to minimize the memory usage and execution time of their programs.

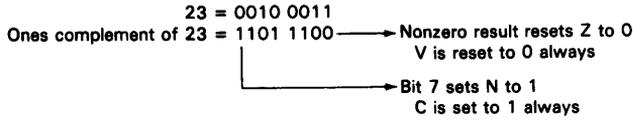
COM — Complement Accumulator or Memory
COMA
COMB
COM

	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
COM				03	6	2	73	7	3	63	6+	2+
COMA	43	2	1									
COMB	53	2	1									

This instruction complements the specified accumulator or a selected byte of memory. This is the ones complement operation, which replaces each 1 in the byte with a 0, and each 0 with a 1. We will illustrate the COM instruction using the indirect indexed mode with zero offset from Index Register X.



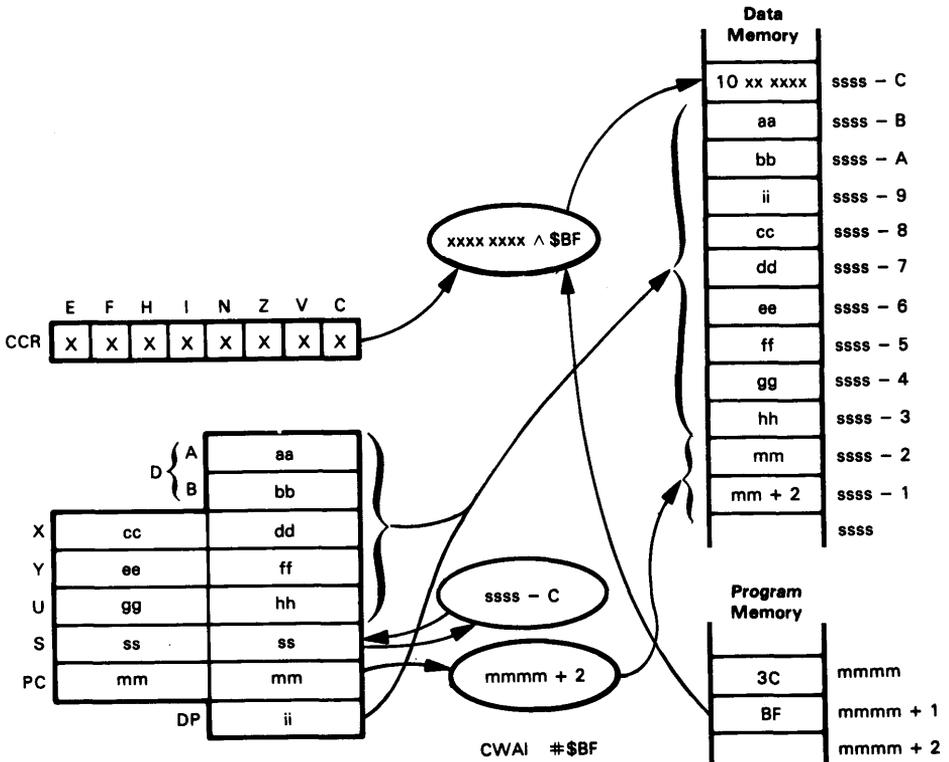
Suppose that the contents of Index Register X are 0100_{16} and that memory locations 0100 and 0101 contain 0113_{16} . If the contents of memory location 0113 are 23_{16} , then after the processor executes the instruction $COM [X]$, memory location 0113 will be changed to DC_{16} .



CWAI – Logically AND Immediate Memory with Condition Code Register and Wait for Interrupt

	Object Code	No. of Cycles	No. of Bytes
CWAI	3C	20	2

This instruction logically ANDs the contents of the following byte of program memory with the contents of the Condition Code Register, saves all the user registers in the Hardware Stack, and halts execution until an external interrupt occurs.



The logical AND immediate is performed in exactly the same way as described in the ANDCC instruction. Note that the operation on the Condition Code Register is done before stacking. The entire flag (E) is set regardless of masking. The normal use of CWAI is to clear one or more of the interrupt flags and hence enable interrupts before suspending operations. So the sensible instructions are:

```

CWA I  %#11101111  ENABLE REGULAR INTERRUPT
CWA I  %#10111111  ENABLE FAST INTERRUPT
CWA I  %#10101111  ENABLE REGULAR AND FAST INTERRUPTS
CWA I  %#11111111  WAIT FOR NONMASKABLE INTERRUPT
    
```

Remember that clearing an interrupt mask enables the interrupt from that source.

After the processor has saved the status of the system in the Hardware Stack as shown in the diagram above, it halts execution until it receives an interrupt. Note that the contents of the Hardware Stack Pointer are not stacked. When an interrupt occurs, the interrupt mask bits are set to 1 and the processor jumps to the address in the appropriate interrupt vector.

This instruction is used to synchronize the CPU with external processes. CWAI does not tri-state the system busses. A fast interrupt may enter its interrupt handler with the entire machine state saved; RTI will automatically restore the entire machine state after testing the E bit of the recovered CCR.

DAA — Decimal Adjust After Addition

	Object Code	No. of Cycles	No. of Bytes
DAA	19	2	1

Convert the contents of Accumulator A to binary-coded decimal form. Suppose that Accumulator A contains 39_{16} and memory location 15E1 contains 47_{16} . After the processor has executed the two instructions

```

ADDA  $15E1
DAA
    
```

Accumulator A will contain 86_{16} , rather than the 80_{16} which would be the ordinary binary result. The Carry flag will be reset to 0 since there was no carry; the Overflow flag is undefined; the Zero flag is reset to 0 since the result is not zero; and bit 7 sets N to 1. This is the only instruction that requires the Half-Carry flag (H); its value is needed to determine if a Carry occurred from the less significant digit.

The Sign and Zero flags are modified to reflect the statuses they represent. The Overflow flag is destroyed and the Carry flag is set or reset as it should be by a hypothetical BCD addition. That is, the Carry flag is set if the BCD sum of the more significant digits produced a carry.

Correction factors of 6 are added to each 4-bit digit of Accumulator A under the following conditions:

1. Less Significant Digit (LSD)
 - a. $H = 1$ or
 - b. $LSD > 9$

- 2. More Significant Digit (MSD)
 - a. $C = 1$ or
 - b. $MSD > 9$ or
 - c. $MSD > 8$ or $LSD > 9$

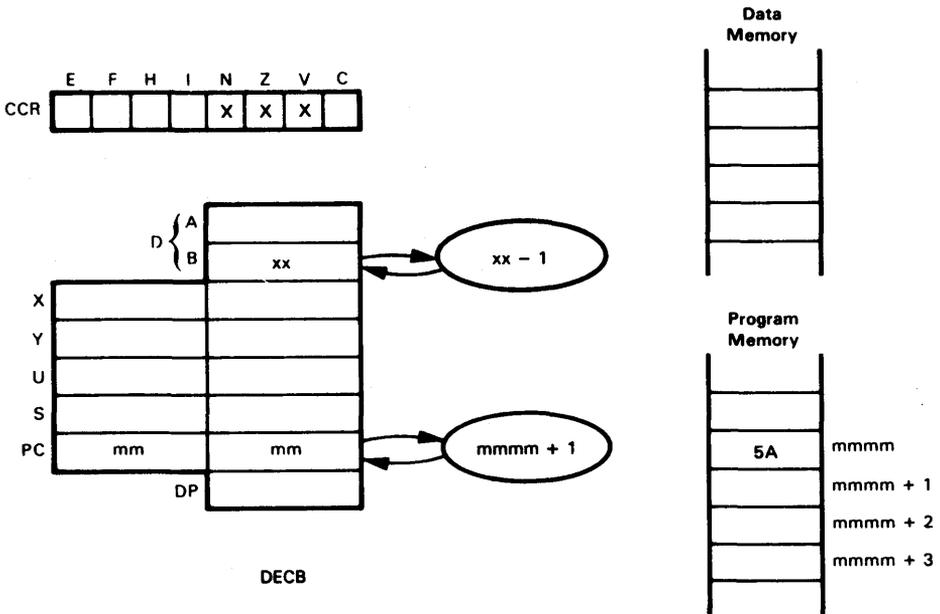
This instruction makes sense only after a binary addition instruction (ADC or ADD); the combinations ADCA, DAA or ADDA, DAA are decimal addition instructions that operate on BCD data and generate BCD results.

DEC — Decrement Accumulator or Memory
DECA
DECB
DEC

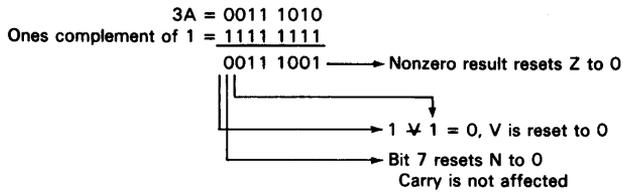
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
DEC	4A	2	1	0A	6	2	7A	7	3	6A	6+	2+
DECA DECB	5A	2	1									

This instruction decrements by one the specified 8-bit accumulator or a selected memory byte.

Let us consider the case of Accumulator B.



Suppose that Accumulator B contains $3A_{16}$. After the processor executes DECB, Accumulator B will contain 39_{16} .



The fact that DEC does not affect the Carry flag is quite important; it allows the programmer to use DEC (or INC) to count iterations of a loop that is performing multiple-precision arithmetic. The Carry flag is essential in such a loop to transfer information (carries or borrows) from one iteration to the next. Decrementing a register or memory location that contains zero produces a result of FF₁₆ but does not set the Carry flag.

After decrements of unsigned values, only BEQ and BNE branches will behave consistently. However, when the operands are two's complement numbers, all signed branches behave properly.

DES — Decrement Hardware Stack Pointer by 1

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction LEAS -1,S — that is, into an instruction that subtracts 1 from the Hardware Stack Pointer. Note that DES provides a 16-bit decrement that does not affect the flags.

DEX — Decrement Index Register X by 1

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction LEAX -1,X — that is, into an instruction that subtracts 1 from Index Register X. Note that DEX provides a 16-bit decrement that affects only the Zero flag.

DEY — Decrement Index Register Y by 1

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction LEAY -1,Y — that is, into an instruction that subtracts 1 from the contents of Index Register Y. Note that DEY provides a 16-bit decrement that affects only the Zero flag.

EOR — Logically Exclusive-OR Memory with Accumulator

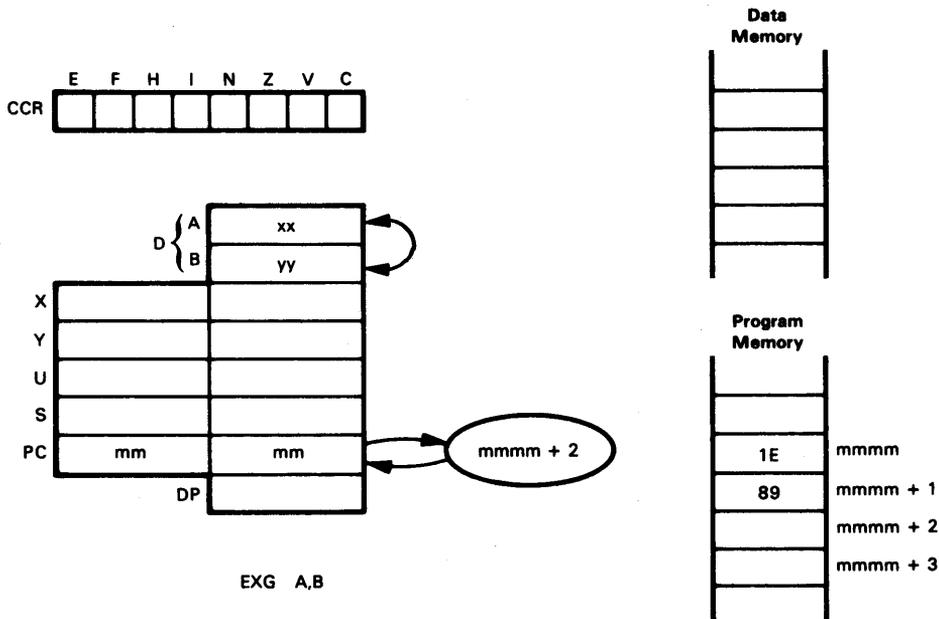
EORA

EORB

	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
EORA	88	2	2	98	4	2	B8	5	3	A8	4+	2+
EORB	C8	2	2	D8	4	2	F8	5	3	E8	4+	2+

that registers may only be exchanged with registers of like size, that is 8-bit with 8-bit and 16-bit with 16-bit.

We will illustrate the execution of the EXG A,B instruction.



Suppose that $xx = 7E_{16}$ and $yy = A5_{16}$; then after the processor executes the EXG A,B instruction, the contents of Accumulator A will be $A5_{16}$ and the contents of Accumulator B will be $7E_{16}$.

The EXG instruction has many miscellaneous applications; for example,

1. **Exchanging accumulators — EXG A,B**
Remember, for example, that only Accumulator A can be operated on with the DAA instruction.
2. **Calling subroutines with a link register — EXG PC,X**
This instruction transfers control to the address in Index Register X and saves the old value of the Program Counter in Index Register X.
3. **Changing the direct page — EXG A,DP**
This instruction not only places the value from Accumulator A in the Direct Page Register, but it also saves the old value of the Direct Page Register in Accumulator A. Note that there is no LD instruction for the Direct Page Register.

EXG is, of course, symmetric — for instance, EXG A,B and EXG B,A are the same operation. Note that all EXG instructions require two bytes of memory — one for the operation code and one for a post byte (immediate data) to specify which registers are to be exchanged. Be careful of the fact that some EXG instructions are meaningless (undefined register codes), while others are illegal (exchanging registers with different lengths).

The post byte definition is as follows:



Bit patterns defining the registers are as follows:

Register	Binary	Hex	Register	Binary	Hex
D	0000	0	PC	0101	5
X	0001	1	A	1000	8
Y	0010	2	B	1001	9
U	0011	3	CCR	1010	A
S	0100	4	DP	1011	B

The remaining bit patterns are undefined.

Returning to the previous example, the post byte was determined as follows:

```

Source:   EXG  A,B
          |
Binary:   1000 1001
          |
Hexadecimal: 1E  8  9
    
```

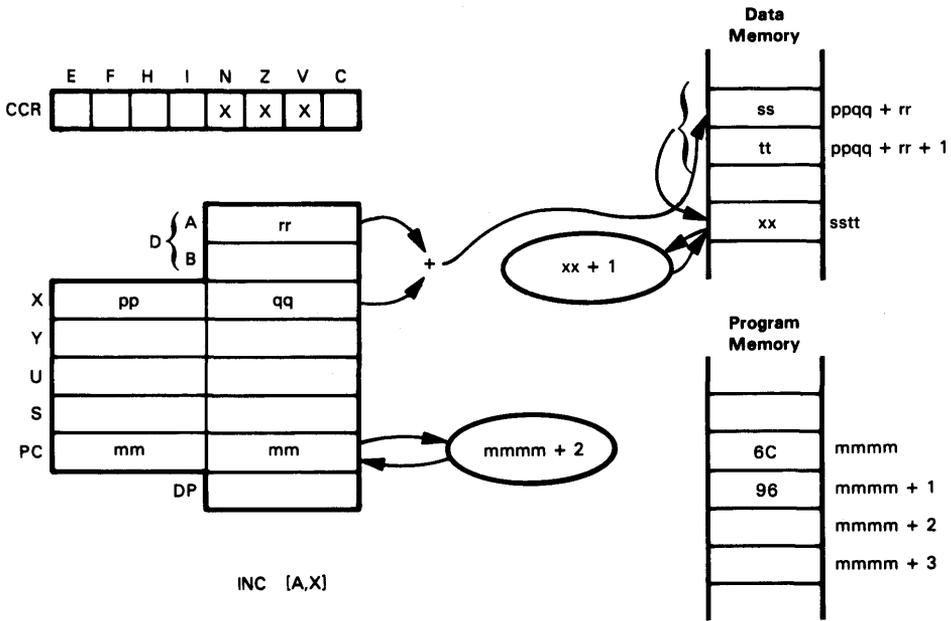
Bits 0 through 3 of the immediate byte of the instruction define one register, while bits 4 through 7 define the other. The condition codes are not affected unless one of the registers is the Condition Code Register itself (CCR).

INC — Increment Accumulator or Memory Location by 1
INCA
INCB
INC

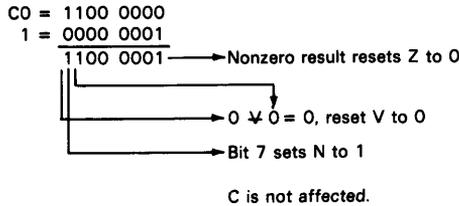
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
INC				0C	6	2	7C	7	3	6C	6+	2+
INCA	4C	2	1									
INCB	5C	2	1									

This instruction adds one to the contents of the specified 8-bit accumulator or byte of memory.

Consider incrementing a memory location addressed by indirect indexed mode with Accumulator A offset from Index Register X.



If $ppqq = 150A_{16}$, $rr = FE_{16}$, $sstt = 25E4_{16}$, and $xx = C0_{16}$, then after the processor executes the instruction `INC [A,X]`, it will have changed the contents of memory locations $25E4$ to $C1_{16}$. Note that rr is interpreted as a two's complement number, so $ppqq + rr = 1508_{16}$.



The fact that `INC` does not affect the Carry flag is quite important; it allows the programmer to use `INC` (or `DEC`) to count iterations of a loop that is performing multiple-precision arithmetic. The Carry flag is essential in such a loop to transfer information (carries or borrows) from one iteration to the next. Incrementing a register or memory location that contains FF_{16} produces a result of 00_{16} but does not set the Carry flag; however, it does set the Zero flag.

`INC` and `DEC` are commonly used to count occurrences of events or numbers of iterations. `INC` is more commonly (and more logically) used for event counting, while `DEC` is more commonly used for looping since the Zero flag is then available as a convenient exit condition.

After increments of unsigned values, only `BNE` and `BEQ` branches will behave consistently. When the operands are two's complement values, all signed branches function correctly.

Note that this instruction does not apply to the Double Accumulator D. Thus, the only accumulator forms implemented are `INCA` and `INCB`.

INS — Increment Hardware Stack Pointer by 1

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction LEAS 1,S — that is, into an instruction that adds 1 to the contents of the Hardware Stack Pointer. Note that INS performs a 16-bit increment that does not affect any flags.

INX — Increment Index Register X by 1

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction LEAX 1,X — that is, into an instruction that adds 1 to the contents of Index Register X. Note that INX performs a 16-bit increment that affects only the Zero flag.

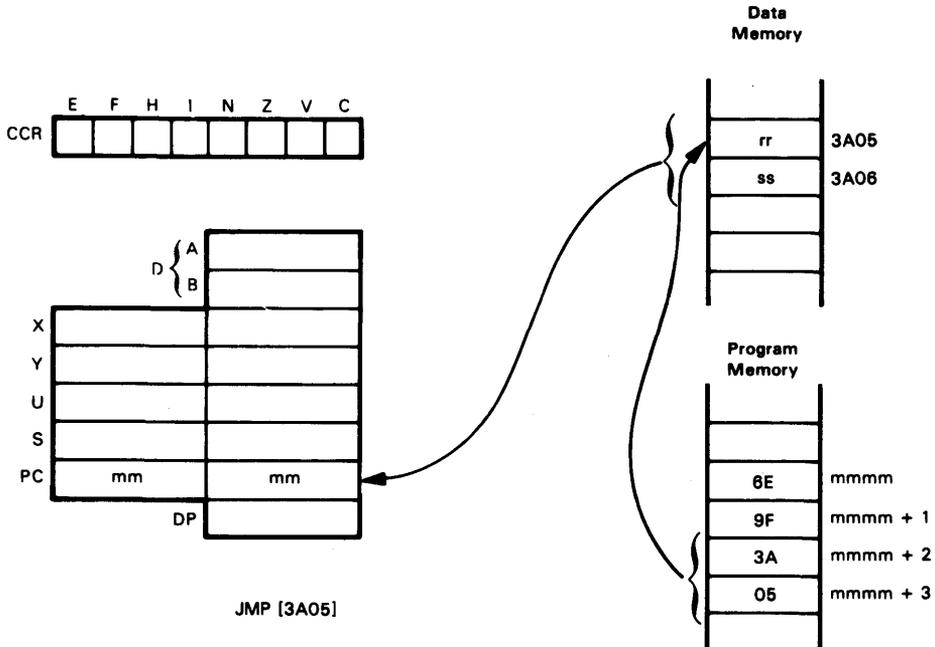
INY — Increment Index Register Y by 1

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction LEAY 1,Y — that is, into an instruction that adds 1 to the contents of Index Register Y. Note that INY performs a 16-bit increment that affects only the Zero flag.

JMP — Unconditional Jump

	Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
JMP	0E	3	2	7E	4	3	6E	3+	2+

Jump to the specified memory address. This instruction provides an unconditional absolute jump capability, as contrasted to the unconditional relative jump capability provided by BRA and LBRA. We will illustrate its execution using the extended indirect mode, but you should note that it can use base page direct, extended direct, or any of the indexed addressing modes.



If, for example, $rrss = D1E5$, then after the processor executes the instruction `JMP [$3A05]`, the Program Counter will contain $D1E5$ and execution will continue from that point.

The terminology here is somewhat confusing (as on most computers) — `JMP` with extended addressing transfers control to the extended address, not to its contents. So `JMP` with extended addressing is similar to other instructions (such as `LD`) with immediate addressing. For example, `JMP $3E08` transfers control to (loads the program counter with) $3E08_{16}$, not the contents of that address. `JMP` with indexed addressing is executed in a similar manner, as if one level of indirection had been removed.

In the following instruction sequence:

```
LDX  INDEX      GET INDEX FOR JUMP TABLE
JMP  [JTABL,X]  JUMP TO APPROPRIATE TABLE ENTRY
```

the `JMP` instruction will perform an indexed jump into a table of addresses starting at `JTABL`, with the index given by the contents of memory addresses `INDEX` and `INDEX+1`. Some part of the program preceding `LDX` must double the contents of `INDEX` and `INDEX+1` to account for the fact that all addresses occupy two bytes.

Note the distinction from the instruction sequence

```
LDX  INDEX      GET INDEX FOR TABLE OF BRANCHES
JMP  JTABL,X    JUMP TO APPROPRIATE BRANCH INSTRUCTION
```

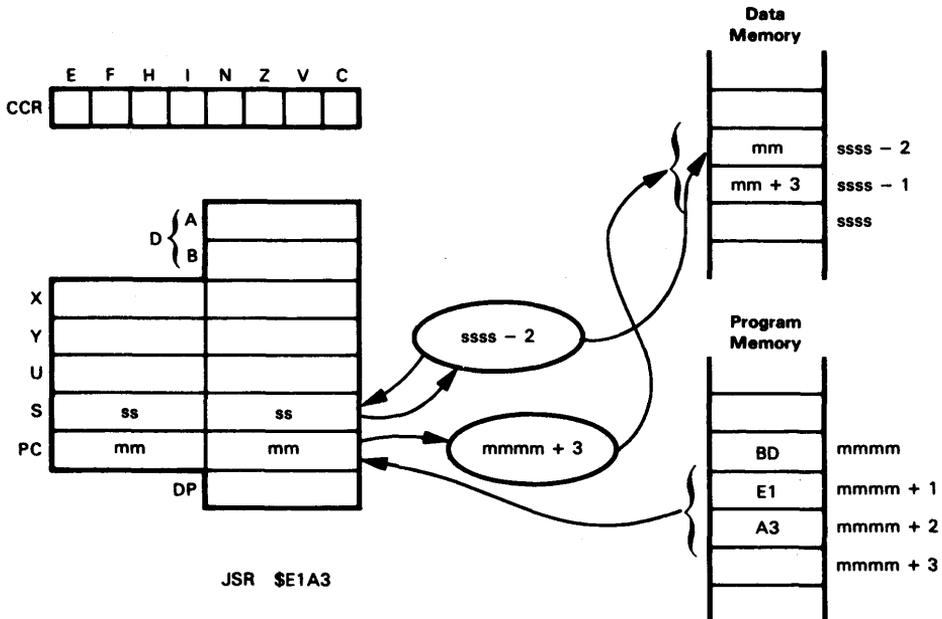
In this sequence, the `JMP` instruction will transfer control to the appropriate position in the table (base address `JTABL`, index given by the contents of memory addresses `INDEX` and `INDEX+1`). That position must contain a `JMP` or branch instruction transferring control to the appropriate routine, rather than just the address of the routine as in the earlier case.

JSR — Jump to Subroutine

	Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
JSR	9D	7	2	8D	8	3	AD	7+	2+

Jump unconditionally to the specified memory address, saving the old value of the Program Counter on the hardware Stack.

We will illustrate the execution of JSR with extended addressing.



If, for example, $mmmm = E56B_{16}$ and $ssss = 08A0_{16}$, then after the processor executes the instruction `JSR $E1A3`, the program counter will contain $E1A3_{16}$ and execution will continue from that point in memory. The value of the program counter at the end of the JSR instruction ($E56B + 0003 = E56E$) will have been saved in the hardware Stack the same way it is saved during the BSR instruction. The final value of the Hardware Stack Pointer will be 2 less than its original value.

JSR is the same as JMP, except that JSR saves the old value of the program counter in the hardware Stack, thus providing a subroutine linkage. An RTS instruction at the end of the subroutine can transfer control back to the instruction immediately following JSR, providing that the subroutine has not changed the return address or the Hardware Stack Pointer. JSR provides an unconditional absolute jump-to-subroutine capability, as compared to the relative jump-to-subroutine capability provided by BSR and LBSR.

JSR, like JMP, can be used to handle jump tables. The only difference is that the return address is saved at the top of the Stack. The same terminology confusion exists

here as with JMP; JSR with extended addressing transfers control to the extended address, not to its contents. All the indexed modes operate similarly, as if one level of indirection had been removed.

LBCC — Long Branch If Carry Clear (C = 0)

	Object Code	No. of Cycles	No. of Bytes
LBCC	10 24	5(6)	4

LBCC is the same as LBRA except that it branches under the same condition as does BCC. LBCC is a 4-byte instruction using 16-bit relative addressing while BCC is a 2-byte instruction using 8-bit relative addressing. See LBRA and BCC for details on the operation of LBCC.

LBCS — Long Branch If Carry Set (C = 1)

	Object Code	No. of Cycles	No. of Bytes
LBCS	10 25	5(6)	4

LBCS is the same as LBRA except that it branches under the same condition as does BCS. LBCS is a 4-byte instruction using 16-bit relative addressing while BCS is only a 2-byte instruction using 8-bit relative addressing. See LBRA and BCS for details on the operation of LBCS.

LBEQ — Long Branch If Equal To Zero (Z = 1)

	Object Code	No. of Cycles	No. of Bytes
LBEQ	10 27	5(6)	4

LBEQ is the same as LBRA except that it causes a branch under the same condition as does BEQ. LBEQ is a 4-byte instruction using 16-bit relative addressing while BEQ is a 2-byte instruction using 8-bit relative addressing. See LBRA and BEQ for details on the operation of LBEQ.

LBGE — Long Branch If Greater Than or Equal To Zero ($N \oplus V = 0$)

	Object Code	No. of Cycles	No. of Bytes
LBGE	10 2C	5(6)	4

LBGE is the same as LBRA except that it causes a branch under the same condition as does BGE. LBGE is a 4-byte instruction using 16-bit relative addressing while

BGE is a 2-byte instruction using 8-bit relative addressing. See LBRA and BGE for details on the operation of LBGE.

LBGT — Long Branch If Greater Than Zero ($Z + (N \oplus V) = 0$)

	Object Code	No. of Cycles	No. of Bytes
LBGT	10 2E	5(6)	4

LBGT is the same as LBRA except that it causes a branch under the same condition as does BGT. LBGT is a 4-byte instruction using 16-bit relative addressing while BGT is a 2-byte instruction using 8-bit relative addressing. See LBRA and BGT for details on the operation of LBGT.

LBHI — Long Branch If Higher ($Z + C = 0$)

	Object Code	No. of Cycles	No. of Bytes
LBHI	10 22	5(6)	4

LBHI is the same as LBRA except that it causes a branch under the same conditions as does BHI. LBHI is a 4-byte instruction using 16-bit relative addressing while BHI is a 2-byte instruction using 8-bit relative addressing. See LBRA and BHI for details on the operation of LBHI.

LBHS — Long Branch If Higher or the Same ($C = 0$)

	Object Code	No. of Cycles	No. of Bytes
LBHS	10 24	5(6)	4

LBHS is the same as LBRA except that it causes a branch under the same condition as does BHS. LBHS is a 4-byte instruction using 16-bit relative addressing while BHS is a 2-byte instruction using 8-bit relative addressing. See LBRA and BHS for details on the operation of LBHS.

LBLE — Long Branch If Less Than or Equal To Zero ($Z + (N \oplus V) = 1$)

	Object Code	No. of Cycles	No. of Bytes
LBLE	10 2F	5(6)	4

LBLE is the same as LBRA except that it causes a branch under the same condition as does BLE. LBLE is a 4-byte instruction using 16-bit relative addressing while BLE is a 2-byte instruction using 8-bit relative addressing. See LBRA and BLE for details on the operation of LBLE.

LBLO — Long Branch If Lower (C = 1)

	Object Code	No. of Cycles	No. of Bytes
LBLO	10 25	5(6)	4

LBLO is the same as LBRA except that it causes a branch under the same condition as does BLO. LBLO is a 4-byte instruction using 16-bit relative addressing while BLO is a 2-byte instruction using 8-bit relative addressing. See LBRA and BLO for details on the operation of LBLO.

LBLS — Long Branch If Lower or Same (C + Z = 1)

	Object Code	No. of Cycles	No. of Bytes
LBLS	10 23	5(6)	4

LBLS is the same as LBRA except that it causes a branch under the same condition as does BLS. LBLS is a 4-byte instruction using 16-bit relative addressing while BLS is a 2-byte instruction using 8-bit relative addressing. See LBRA and BLS for details on the operation of LBLS.

LBLT — Long Branch If LESS Than Zero (N ⊕ V = 1)

	Object Code	No. of Cycles	No. of Bytes
LBLT	10 2D	5(6)	4

LBLT is the same as LBRA except that it causes a branch under the same condition as does BLT. LBLT is a 4-byte instruction using 16-bit relative addressing while BLT is a 2-byte instruction using 8-bit relative addressing. See LBRA and BLT for details on the operation of LBLT.

LBMI — Long Branch If Minus (N = 1)

	Object Code	No. of Cycles	No. of Bytes
LBMI	10 2B	5(6)	4

LBMI is the same as LBRA except that it causes a branch under the same condition as does BMI. LBMI is a 4-byte instruction using 16-bit relative addressing while BMI is a 2-byte instruction using 8-bit relative addressing. See LBRA and BMI for details on the operation of LBMI.

LBNE — Long Branch If Not Equal To Zero (Z = 0)

	Object Code	No. of Cycles	No. of Bytes
LBNE	10 26	5(6)	4

LBNE is the same as LBRA except that it causes a branch under the same condition as does BNE. LBNE is a 4-byte instruction using 16-bit relative addressing while BNE is a 2-byte instruction using 8-bit relative addressing. See LBRA and BNE for details on the operation of LBNE.

LBPL — Long Branch If Plus (N = 0)

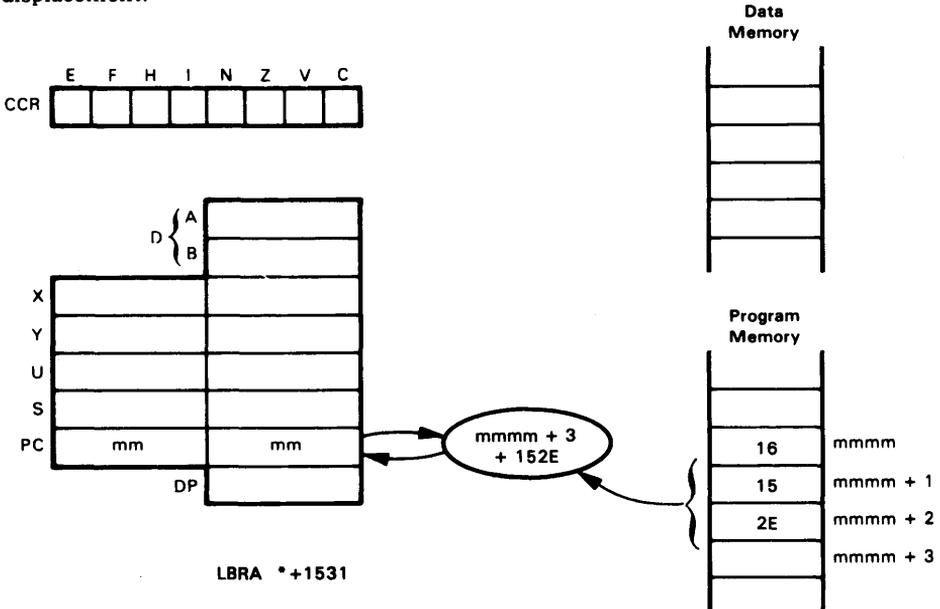
	Object Code	No. of Cycles	No. of Bytes
LBPL	10 2A	5(6)	4

LBPL is the same as LBRA except that it causes a branch under the same condition as does BPL. LBPL is a 4-byte instruction using 16-bit relative addressing while BPL is a 2-byte instruction using 8-bit relative addressing. See LBRA and BPL for details on the operation of LBPL.

LBRA — Long Branch Always

	Object Code	No. of Cycles	No. of Bytes
LBRA	16	5	3

LBRA places the specified address in the Program Counter, thus always causing a program branch. The specified address is the sum of the current value of the Program Counter (after the processor has fetched the LBRA instruction from memory) and the displacement.



If $mmmm = 1023_{16}$, then after execution of `LBRA *+$1531`, the Program Counter would contain $1023 + 3 + 152E = 2554_{16}$ and execution would continue with the instruction at that location.

The displacement — the contents of the second and third bytes of the instruction — forms a 16-bit twos complement number. Thus the overall effect of an `LBRA` instruction is:

$$PC = PC + 3 + disp$$

The extra factor of 3 is the result of the 3 bytes occupied by the `LBRA` instruction itself. `LBRA` does not affect any flags or registers except the Program Counter (its previous value is lost). Note that `LBRA` requires only a 1-byte operation code, while the various conditional long branches (`LBCC`, `LBCS`, `LBEQ`, etc.) require 2-byte operation codes. Thus the displacement formula for the long conditional branches is

$$PC = PC + 4 + disp$$

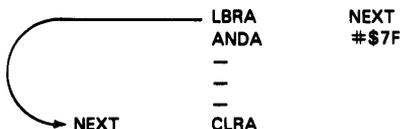
Since the displacement is now a 16-bit twos complement number, its range has increased to

$$-32768_{10}(1000\ 0000\ 0000\ 0000) \leq disp \leq +32767_{10}(0111\ 1111\ 1111\ 1111_2)$$

the range of the long branches is therefore

$$* - 32764_{10} \leq destination \leq * + 32771_{10}$$

where `*` refers to the value of the Program Counter at the start of the instruction. Consider the following section of a program:



After the `LBRA` instruction, the processor will always execute the `CLRA` instruction next. It will never execute the `ANDA` instruction unless a branch or jump instruction somewhere else in the program jumps to that instruction.

See the description of `BRA` for the short relative form, used when the destination is close enough for a one-byte offset.

LBRN — Long Branch Never

	Object Code	No. of Cycles	No. of Bytes
LBRN	10 21	5	4

`LBRN` is the same as `LBRA` except that, like `BRN`, no branch ever occurs. Thus `LBRN` is essentially a no-operation; that is, control always passes to the next instruction with no other changes ever occurring. Note that `LBRN` is a 4 byte no-op, since it requires a 2-byte operation code followed by a 2-byte relative address (`BRN` is a 2-byte no-op). Of course, the relative address could have any value, since it will never be used. `LBRN` is useful as a byte filler or for tuning a delay routine. Generally it is not a very useful instruction; it makes the set of long branches logically complete. See the description of `NOP` for a discussion of uses for no-operations.

LBRSR — Long Branch to Subroutine

	Object Code	No. of Cycles	No. of Bytes
LBRSR	17	9	3

LBRSR is the same as LBRA except that it saves the value of the Program Counter in the same fashion as BSR. LBRSR offers 16-bit relative addressing while BSR offers 8-bit relative addressing. Note that LBRSR requires a one-byte operation code as does LBRA. See LBRA and BSR for details on the operation of LBRSR.

LBVC — Long Branch If Overflow Clear (V = 0)

	Object Code	No. of Cycles	No. of Bytes
LBVC	10 28	5(6)	4

LBVC is the same as LBRA except that it branches under the same condition as BVC. LBVC is a 4-byte instruction using 16-bit relative addressing while BVC is a 2-byte instruction using 8-bit relative addressing. See LBRA and BVC for details on the operation of LBVC.

LBVS — Long Branch If Overflow Set (V = 1)

	Object Code	No. of Cycles	No. of Bytes
LBVS	10 29	5(6)	4

LBVS is the same as LBRA except that it branches under the same condition as BVS. LBVS is a 4-byte instruction using 16-bit relative addressing while BVS is a 2-byte instruction using 8-bit relative addressing. See LBRA and BVS for details on the operation of LBVS.

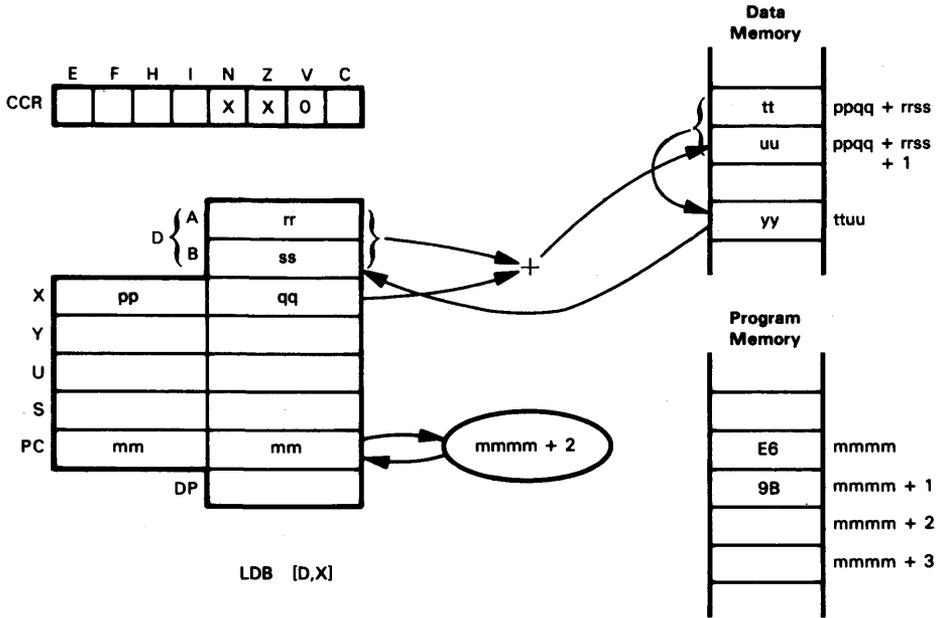
LD — Load Register from Memory

LDA
LDB
LDD
LDS
LDU
LDX
LDY

	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
LDA	86	2	2	96	4	2	B6	5	3	A6	4+	2+
LDB	C6	2	2	D6	4	2	F6	5	3	E6	4+	2+
LDD	CC	3	3	DC	5	2	FC	6	3	EC	5+	2+
LDS	10 CE	4	4	10 DE	6	3	10 FE	7	4	10 EE	6+	3+
LDU	CE	3	3	DE	5	2	FE	6	3	EE	5+	2+
LDX	8E	3	3	9E	5	2	BE	6	3	AE	5+	2+
LDY	10 8E	4	4	10 9E	6	3	10 BE	7	4	10 AE	6+	3+

This instruction loads the contents of the selected memory byte(s) into the specified register and sets the Condition flags accordingly. **There are two forms of this instruction, an 8-bit form and a 16-bit form.** The 8-bit form is associated with Accumulators A and B, while the 16-bit form is associated with the 16-bit registers D, X, Y, S, and U.

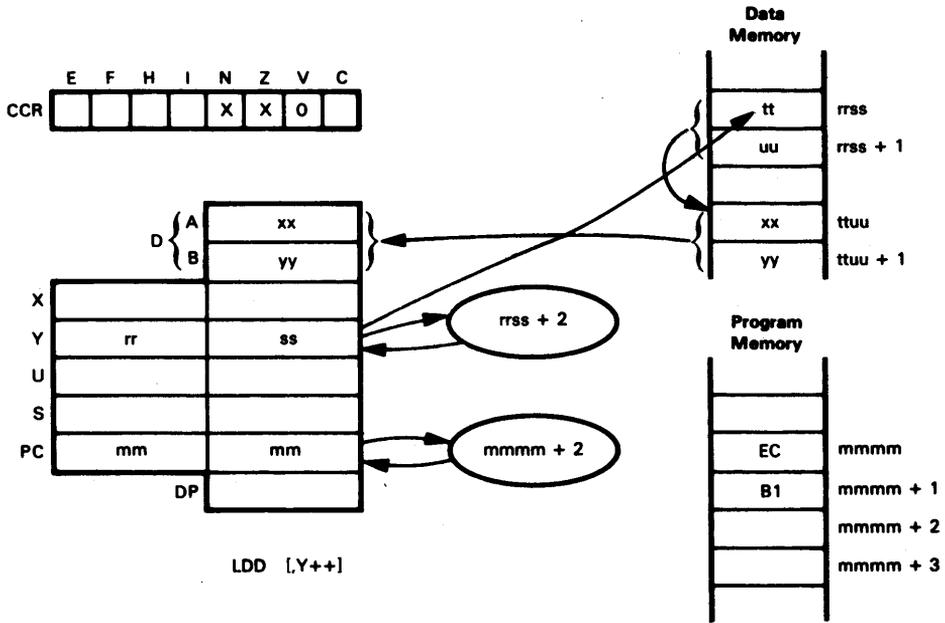
Consider an 8-bit LD using indirect indexed addressing with a double accumulator (D) offset from Index Register X; Accumulator B receives the data.



Suppose that $ppqq = 13E1_{16}$, $rrss = 20B8_{16}$, $ttuu$ (the contents of memory locations 3499 and 349A) = $A47D_{16}$, and yy (the contents of memory location $A47D$) = AA_{16} . After the processor executes the instruction LDB [D,X], Accumulator B will contain AA_{16} .

AA = 1010 1010 → Nonzero result resets Z to 0
 Bit 7 sets N to 1 → V is cleared

Now consider the 16-bit LD of the double accumulator using indirect indexed addressing with autoincrementing (by 2) of Index Register Y.



Suppose that $rrss = 8E05_{16}$, $ttuu$ (the contents of memory locations $8E05$ and $8E06$) = $B394_{16}$, xx (the contents of memory location $B394$) = 07_{16} , and yy (the contents of memory location $B395$) = $F2_{16}$. After the processor has executed the instruction **LDD [Y++]**, Accumulator A will contain 07_{16} , Accumulator B will contain $F2_{16}$ (thus Accumulator D will contain $07F2_{16}$), and Index Register Y will contain $8E05_{16} + 2 = 8E07_{16}$.

$07F2 = 0000\ 0111\ 1111\ 0010 \rightarrow$ Nonzero result resets Z to 0

Bit 15 resets N to 0

V is cleared always.

Note that the flags are affected according to the 16-bit value being loaded and not the separate 8-bit values.

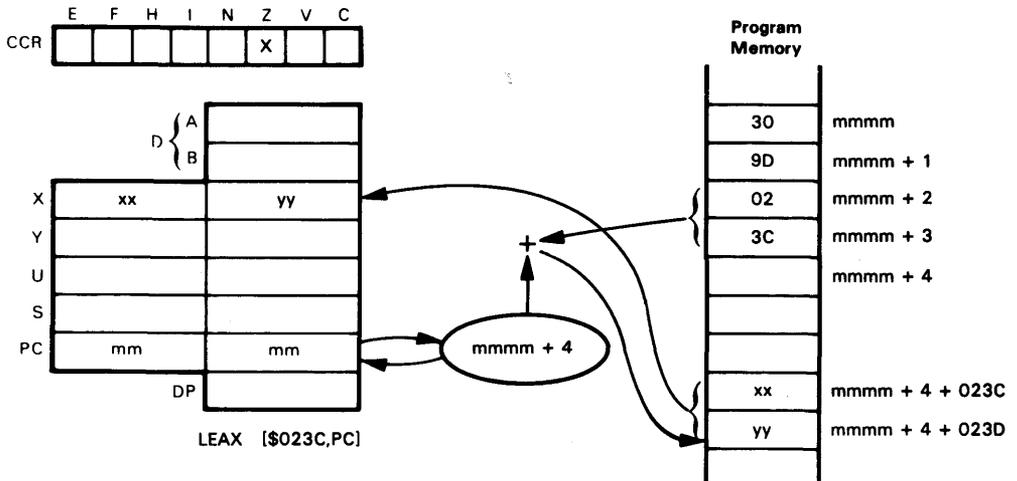
LEA — Load Effective Address Into 16-Bit Register

LEAS
LEAU
LEAX
LEAY

	Object Code	No. of Cycles	No. of Bytes
LEAS	32	4+	2+
LEAU	33	4+	2+
LEAX	30	4+	2+
LEAY	31	4+	2+

Form an effective address using one of the indexed addressing modes. Load that address into a 16-bit register (Index Register X, Index Register Y, Hardware Stack Pointer S, or User Stack Pointer U).

We will illustrate LEA using the indirect indexed mode based on a 16-bit constant offset from the Program Counter. The result is stored in Index Register X.



Suppose that $mmmm = E385_{16}$, xx (the contents of memory address $E389 + 023C = E5C5$) = DE_{16} , and yy (the contents of memory address $E5C6$) = $2F_{16}$. After the processor executes the instruction `LEAX [$023C,PC]`, Index Register X will contain $DE2F_{16}$ and the Zero flag will be cleared since the result is not zero. Note that `LEAX` and `LEAY` affect the Zero flag, while `LEAU` and `LEAS` do not; this difference is necessary to maintain compatibility with the 6800 microprocessor. The 6809 assembler translates the 6800 instructions `DEX` and `INX` into `LEAX` instructions (`LEAX -1,X` and `LEAX 1,X` respectively); 6800 programs often use `DEX` or `INX` for counting purposes and employ the zero flag as an exit condition.

The LEA instruction brings to the programmer a great deal more capability than a casual examination would suggest. It permits the easy generation of position-independent code and simplifies the handling of local data on stacks, as well as permitting the implementation of several other interesting and useful operations.

The following program segment illustrates one use of `LEA`. The table of values is located $10D_{16}$ bytes from the occurrence of the `LEAX` instruction. At assembly time, the assembler computes this offset and inserts it as the two bytes following the `LEAX` code. Note that the post byte for the two-byte offset case is $8D$. Note also that the offset is the distance from the updated PC following execution of `LEAX`.

0100	30	8D	0109	START	LEAX	TABLE,PCR
0104	A6	80		LOOP	LDA	,X+
0106
.
.
020D	.	.		TABLE	FCC	/TABLE OF CHARACTERS/

Assume that the program is stored at the locations shown. During execution, the offset 0109 is added to the updated program counter value (0104) to yield address `TABLE` ($020D$). This value is loaded into Index Register X, rather than output on the address bus. When the indexed instruction `LDA ,X+` is executed, this newly computed address (stored in the index register) is used to access data from the table.

LEA can also be used to perform arithmetic on the contents of index registers and stack pointers. For example,

1. LEAX 1,X increments (adds 1 to) the contents of Index Register X. Similarly, LEAY -1,Y decrements (subtracts 1 from) the contents of Index Register Y.
2. LEAU \$2C05,U adds 2C05₁₆ to the contents of the User Stack Pointer.
3. LEAX 0,PC loads the Program Counter value at the end of the instruction into Index Register X. Note that, in position-independent code, it may be essential to determine that value and make it readily available for use in addressing.

LSL — Shift Accumulator or Memory Byte Left Logically

LSLA
LSLB
LSL

	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
LSL	48	2	1	08	6	2	78	7	3	68	6+	2+
LSLA LSLB	58	2	1									

Perform a one-bit logical left shift of the contents of Accumulator A or B or the contents of a selected byte of memory. This instruction is exactly the same as Arithmetic Shift Left or ASL; consult ASL for a description of its execution. The mnemonic is available for the sake of completeness.

LSR — Shift Accumulator or Memory Byte Right Logically

LSRA
LSRB
LSR

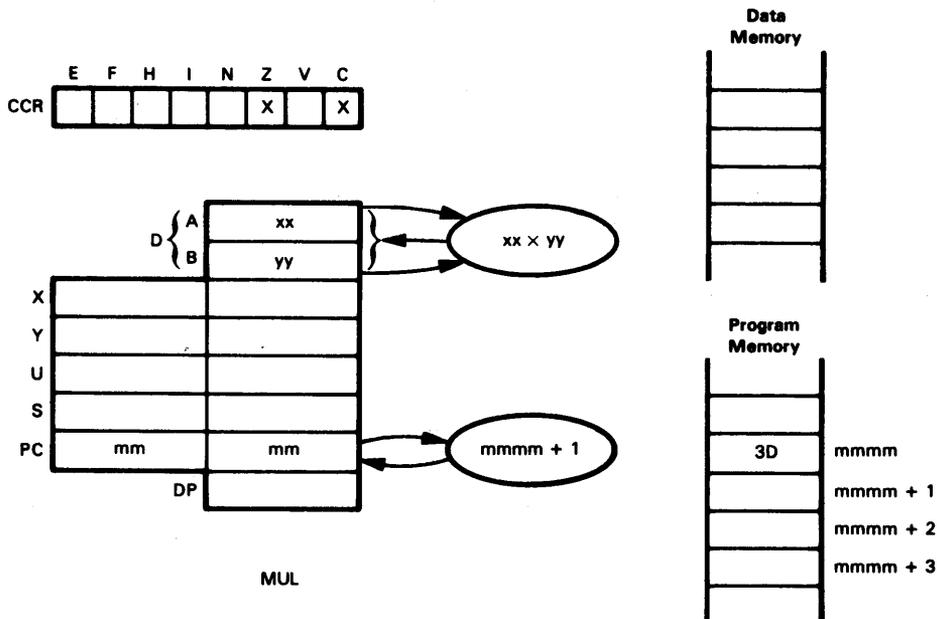
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
LSR	44	2	1	04	6	2	74	7	3	64	6+	2+
LSRA LSRB	54	2	1									

The LSR instruction is identical to the ASR instruction except that LSR causes a 0 to be shifted into bit 7 instead of keeping it intact as does the ASR instruction. Flags are affected the same way by both instructions except for the H flag, which is not affected by LSR. Of course in the LSR instruction, the N flag is cleared since bit 7 is cleared. Consult ASR for more details on LSR.

MUL — Multiply Unsigned Numbers

	Object Code	No. of Cycles	No. of Bytes
MUL	3D	11	1

Multiply the unsigned number in Accumulator A by the unsigned number in Accumulator B. Place the result in both accumulators with the most significant bits in Accumulator A — that is, Accumulator D holds the result.



If, for example, Accumulator A contains $6F_{16}$ and Accumulator B contains 61_{16} , after the processor executes the MUL instruction, Accumulator A will contain $2A_{16}$ and Accumulator B will contain $0F_{16}$ (that is, Accumulator D will contain $2A0F_{16}$).

$$\begin{array}{r}
 6F = 0110\ 1111 \\
 61 = 0110\ 0001 \\
 \hline
 \ 1111 \\
 0\ 1101\ 111 \\
 01\ 1011\ 11 \\
 \hline
 0010\ 1010\ 0000\ 1111
 \end{array}$$

→ Nonzero result resets Z to 0.
 → Bit 7 resets C to 0.

Only two flags are affected by MUL:

1. The Zero flag is set if the entire result is zero and cleared otherwise.
2. The Carry flag is set to the final value of bit 7 of Accumulator B, thus allowing rounding to an 8-bit result in A with the sequence:

```

MUL      MULTIPLY
ADCA    #0  ROUND TO 8 BITS
  
```

NEG — Twos Complement (Negate) Accumulator or Memory

NEGA

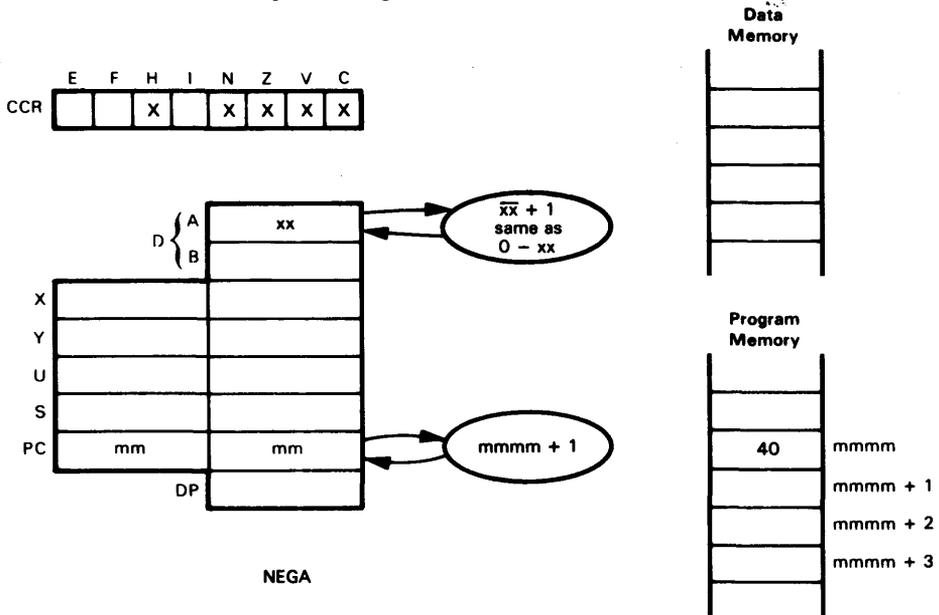
NEGB

NEG

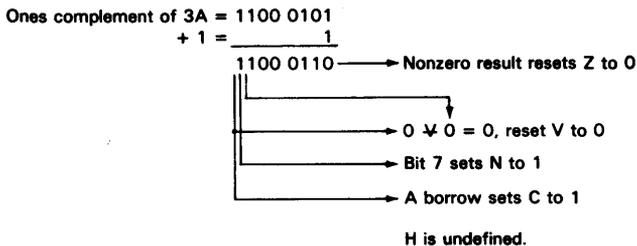
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
NEG	40	2	1	00	6	2	70	7	3	80	6+	2+
NEGA												
NEGB	50	2	1									

This instruction replaces the contents of the selected accumulator or the specified byte of memory with its twos complement. The twos complement of a number is the value that, when added to the original number, produces a sum of zero. The twos complement of n is thus $0 - n$.

Consider twos complementing Accumulator A.



If, for example, Accumulator A contains $3A_{16}$, after the processor executes the NEGA instruction, Accumulator A will contain $C6_{16}$.



The Carry flag (C) represents a borrow and is set to the complement of the resulting binary carry. The V flag is set if and only if the original operand was $1000\ 0000_2$. The value 00_{16} is replaced by itself, and only in this case is C cleared.

In the illustration above, we defined the two's complement as the ordinary (ones) complement plus 1. The sum of any number and its ones complement must have ones in every bit position, since any position that is 0 in the original number will be 1 in the ones complement and vice versa. Adding 1 to the number with ones in every bit position gives a sum of zero (with a carry, which is ignored), so adding 1 to the ones complement must give the two's complement.

NOP — No Operation

	Object Code	No. of Cycles	No. of Bytes
NOP	12	2	1

NOP is a one-byte instruction that does nothing except increment the program counter.

Typical uses of NOP are the following:

1. To provide a position for a label without affecting the object program.
2. To produce a precise delay time. Each NOP instruction adds two clock cycles to the execution time of a sequence.
3. To replace instructions that are no longer needed because of corrections or changes.
4. To replace instructions (such as subroutine calls) that you may not want to include in debugging runs.

NOP is seldom used in completed programs, but it is often quite handy in the debugging and testing stages.

OR — Logical (Inclusive) OR

ORA

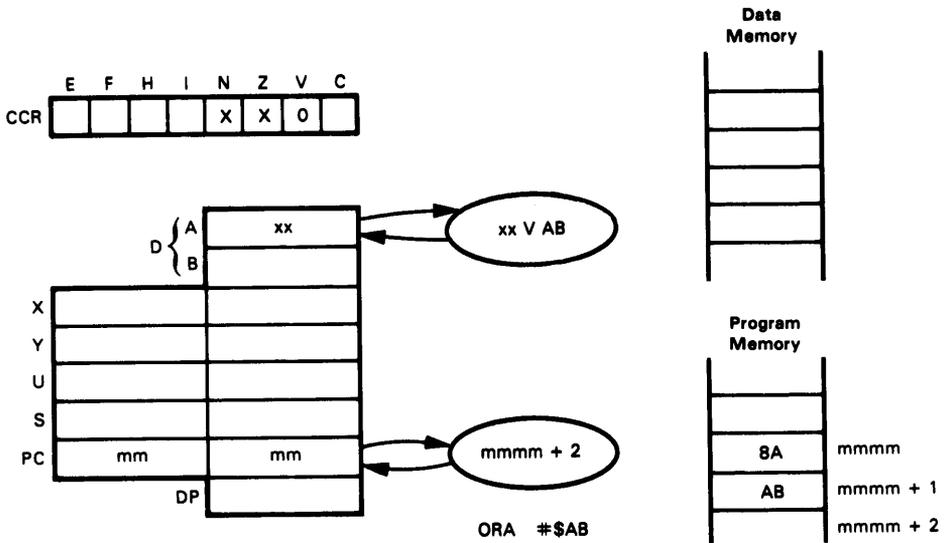
ORB

ORCC

	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ORA	8A	2	2	9A	4	2	BA	5	3	AA	4+	2+
ORB	CA	2	2	DA	4	2	FA	5	3	EA	4+	2
ORCC	1A	3	2									

This instruction logically (inclusive) ORs the contents of a memory location with the contents of Accumulator A or B or the Condition Code Register. Only immediate addressing can be used with CCR.

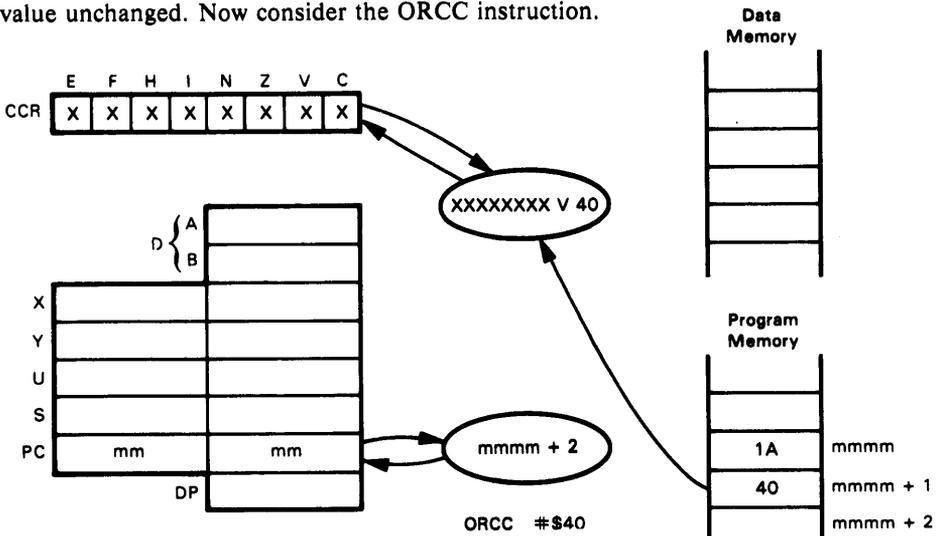
First consider the accumulator OR using immediate addressing and Accumulator A.



Suppose that $xx = E3_{16}$. After the processor has executed the instruction ORA #\$AB, Accumulator A will contain EB_{16} .

$E3 = 1110\ 0011$
 $AB = 1010\ 1011$
 $1110\ 1011 \rightarrow$ Nonzero result resets Z to 0
 $\quad \quad \quad \rightarrow$ Bit 7 sets N to 1
 V is always cleared

OR is a common logical instruction, most often used to set a bit to a '1' value. For example, the instruction ORA #%10000000 sets bit 7 of Accumulator A to 1 while leaving the other bits of the accumulator unchanged. Note that logically ORing a bit position with a '1' produces a result of '1', while logically ORing a bit position with '0' leaves the value unchanged. Now consider the ORCC instruction.



This instruction logically ORs the contents of the following byte of program memory with the contents of the Condition Code Register. This has the effect of setting all the flags that are logically ORed with '1's and leaving unchanged all the flags that are logically ORed with '0's. The following patterns will set individual flags:

Flag	Required Mask	
	Binary	Hexadecimal
E	10000000	80
F	01000000	40
H	00100000	20
I	00010000	10
N	00001000	08
Z	00000100	04
V	00000010	02
C	00000001	01

Of course, setting more than one bit position will set more than one flag at a time. However, only a few of the possible operations are really useful. In particular, we should note:

ORCC	##%01000000	DISABLE FAST INTERRUPTS
ORCC	##%00010000	DISABLE REGULAR INTERRUPTS
ORCC	##%00000010	SET OVERFLOW
ORCC	##%00000001	SET CARRY

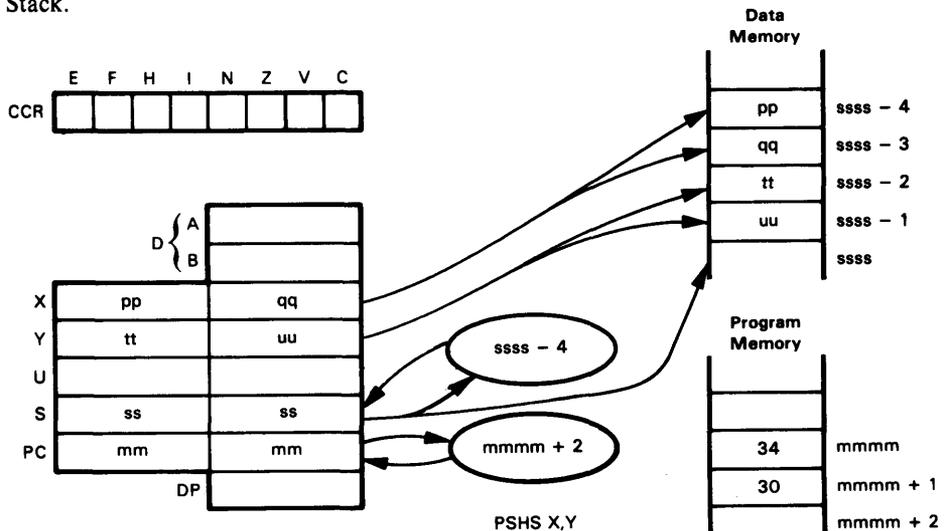
Remember that setting an interrupt mask disables the interrupt from that source.

PSH — Push Registers onto the Stack

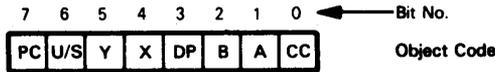
PSHU
PSHS

	Object Code	No. of Cycles	No. of Bytes
PSHS	34	5+	2+
PSHU	36	5+	2+

The PSH instruction will push onto either stack any or all registers except the Stack Pointer being used. Consider pushing both index registers onto the Hardware Stack.



The second byte of the instruction specifies the registers to be saved as follows:

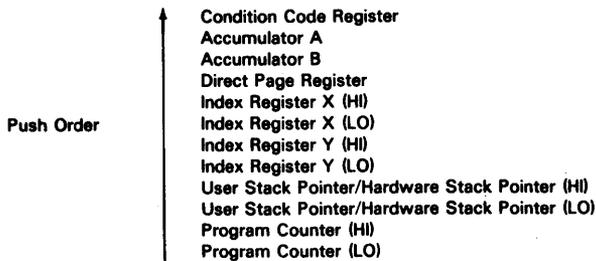


Each bit position that contains 1 causes the corresponding register to be saved on the stack. If the register is 16 bits in length, the processor saves its contents as follows:

1. Decrement the Stack Pointer and store the low-order byte of the register at the address in the Stack Pointer.
2. Decrement the stack pointer again and store the high-order byte of the register at the address in the Stack Pointer.

If the register is 8 bits long, the processor performs only one storage operation and decrements the stack pointer only once.

The order is as follows (with the lowest memory address at the top):



Note that some (or even all) of the registers can be omitted. The processor decrements the Stack Pointer once for each byte that it saves. The internal hardware determines the order in which registers are stacked; the order in which the programmer specifies registers does not matter. For example, the instructions PSHS A,B,X,DP and PSHS DP,A,B,X are identical. You may specify the double accumulator D instead of A and B.

Note that the Hardware Stack Pointer (S) cannot be pushed onto the Hardware Stack, nor can the User Stack Pointer (U) be pushed onto the User Stack. Thus the stack in use determines the meaning of bit 6 of the post byte. Setting bit 6 for PSHS will cause the User Stack Pointer to be pushed onto the Hardware Stack, while setting bit 6 for PSHU will cause the Hardware Stack Pointer to be pushed onto the User Stack.

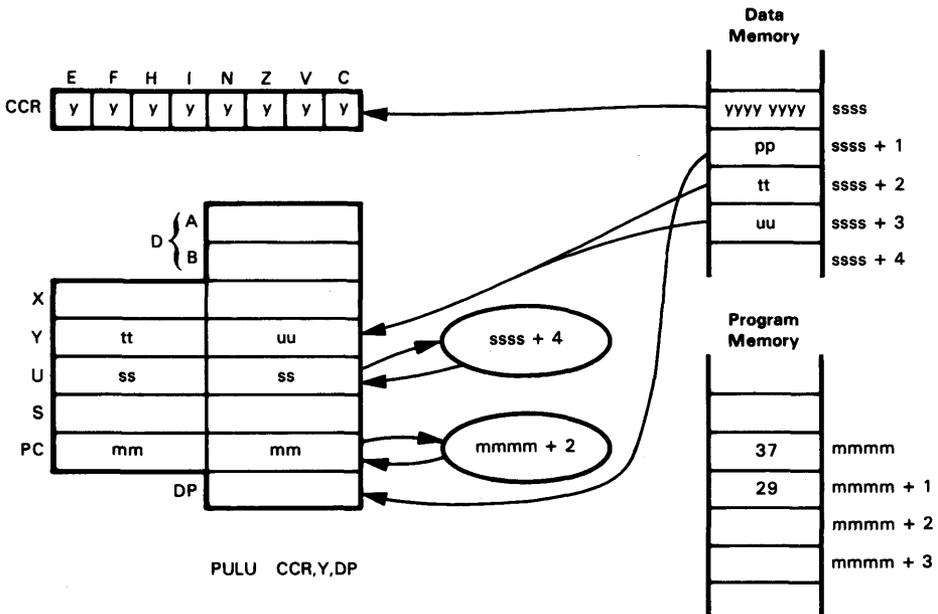
PUL — Pull Registers from the Stack

PULU

PULS

	Object Code	No. of Cycles	No. of Bytes
PULS	35	5+	2+
PULU	37	5+	2+

The PUL instruction will pull from either stack any or all registers except the designated Stack Pointer. Consider pulling the Condition Code Register (CCR), Index Register Y, and the Direct Page Register (DP) from the user stack.



The second byte of the instruction specifies the registers to be loaded exactly as for PSH. The order in which registers are pulled from the stack is the opposite of that in which they are pushed. As with PSH, you can omit any or all of the registers, you cannot

pull a stack pointer from its own stack, and the order in which registers are specified in the assembly language instruction has no effect on the order in which they are pulled from the stack.

Note that, unless you are loading the Condition Code Register itself, loading registers in this way does not affect the flags. If you wish to load a register from a stack and affect the flags, you should use the LD instruction in the autoincrement mode with the appropriate Stack Pointer. For example, to load Index Register Y from the user stack and set flags accordingly, use the instruction LDY ,U+ +.

The instruction PULU PC loads the Program Counter from the user stack and thus serves as a Return from Subroutine instruction in which the linkage is in the user stack, rather than the Hardware Stack. Pulling any set of registers that includes the Program Counter has a similar effect. The programmer can transfer control to and from subroutines through the user stack; a sequence like

```
PSHU PC
JMP SUBR
```

transfers control to subroutine SUBR after saving the Program Counter in the user stack. Note, however, that the subroutine will have to increment the return address past the JMP instruction.

ROL — Rotate Accumulator or Memory Byte Left through Carry

ROLA

ROLB

ROL

	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
ROL	49	2	1	09	6	2	79	7	3	69	6+	2+
ROLA	59	2	1									
ROLB												

This instruction rotates the specified accumulator or the selected byte of memory one bit to the left through the Carry flag.

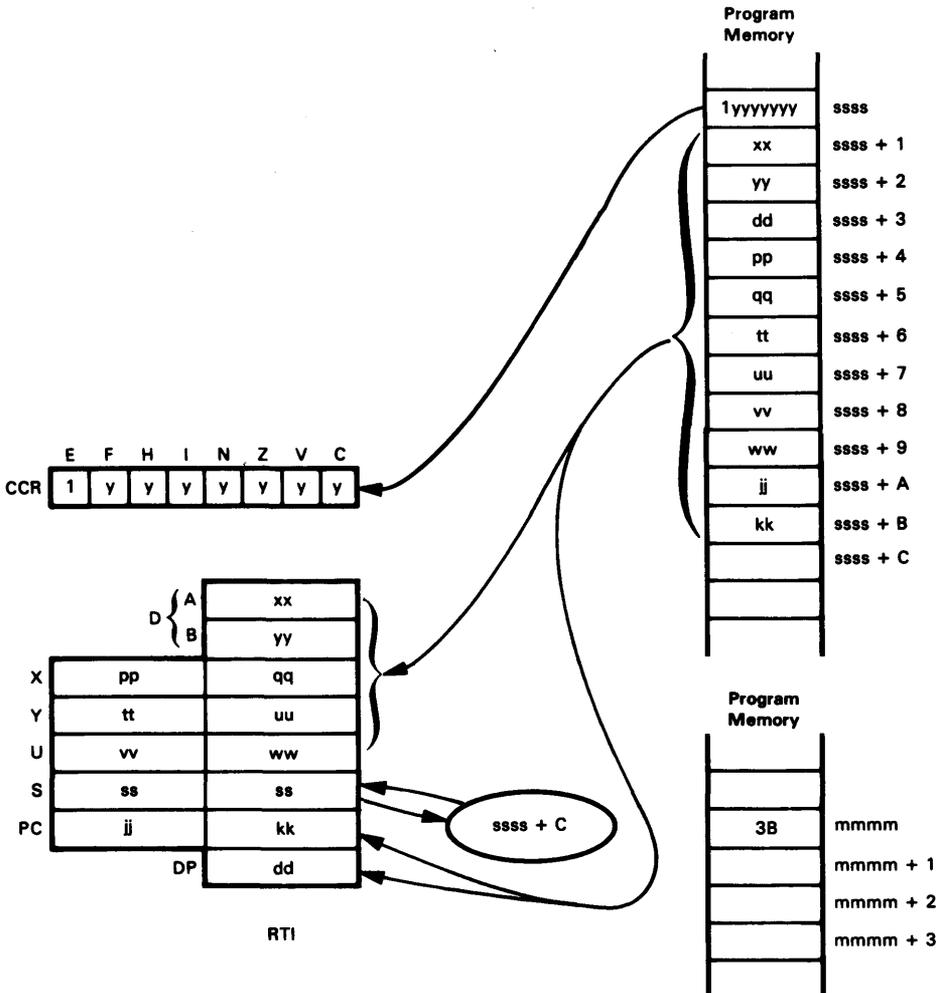
Consider rotating a memory byte using indexed addressing with zero offset from Index Register Y.

This instruction is the same as ROL except that the rotation is from left to right. The flags are affected in the same way except that C is now loaded with bit 0 and V is unaffected. Consult the description of ROL for more details.

RTI — Return from Interrupt

	Object Code	No. of Cycles	No. of Bytes
RTI	3B	6/15	1

This instruction restores the state of an interrupted task by loading the Condition Code Register and Program Counter from the hardware stack. If the Entire flag (E) is set, the instruction loads all the other user registers from the hardware stack.



Suppose that $yyy\ yyyy = 110\ 1101_2$, $xx = CB_{16}$, $yy = 14_{16}$, $dd = 2E_{16}$, $ppqq = 37A1_{16}$, $ttuu = E50B_{16}$, $vvww = 027F_{16}$, and $jjkk = E115_{16}$.

After the processor executes the RTS instruction, the registers will appear as follows:

```

CCR = 1110 11012
A = CB
B = 14
DP = 2E
X = 37A1
Y = E50B
U = 027F
PC = E115      Execution will continue from this address.
    
```

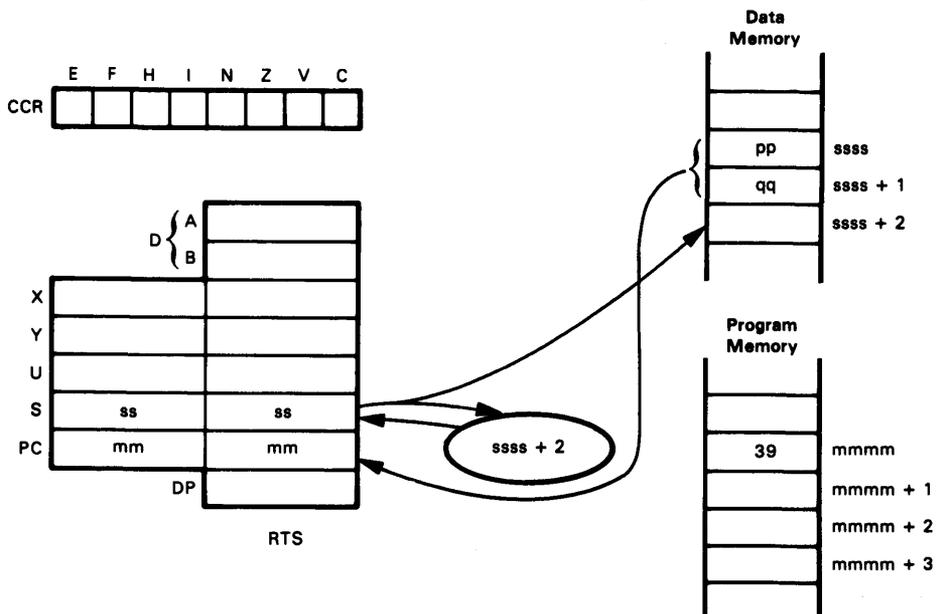
First, the condition code is pulled from the stack, and the contents of bit 7 (E) are examined by the hardware of the CPU to determine whether the entire machine status has been stacked, or just the subset CCR and PC. The order of the registers in the stack is the same as in the instructions PSHS and PULS. Note that, as in those instructions, the Hardware Stack Pointer is not saved in its own stack.

The interrupt masks will be automatically restored to their original states. The previous values of all the user registers are lost. The Hardware Stack Pointer ends with a value 12 (C_{16}) larger than its starting value when E is set and a value 3 larger when E is cleared (by a Fast Interrupt request).

RTS — Return from Subroutine

	Object Code	No. of Cycles	No. of Bytes
RTS	39	5	1

Program control is returned from the subroutine to the calling program by pulling the return address from the stack and placing it in the Program Counter.



The previous contents of the program counter are lost. The processor increments the Hardware Stack Pointer after loading each byte, so the final value of that pointer is two larger than its starting value.

Each subroutine normally contains at least one RTS instruction; this is the last instruction executed within the subroutine and causes control to return to the calling program. RTS does not affect any flags. Note, however, that no RTS instruction is necessary if the last instruction in the subroutine restores a set of registers including the program counter from the hardware stack. For example, the instruction PULS A,B,CC,PC will restore Accumulators A and B and the Condition Code Register from the hardware stack before returning control to the main program. Of course, the subroutine must include an appropriate PSHS instruction, such as PSHS A,B,CC.

SBA — Subtract Accumulator B from Accumulator A

The 6809 assembler translates this 6800 instruction into

```
PSHS  B
SUBA  ,S+
```

This instruction subtracts Accumulator B from Accumulator A and sets the condition flags accordingly. The 6809 assembler handles this instruction to allow source compatibility with the 6800 processor.

SBC — Subtract Memory from Accumulator with Borrow

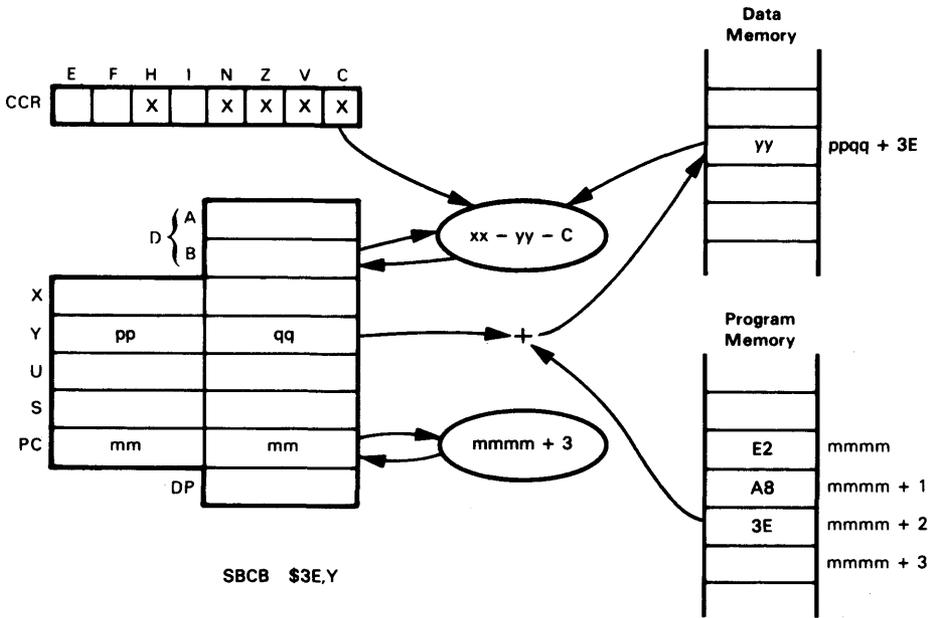
SBCA

SBCB

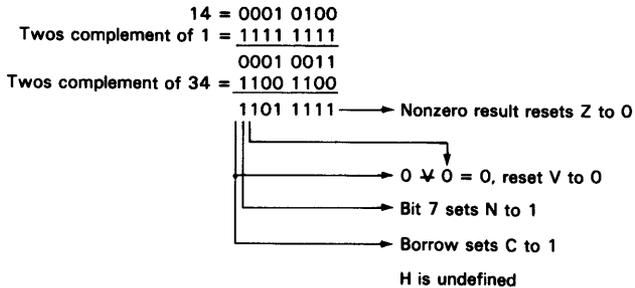
	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
SBCA	82	2	2	92	4	2	B2	5	3	A2	4+	2+
SBCB	C2	2	2	D2	4	2	F2	5	3	E2	4+	2+

This instruction subtracts the contents of the selected byte of memory and the contents of the carry flag from the contents of the specified accumulator.

Consider SBCB using an 8-bit constant offset from Index Register Y.



Suppose that $ppqq = 105A_{16}$, $xx = 14_{16}$, yy (contents of address 1098) = 34_{16} , and $C = 1$. After the processor executes the instruction `SBCB $3E,Y` the contents of Accumulator B will be DF_{16} .



Note that C is the complement of the resulting carry since it represents a borrow.

The most common use of SBC is in implementing multiple-precision arithmetic; this instruction allows borrows from previous byte-length operations to be included in the current byte-length operation.

SEC — Set Carry Flag

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction `ORCC#%00000001` — that is, into an instruction that sets to 1 the least significant bit of the condition code register (the carry flag). No other flags or registers are affected. The COM instruction also sets the carry flag.

SEF — Set Fast Interrupt Mask

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction ORCC #%01000000 — that is, into an instruction that sets to 1 bit 6 of the Condition Code Register (the fast interrupt mask). This instruction disables the fast interrupt — that is, the 6809 will not respond to the fast interrupt request control line. No other registers or flags are affected.

SEI — Set Interrupt Mask

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction ORCC #%00010000 — that is, into an instruction that sets to 1 bit 4 of the Condition Code Register (the regular interrupt mask bit). This instruction disables the 6809's regular interrupt — that is, the 6809 will not respond to the interrupt request control line. No other registers or flags are affected.

SEIF — Set Regular and Fast Interrupt Masks

The 6809 assembler translates this 6800-like instruction into the equivalent 6809 instruction ORCC #%01010000 — that is, into an instruction that sets to 1 bits 4 and 6 of the condition code register (the fast and regular interrupt mask bits). This instruction disables both of the 6809's maskable interrupts — that is, the 6809 will not respond to either the fast interrupt request control line or to the (regular) interrupt request control line. No other registers or flags are affected.

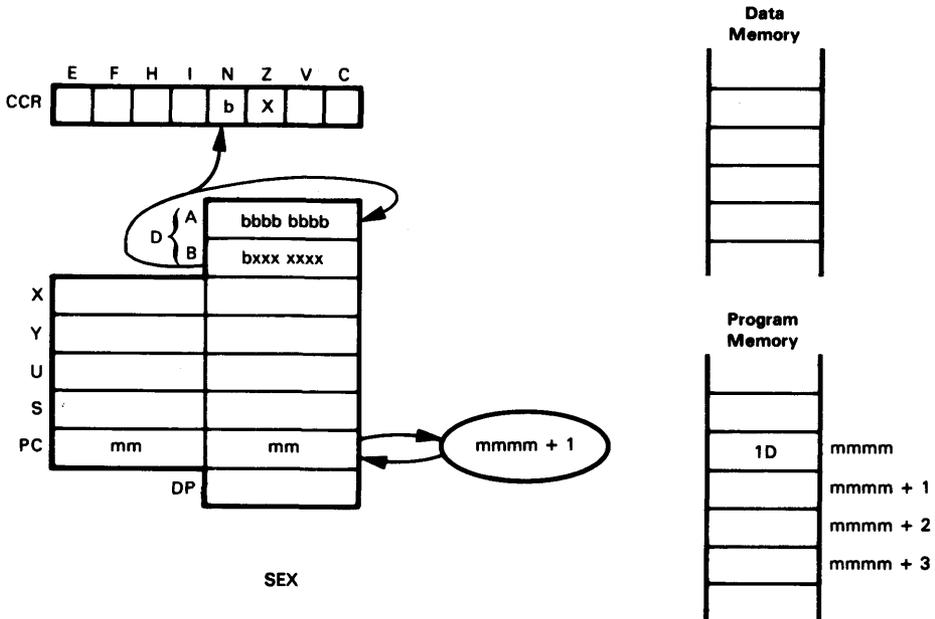
SEV — Set Overflow Flag

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction ORCC #%00000010 — that is, into an instruction that sets to 1 bit 1 of the Condition Code Register (the overflow flag). No other flags or registers are affected.

SEX — Sign Extend Accumulator B into Accumulator A

	Object Code	No. of Cycles	No. of Bytes
SEX	1D	2	1

This instruction transforms an 8-bit two's complement number in Accumulator B into a 16-bit two's complement number in Accumulator A.



SEX accomplishes this transformation by extending bit 7 of Accumulator B into Accumulator A. Thus Accumulator A is set to 00_{16} if bit 7 of Accumulator B is 0 and to FF_{16} if bit 7 of Accumulator B is 1. SEX affects the sign flag (set according to the most significant bit of the result — the same as bit 7 of Accumulator B) and the Zero flag (set if the result is zero — that is, if Accumulator B contains zero). This instruction is useful in performing twos complement and floating point arithmetic.

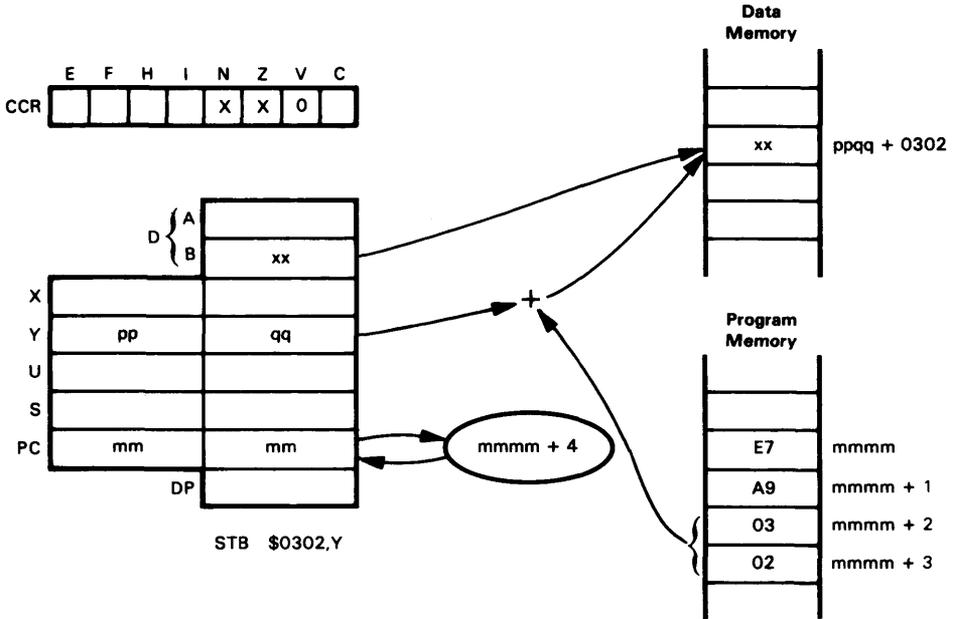
ST — Store Register into Memory

- STA
- STB
- STD
- STS
- STU
- STX
- STY

	Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
STA	97	4	2	B7	5	3	A7	4+	2+
STB	D7	4	2	F7	5	3	E7	4+	2+
STD	DD	5	2	FD	6	3	ED	5+	2+
STS	10 DF	6	3	10 FF	7	4	10 EF	6+	3+
STU	DF	5	2	FF	6	3	EF	5+	2+
STX	9F	5	2	BF	6	3	AF	5+	2+
STY	10 9F	6	3	10 BF	7	4	10 AF	6+	3+

Store the contents of the specified register at the selected memory address. There are two forms of this instruction, an 8-bit form and a 16-bit form. The 8-bit form is associated with Accumulators A and B, while the 16-bit form is associated with the 16-bit registers D, X, Y, S and U.

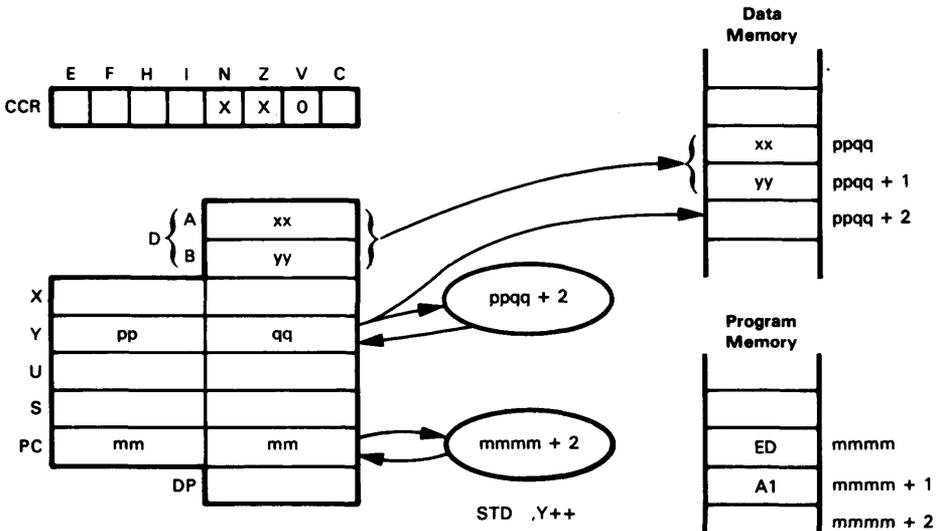
Consider the 8-bit case, storing Accumulator B using the indexed addressing mode with a constant 16-bit offset from Index Register Y.



Suppose that $xx = 63_{16}$ and $ppqq = 0238_{16}$. After the processor executes the instruction STB \$0302,Y memory location 053A will contain 63_{16} .

63 = 0110 0011 → Nonzero result resets Z to 0
 Bit 7 resets N to 0
 V is cleared always.

Now consider the 16-bit case, storing the D register using indexed addressing, autoincrementing Index Register Y by 2.



Suppose that $ppqq = 1430_{16}$, $xx = C1_{16}$, and $yy = 9A_{16}$. After the processor has executed the instruction $STD, Y++$ memory location 1430 will contain $C1_{16}$, memory location 1431 will contain $9A_{16}$, and Index Register Y will contain 1432_{16} .

$C19A = 1100\ 0001\ 1001\ 1010$ — Nonzero result resets Z to 0.
 — Bit 15 sets N to 1
 V is always cleared

Note that the STS and STY instructions have a two byte object code, the first byte being 10_{16} .

SUB — Subtract Memory from Register

SUBA

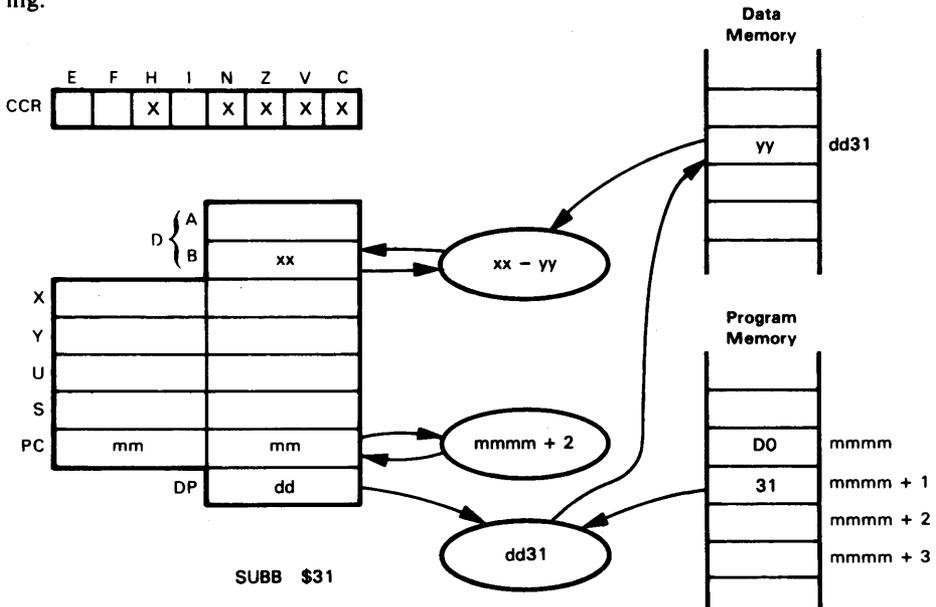
SUBB

SUBD

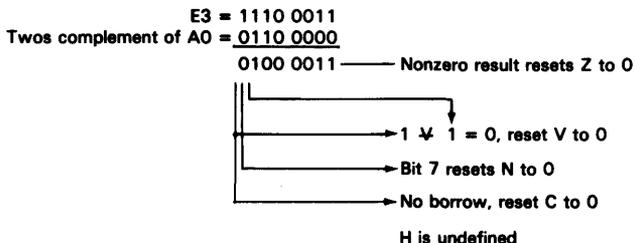
	Immediate			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
SUBA	80	2	2	90	4	2	B0	5	3	A0	4+	2+
SUBB	C0	2	2	D0	4	2	F0	5	3	E0	4+	2+
SUBD	83	4	3	93	6	2	B3	7	3	A3	6+	2+

Subtract the contents of the selected byte of memory from the contents of the specified accumulator. There are two forms of this instruction, a one-byte form and a two-byte form. The one-byte form uses an 8-bit accumulator (A or B), while the two-byte form uses the 16-bit Accumulator D.

First consider 8-bit subtraction using Accumulator B and base page direct addressing.

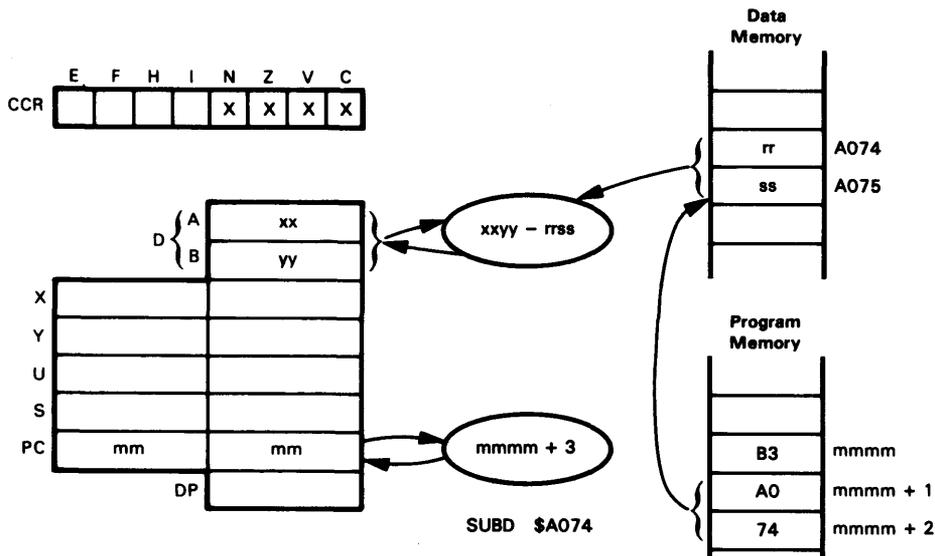


Suppose, for example, that $xx = E3_{16}$, $dd = 6B_{16}$, and yy (in memory location $6B31$) = $A0_{16}$. After the processor executes the instruction $SUBB \$31$, the contents of Accumulator B will be 43_{16} .

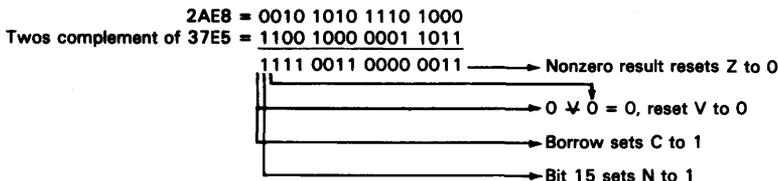


The **SUBA** and **SUBB** instructions are used to perform single-byte subtractions or to perform the subtraction of the low-order bytes in multibyte operations.

Now consider the 16-bit form **SUBD**. We will illustrate its execution using extended direct addressing.



For example, suppose that $xx = 2A_{16}$, $yy = E8_{16}$, $rr = 37_{16}$ and $ss = E5_{16}$. After the processor executes the instruction $SUBD \$A074$, the contents of Accumulator D will be $F303_{16}$ — that is, Accumulator A will contain $F3_{16}$ and Accumulator B will contain 03_{16} .



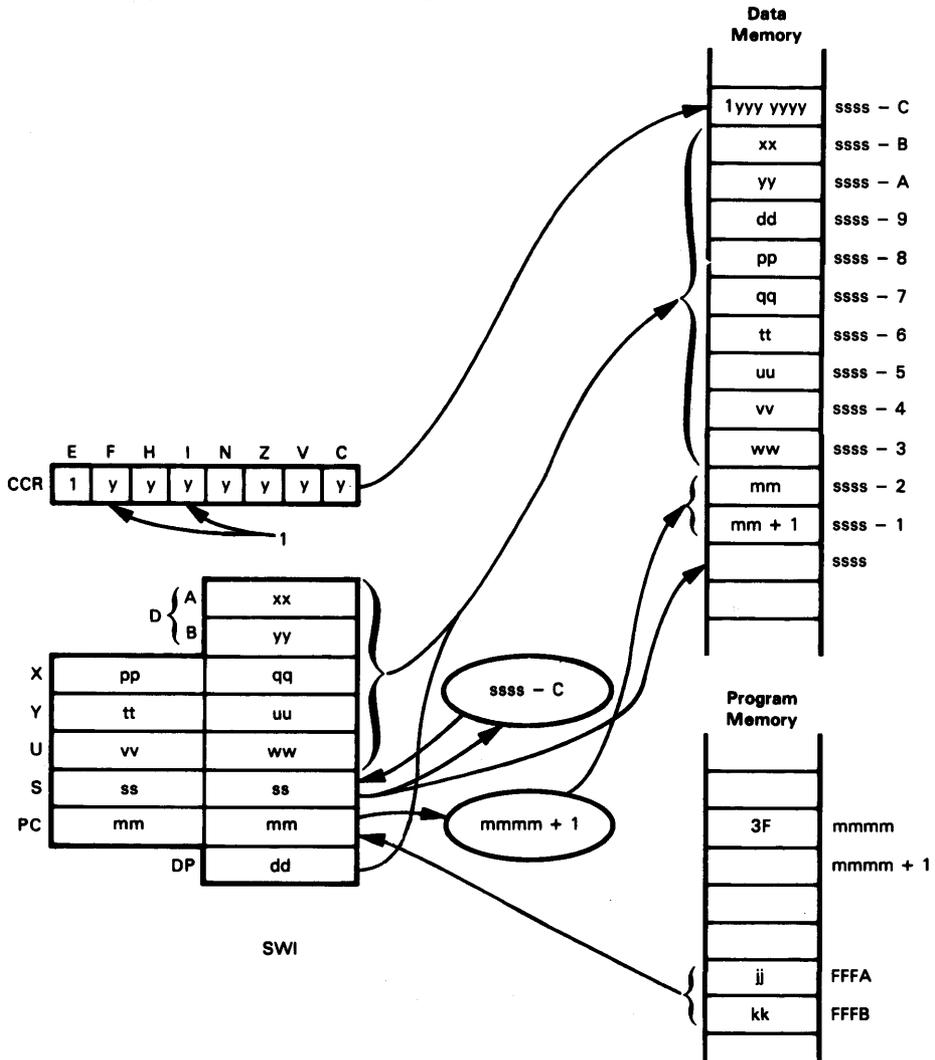
Note that the Half-carry flag (H) is not affected by **SUBD** and that C is the complement of the resulting carry, since it represents a borrow.

SWI – Software Interrupt

SWI
SWI2
SWI3

	Object Code	No. of Cycles	No. of Bytes
SWI	3F	19	1
SWI2	10 3F	20	2
SWI3	11 3F	20	2

This instruction increments the program counter, sets the E flag (bit 7 of CCR) – indicating that the entire state of the processor has been saved, and stores all the user registers except the Hardware Stack Pointer in the hardware Stack. Control is then passed through a vector table at the high end of memory.



Note that control is passed to a service routine by placing into the program counter the address located at FFFA and FFFB, and that the interrupt and fast interrupt mask bits (bits 4 and 6 of the CCR) are set, disabling the maskable interrupts.

The processor stores the user registers in the same order as the PSHS instruction. The final contents of the Hardware Stack Pointer are the original contents minus 12 (C₁₆). The E flag is set to 1 so that an RTI instruction will restore the original state, except that the Program Counter will have been incremented by 1. Thus an RTI will cause the resumption of execution of the suspended program at the instruction immediately following SWI. SWI disables both the regular interrupt and the fast interrupt by setting the I and F bits of the CCR but only after the CCR has been pushed onto the stack. Therefore interrupts are not honored during the SWI service routine, but interrupt status is restored by execution of an RTI.

Software interrupt instructions SWI2 and SWI3 are similar to SWI. The vector for SWI2 is at FFF4 and FFF5, while the vector for SWI3 is at FFF2 and FFF3. SWI2 and SWI3 are intended for the user, rather than the system software. Motorola guarantees never to use SWI2 in any of its packaged software. SWI2 and SWI3 do not set the I and F bits as does SWI, and thus interrupt status is maintained. SWI2 and SWI3 require two byte operation codes.

The SWI instruction can be used for a variety of functions. The entry point for any software package — a debug monitor, a disk operating system, or a group of system subroutines — can be inserted into a software interrupt pointer. The software system can then be entered by execution of a SWI instruction.

SYNC — Synchronize to External Event (Wait for Interrupt)

	Object Code	No. of Cycles	No. of Bytes
SYNC	13	2	1

This instruction simply halts CPU execution until a peripheral device requests an interrupt. Any interrupt clears the halt. If the interrupt is enabled and lasts 3 cycles or more, the processor will respond to it, stacking the registers and transferring control to the appropriate vector address. If the interrupt is masked (disabled) or is shorter than 3 cycles long, the processor simply continues to the next instruction without stacking registers or transferring control to a service routine. SYNC differs from CWAI as follows:

1. SYNC does not provide a means for enabling interrupts during instruction execution.
2. The processor tristates its busses while executing SYNC, but not while executing CWAI.
3. SYNC provides a continuation exit without an interrupt response, whereas CWAI requires an interrupt response.

SYNC is normally used with interrupts disabled as a HALT instruction which is cleared by any interrupt input. CWAI is normally used with the appropriate interrupt enabled as a Wait for Interrupt instruction. Figure 22-1 is a flowchart of the logic of the SYNC instruction.

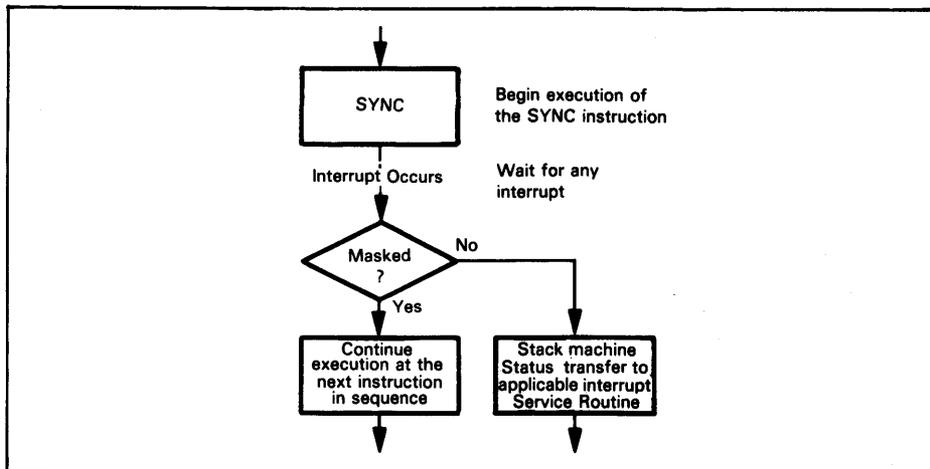


Figure 22-1. MC6809 SYNC Logic

TAB – Transfer Accumulator A to Accumulator B

The 6809 assembler translates this 6800 instruction into

TFR A, B
TSTA

This instruction transfers the contents of Accumulator B to Accumulator A and sets the flags accordingly. The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor.

TAP – Transfer Accumulator A to Condition Code Register

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction TFR A,CC – that is, into an instruction that transfers the contents of Accumulator A to the Condition Code Register (CCR). The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor.

TBA – Transfer Accumulator B to Accumulator A

The 6809 assembler translates this 6800 instruction into

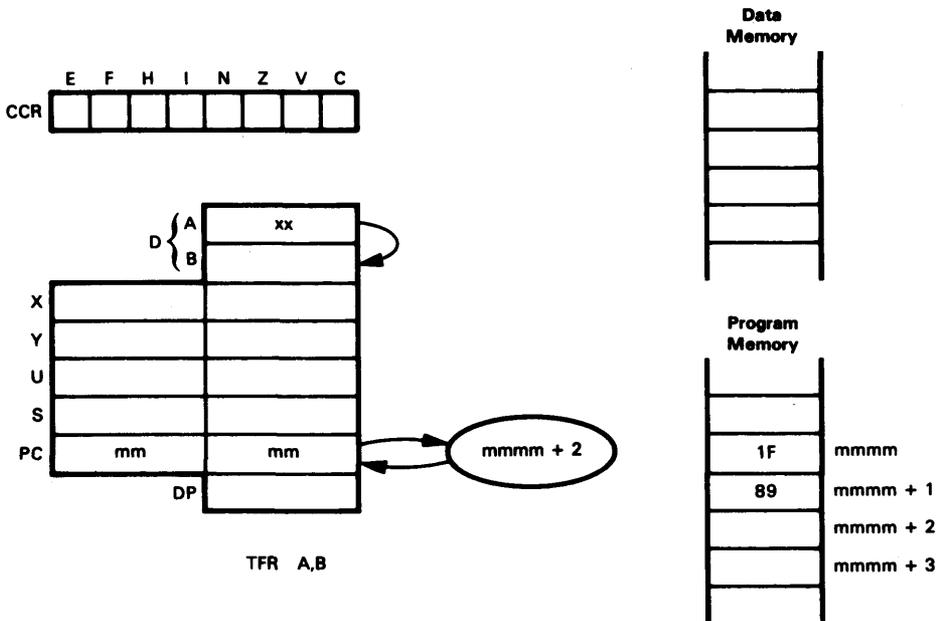
TFR B, A
TSTA

This instruction is used to transfer data from one register to another. Data may only be transferred between registers of like size. In contrast to the EXG instruction, this is a one-way transfer. Consider the transfer from Accumulator A to B.

TFR – Transfer Register to Register

	Object Code	No. of Cycles	No. of Bytes
TFR	1F	7	2

This instruction transfers the contents of Accumulator A to Accumulator B and sets the flags accordingly. The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor.



Suppose $xx = 6A_{16}$ and the original contents of Accumulator B are 57_{16} . At the end of the execution of TFR A,B both accumulators will contain the number $6A_{16}$.

The TFR instruction has many miscellaneous applications:

1. **Moving the contents of one accumulator to another – TFR A,B or TFR B,A.** Remember that only Accumulator A can be operated on with the Decimal Adjust instruction.
2. **Loading the direct page register – TFR A,DP.** This instruction loads the direct page register from Accumulator A. There is no LD instruction for the direct page register.
3. **Transferring control to an address contained in an Index Register – TFR X,PC.** This instruction loads the program counter with the contents of Index Register X; the next instruction to be executed will be taken from that address.

Note that TFR destroys the old contents of the destination register. You can save those contents in the source register by using EXG instead. TFR, however, does not change the contents of the source register.

All TFR instructions require two bytes of program memory – the operation code and the post byte (the immediate data), which specifies the source and destination registers. Be careful of the fact that some TFR instructions are meaningless (undefined register codes), while others are illegal (transferring contents between registers of different sizes).

The post byte for the TFR instruction is identical to the post byte illustrated in the EXG instruction description. The TFR post byte's higher order four bits define the source register while the lower order four bits define the destination register. The flags are unaffected unless CCR is the destination register.

TPA – Transfer Condition Code Register to Accumulator A

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction TFR CC,A – that is, into an instruction that transfers the contents of the Condition Code Register (CCR) to Accumulator A. The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor.

TST – Test the Contents of an Accumulator or Memory Byte

TSTA

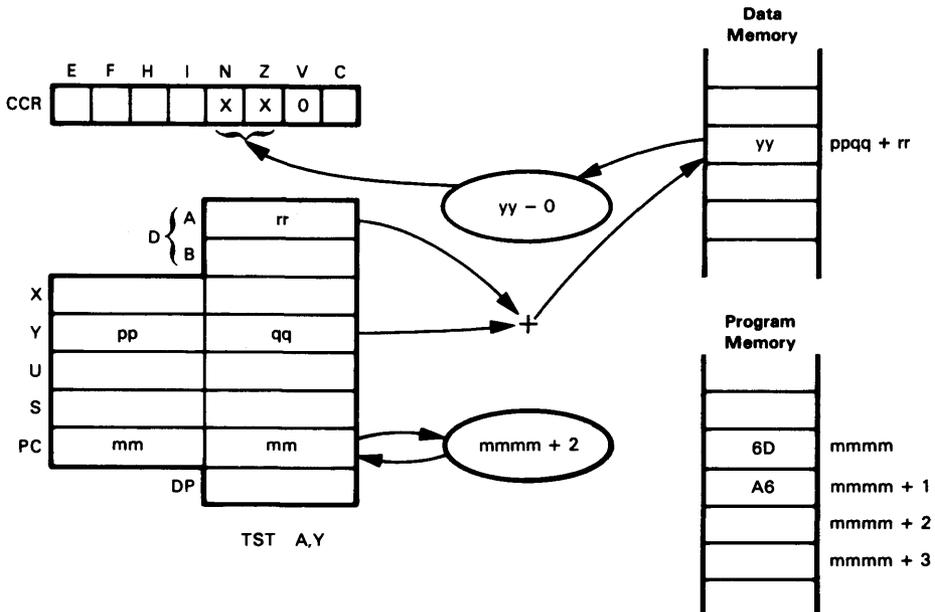
TSTB

TST

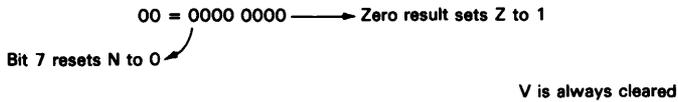
	Inherent			Direct			Extended			Indexed/Indirect		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
TST				0D	6	2	7D	7	3	6D	6+	2+
TSTA	4D	2	1									
TSTB	5D	2	1									

This instruction sets the Sign and Zero flags according to the contents of the specified accumulator or the selected byte of memory.

Consider testing a byte of memory addressed by Index Register Y with Accumulator A offset.



Suppose that Index Register Y contains 0100_{16} , Accumulator A contains 02_{16} , and the contents of memory location 0102 are 00_{16} . After the processor executes the instruction TST A,Y the sign and overflow flags will contain zero and the zero flag will contain one. No registers or memory locations will be changed.



The TST instruction lets the programmer set the flags according to the contents of an accumulator or a byte of memory without performing any operations or changing any registers.

TSX — Transfer Stack Pointer S to Index Register X

The 6809 assembler translates this 6800 instruction into the equivalent 6809 instruction TFR S,X — that is, into an instruction that transfers the contents of the Hardware Stack Pointer S to Index Register X. The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor. There is a slight difference between the 6800 and 6809 stack pointers: the 6809 stack pointer points to the last occupied byte of the stack, while the 6800 pointer indicates the next empty byte. That is, the 6800 stack pointer value is one less than the 6809's for the same stack condition. Therefore, although TFR S,X does not increment the value before loading the index register, as 6800 TSX does, the result is still a correct transfer of pointers.

TXS — Transfer Index Register X to Stack Pointer S

The 6809 assembler will translate this 6800 instruction into the equivalent 6809 instruction TFR X,S — that is, into an instruction that transfers the contents of Index Register X to the Hardware Stack Pointer, S. The instruction is handled by the 6809 assembler to allow source compatibility with the 6800 processor. There is a slight difference between the 6800 and 6809 stack pointers: the 6809 stack pointer points to the last occupied byte of the stack, while the 6800 pointer indicates the next empty byte. That is, the 6800 stack pointer value is one less than the 6809's for the same stack condition. Therefore, although TFR X,S does not decrement the value before loading the Stack Pointer, as 6800 TXS does, the result is still a correct transfer of pointers.

WAI — Wait for Interrupt

The 6809 assembler translates this 6800 instruction into the 6809 instruction CWAI #\$FF. This instruction is handled to allow source compatibility with the 6800 processor.

Appendices

The following section presents a complete set of reference tables for the 6809 instruction set.

Appendix A summarizes 6809 instruction operations and effects, and is organized by function to display the capabilities of the 6809 processor. Appendix B summarizes the indexed and indirect addressing modes: which modes are available, their assembly language forms, and the resulting object code post bytes. For each instruction mnemonic Appendix C shows the available addressing modes, its object code, execution time in machine cycles, and the number of bytes occupied by the instruction. Appendices B and C can serve as aids to hand assembly of 6809 instructions. Appendix D lists all the valid object codes and their instruction mnemonics, and Appendix E lists all the indexed and indirect addressing post bytes and the assembler forms which generate them. These two tables can be used for hand checking and disassembly of object code, tasks sometimes required in the debugging of an assembly language program.

A

Summary of the 6809 Instruction Set

Appendix A uses the following symbols:

The registers:

A, B	Accumulators
D	Double Accumulator (A and B concatenated, A high-order)
DP	Direct Page Register
X, Y	Index registers
PC	Program Counter
S	Hardware Stack Pointer
U	User Stack Pointer
CC	Status (Condition Code) Register

The flags (statuses), starting with bit 0 of the condition code register and proceeding to bit 7:

C	Carry (Borrow) flag
V	Overflow flag
Z	Zero flag
N	Sign (Negative) flag
I	(Regular) Interrupt Mask bit
H	Half-Carry flag
F	Fast Interrupt Mask bit
E	Entire flag

Symbols in the Status (flags) columns:

(blank)	Operation does not affect flag
X	Operation affects flag
0	Operation clears flag
1	Operation sets flag

Other symbols and abbreviations:

ACx	An accumulator, either Accumulator A or Accumulator B
adr8	An 8-bit address, a 1-byte quantity which may be used to directly address memory locations on the base (direct) page
adr16	A 16-bit memory address
b0-b7	Bits of a Post Byte or an 8-bit register
C	Contents of the Carry flag, either 0 or 1
data8	An 8-bit unit of binary data
data16	A 16-bit unit of binary data
disp8	An 8-bit signed binary address displacement
disp16	A 16-bit signed binary address displacement
EA	Effective address calculated by any addressing method
M	Memory address as determined by base page direct, extended direct, indexed, or indirect addressing
[M]	Contents of M
[M] : [M + 1]	16-bit data item; its high-order byte is the contents of M, and its low-order byte is the contents of the next higher address.
reg	A 16-bit index register or stack pointer (S, U, X, or Y)
reg. list	A list of registers to be stored on or retrieved from a stack
R16	A 16-bit register (D, S, U, X, or Y)
R1, R2	Two registers, both 8-bit or both 16-bit
SP	A stack pointer (either S or U)
ind. forms	Any of the indexed or indirect addressing methods described in Appendix B
[Interrupt vector]	The address contained in one of the interrupt vectors (see Table 15-1)
xx(HI)	The high-order 8 bits of the 16-bit quantity xx
xx(LO)	The low-order 8 bits of the 16-bit quantity xx
[]	Contents of location enclosed by brackets
[[]]	Implied memory address: the contents of the memory location designated by the contents of a register
∧	Logical AND
∨	Logical (Inclusive) OR
⊕	Logical Exclusive-OR
→	Data is transferred in the direction of the arrow
↔	Data is transferred in both directions simultaneously, thus exchanging the contents of the source and the destination.

Appendix A. A Summary of the 6809 Instruction Set

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status								Operation Performed
					E	F	H	I	N	Z	V	C	
Primary Memory Reference and I/O	LDA { LDB }	adr8 adr16 ind. forms	2 3 2+	4 5 4+				X	X	X	0		The 6809 permits the following addressing modes for all Primary Memory Reference instructions: base page direct, extended direct, indexed, and indirect. [ACx] ← [M] Load Accumulator A or B from specified memory location. [M] ← [ACx] Store contents of Accumulator A or B in specified memory location. [D] ← [M]:[M + 1] Load double Accumulator from specified memory location. Sign flag (N) takes the value of bit 15 of the data (bit 7 of Accumulator A). [M]:[M + 1] ← [D] Store contents of double Accumulator in specified memory location. Sign flag takes the value of bit 15 of the data (bit 7 of Accumulator A). [reg] ← [M]:[M + 1] Load specified register (X, Y, U, or S) from memory. Sign flag (N) takes the value of bit 15 of the data. [M]:[M + 1] ← [reg] Store contents of specified register (X, Y, U, or S) in memory. Sign flag (N) takes the value of bit 15 of the register.
	STA { STB }	adr8 adr16 ind. forms	2 3 2+	4 5 4+				X	X	X	0		
	LDD	adr8 adr16 ind. forms	2 3 2+	5 6 5+				X	X	X	0		
	STD	adr8 adr16 ind. forms	2 3 2+	5 6 5+				X	X	X	0		
	LDX { LDU }	adr8 adr16 ind. forms	2 3 2+	5 6 5+				X	X	X	0		
	LDY { LDS }	adr8 adr16 ind. forms	3 4 3+	6 7 6+				X	X	X	0		
	STX { STU }	adr8 adr16 ind. forms	2 3 2+	5 6 5+				X	X	X	0		
	STY { STS }	adr8 adr16 ind. forms	3 4 3+	6 7 6+				X	X	X	0		

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed
					E	F	H	I	N	Z	V	C			
Secondary Memory Reference (Memory Operate)	ADCA } ADCB }	adr8	2	4				X	X	X	X	X	X	X	The 6809 permits the following addressing modes for all Secondary Memory Reference instructions: base page direct, extended direct, indexed, and indirect. [ACx] ← [ACx] + [M] + C Add with carry to Accumulator A or B.
		adr16	3	5				X	X	X	X	X	X	X	
		ind. forms	2+	4+				X	X	X	X	X	X	X	
	ADDA } ADDB }	adr8	2	4				X	X	X	X	X	X	X	[ACx] ← [ACx] + [M] Add contents of specified memory location to Accumulator A or B.
		adr16	3	5				X	X	X	X	X	X	X	
		ind. forms	2+	4+				X	X	X	X	X	X	X	
	ADDD	adr8	2	6				X	X	X	X	X	X	X	[D] ← [D] + [M]:[M + 1] Add 16-bit value from memory to double Accumulator. The operand's high-order byte is in the specified memory location; the low-order byte is in the next higher address.
		adr16	3	7				X	X	X	X	X	X	X	
		ind. forms	2+	6+				X	X	X	X	X	X	X	
	ANDA } ANDB }	adr8	2	4				X	X	X	X	0	0	0	[ACx] ← [ACx] ∧ [M] AND contents of specified memory location with Accumulator A or B.
		adr16	3	5				X	X	X	X	0	0	0	
		ind. forms	2+	4+				X	X	X	X	0	0	0	
	BITA } BITB }	adr8	2	4				X	X	X	X	X	X	X	[ACx] ∧ [M] AND contents of specified memory location with Accumulator A or B. Only the Status register is affected.
		adr16	3	5				X	X	X	X	X	X	X	
		ind. forms	2+	4+				X	X	X	X	X	X	X	
CMPA } CMPB }	adr8	2	4				X	X	X	X	X	X	X	[ACx] - [M] Compare contents of specified memory location with Accumulator A or B. Only the Status register is affected.	
	adr16	3	5				X	X	X	X	X	X	X		
	ind. forms	2+	4+				X	X	X	X	X	X	X		
CMPD	adr8	3	7				X	X	X	X	X	X	X	[D] - [M]:[M + 1] Compare 16-bit data with double Accumulator. Only the status register is affected. The high-order byte of the data is in the specified memory location; the low-order byte is in the next higher address.	
	adr16	4	8				X	X	X	X	X	X	X		
	ind. forms	3+	7+				X	X	X	X	X	X	X		
CMPS } CMPU } CMPY }	adr8	3	7				X	X	X	X	X	X	X	[reg] - [M]:[M + 1] Compare 16-bit data with specified register (S, U, X, or Y). Only the status register is affected. The high-order byte of the data is in the specified memory location; the low-order byte is in the next higher address.	
	adr16	4	8				X	X	X	X	X	X	X		
	ind. forms	3+	7+				X	X	X	X	X	X	X		

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status								Operation Performed
					E	F	H	I	N	Z	V	C	
Secondary Memory Reference (Memory Operate) (Continued)	CMPX	adr8 adr16 ind. forms	2 3 2+	6 7 6+					X	X	X	X	Same as CMPS/CMPU/CMPY. See page A-4.
	EORA EORB	adr8 adr16 ind. forms	2 3 2+	4 5 4+				X	X	0	0	0	[ACx] ← [ACx] ∨ [M] Logical Exclusive-OR contents of specified memory location with Accumulator A or B.
	ORA ORB	adr8 adr16 ind. forms	2 3 2+	4 5 4+				X	X	0	0	0	[ACx] ← [ACx] V [M] Logical (Inclusive) OR contents of specified memory location with Accumulator A or B.
	SBCA SBCB	adr8 adr16 ind. forms	2 3 2+	4 5 4+		X		X	X	X	X	X	[ACx] ← [ACx] - [M] - C Subtract contents of specified memory location and contents of Carry flag from Accumulator A or B.
	SUBA SUBB	adr8 adr16 ind. forms	2 3 2+	4 5 4+		X	X	X	X	X	X	X	[ACx] ← [ACx] - [M] Subtract contents of specified memory location from Accumulator A or B.
	SUBD	adr8 adr16 ind. forms	2 3 2+	6 7 6+			X	X	X	X	X	X	[D] ← [D] - [M]; [M] + 1 Subtract 16-bit value in memory from double Accumulator. The operand's high-order byte is in the specified memory address; the low-order byte is in the next higher address.
	ASL	adr8 adr16 ind. forms	2 3 2+	6 7 6+		X			X	X	X	X	 Arithmetic shift left. Bit 0 is set to 0.
	ASR	adr8 adr16 ind. forms	2 3 2+	6 7 6+		X	X		X	X	X	X	 Arithmetic shift right. Bit 7 stays the same.

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed
					E	F	H	I	N	Z	V	C			
Secondary Memory Reference (Memory Operate) (Continued)	CLR	adr8 adr16 ind. forms	2 3 6+	6 7 6+					0	1	0	0	0	$[M] \leftarrow 00_{16}$ Clear specified memory location.	
	COM	adr8 adr16 ind. forms	2 3 2+	6 7 6+				X	X	X	0	1	$[M] \leftarrow [M]$ Ones complement contents of memory location.		
	DEC	adr8 adr16 ind. forms	2 3 2+	6 7 6+				X	X	X	X	X	$[M] \leftarrow [M]-1$ Decrement (by 1) contents of memory location.		
	INC	adr8 adr16 ind. forms	2 3 2+	6 7 6+				X	X	X	X	X	$[M] \leftarrow [M]+1$ Increment (by 1) contents of memory location.		
	LSL	adr8 adr16 ind. forms	2 3 2+	6 7 6+		X		X	X	X	X	X	 Logical shift left. Same as ASL.		
	LSR	adr8 adr16 ind. forms	2 3 2+	6 7 6+			0	X	X	X	X	X	 Logical shift right. Bit 7 is set to 0.		
	NEG	adr8 adr16 ind. forms	2 3 2+	6 7 6+		X		X	X	X	X	X	$[M] \leftarrow 00_{16} - [M]$ Two's complement (negate) contents of memory location. Set Carry if result is 00_{16} and clear Carry otherwise. Set Overflow if result is 80_{16} and clear Overflow otherwise.		
	ROL	adr8 adr16 ind. forms	2 3 2+	6 7 6+				X	X	X	X	X	 Rotate contents of memory location left through Carry flag.		

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed
					E	F	H	I	N	Z	V	C			
Secondary Memory Reference	ROR	adr8 adr16 ind. forms	2 3 2+	6 7 6+					X	X				X	 <p>Rotate contents of memory location right through Carry flag.</p> <p>[M]</p>
	TST	adr8 adr16 ind. forms	2 3 2+	6 7 6+				X	X		X	0			<p>[M] — 0016 Test contents of memory location for zero or negative value.</p>
Immediate	LDA } LDB }	data8	2	2				X	X			0			<p>[ACx] — data8 Load Accumulator A or B immediate.</p>
	LDD	data16	3	3				X	X			0			<p>[D] — data16 Load double Accumulator immediate. Sign flag reflects bit 15 of the data (bit 7 of Accumulator A).</p>
	LDU } LDX }	data16	3	3				X	X			0			<p>[reg] — data16 Load specified register (X,Y,U, or S) immediate. Sign flag (N) reflects bit 15 of the register.</p>
	LDS } LDY }	data16	4	4				X	X			0			
Immediate Operate	ADCA } ADCB }	data8	2	2		X		X	X			X	X	X	<p>[ACx] — [ACx] + data8 + C Add immediate with carry to Accumulator A or B.</p>
	ADDA } ADDB }	data8	2	2		X		X	X			X	X	X	<p>[ACx] — [ACx] + data8 Add immediate to Accumulator A or B.</p>
	ADDD	data16	3	4				X	X			X	X	X	<p>[D] — [D] + data16 Add 16-bit data to double Accumulator. The high-order byte follows the operation code; the low-order byte follows the high-order byte.</p>

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed
					E	F	H	I	N	Z	V	C			
(Immediate Operate (Continued))	ANDA ANDB	data8	2	2				X	X	0					[ACx] ← [ACx] ∧ data8 Logical AND immediate with Accumulator A or B.
	BITA BITB	data8	2	2				X	X	0					[ACx] ∧ data8 Logical AND immediate with Accumulator A or B but affect only the status register.
	CMPA CMPB	data8	2	2			X	X	X	X					[ACx] - data8 Subtract immediate from Accumulator A or B but affect only the status register.
	CMPD	data16	4	5				X	X	X	X				[D] - data16 Subtract immediate from double Accumulator but affect only the status register.
	CMPS CMPU CMPY	data16	4	5				X	X	X	X				[reg] - data16 Subtract immediate from specified register (S, U, X, or Y), but affect only the status register.
	CMPX	data16	3	4				X	X	X	X				[ACx] ← [ACx] ∨ data8 Logical Exclusive-OR immediate with Accumulator A or B.
	EORA EORB	data8	2	2				X	X	0					[ACx] ← [ACx] ∨ data8 Logical Inclusive-OR immediate with Accumulator A or B.
	ORA ORB	data8	2	2				X	X	X	X				[ACx] ← [ACx] - data8 - C Subtract with borrow (carry) immediate from Accumulator A or B.
	SBCA SBCB	data8	2	2			X	X	X	X	X				[ACx] ← [ACx] - data8 Subtract immediate from Accumulator A or B.
	SUBA SUBB	data16	3	4				X	X	X	X				[D] ← [D] - data16 Subtract immediate from double Accumulator D.

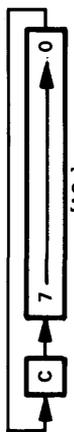
Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status								Operation Performed		
					E	F	H	I	N	Z	V	C			
Branch on Condition (Continued)	BVC	disp8	2	3											V = 0
	BVS	disp8	2	3											V = 1
	LBCC	disp16	4	5(6)											[PC] ← [PC] + 2, if the given condition is not true.
	LBCCS	disp16	4	5(6)											Note that BHS and BCC are different mnemonics for the same operation code, as are BLO and BCS.
	LBEO	disp16	4	5(6)											[PC] ← [PC] + disp16 + 4 if the given condition is true:
	LBGE	disp16	4	5(6)											C = 0
	LBGT	disp16	4	5(6)											C = 1
	LBHI	disp16	4	5(6)											Z = 1
	LBHS	disp16	4	5(6)											N ≠ V = 0
	LBLE	disp16	4	5(6)											Z V(N ≠ V) = 0
	LBLO	disp16	4	5(6)											C V Z = 0
	LBLE	disp16	4	5(6)											C = 0
	LBLE	disp16	4	5(6)											Z V(N ≠ V) = 1
	LBLE	disp16	4	5(6)											C = 1
	LBLE	disp16	4	5(6)											C V Z = 1
LBLE	disp16	4	5(6)											N ≠ V = 1	
LBLE	disp16	4	5(6)											N = 1	
LBLE	disp16	4	5(6)											Z = 0	
LBLE	disp16	4	5(6)											N = 0	
LBLE	disp16	4	5(6)											V = 0	
LBLE	disp16	4	5(6)											V = 1	
															[PC] ← [PC] + 4 if the given condition is not true.
															Note that LBHS and LBCC are different mnemonics for the same operation code, as are LBL0 and LBCCS. A long branch instruction takes 6 cycles to execute if it performs the branch and 5 cycles otherwise.
Register to Register Move	EXG	R1, R2	2	8											[R1] ↔ [R2] Exchange contents of specified registers. No effect on Status register (CC) unless R1 or R2 is Status register.
	TFR	R1, R2	2	7											[R2] ← [R1] Transfer contents of R1 to R2. No effect on Status register (CC) unless R2 is CC.

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed			
					E	F	H	I	N	Z	V	C						
Register-Register Operate	ABX		1	3													$[X] \leftarrow [X] + [B]$ Add unsigned contents of Accumulator B to Index Register X.	
	MUL		1	11						X							$[D] \leftarrow [A] \times [B]$ Multiply unsigned numbers in Accumulators A and B and place result in D. Carry flag takes the value of bit 7 of Accumulator B.	
	SEX		1	2				X									$[A] \leftarrow FF_{16}$ if bit 7 of Accumulator B is 1. $[A] \leftarrow 00_{16}$ if bit 7 of Accumulator B is 0. Transform an 8-bit twos complement number in B into a 16-bit twos complement number in D.	
Register Operate	ASLA } ASLB }		1	2		X		X		X		X					 $[ACx] \leftarrow 0$, $V = N \neq C$ Arithmetic shift left Accumulator A or B. Bit 0 is set to 0.	
	ASRA } ASRB }		1	2		X		X		X		X					 Arithmetic shift right Accumulator A or B. Bit 7 stays the same.	
	CLRA } CLRB }		1	2					0	1	0	0	0					$[ACx] \leftarrow 00_{16}$ Clear Accumulator A or B.
	COMA } COMB }		1	2				X		X	0	1	0					$[ACx] \leftarrow [ACx]$ Ones complement contents of Accumulator A or B.
	DAA		1	2				X		X	X	X	X					Decimal adjust Accumulator A. Convert contents of Accumulator A (assumed to be the binary sum of BCD operands) to BCD format. Carry is set if it was previously set or if the adjustment results in a carry.
	DECA } DECB }		1	2				X		X	X	X	X					$[ACx] \leftarrow [ACx] - 1$ Decrement (by 1) contents of Accumulator A or B. Set Overflow flag if result is $7F_{16}$ and clear Overflow flag otherwise.

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status								Operation Performed	
					E	F	H	I	N	Z	V	C		
Register Operate (Continued)	INCA INCB		1	2				X	X	X		X		$[ACx] \leftarrow [ACx] + 1$ Increment (by 1) contents of Accumulator A or B. Set Overflow flag if result is 80 ₁₆ and clear Overflow flag otherwise.
	LSLA LSLB		1	2			X	X	X	X		X		 $\leftarrow 0, V = N \nabla C$ Logical shift left Accumulator A or B. Bit 0 is set to 0. Same as ASL.
	LSRA LSRB		1	2				0	X	X		X		 Logical shift right Accumulator A or B. Bit 7 is set to 0.
	NEGA NEGB		1	2			X	X	X	X		X		$[ACx] \leftarrow 00_{16} - [ACx]$ Two's complement (negate) contents of Accumulator A or B. Set Carry flag if result is 00 ₁₆ and clear Carry flag otherwise. Set Overflow flag if result is 80 ₁₆ and clear Overflow flag otherwise.
	ROLA ROLB		1	2				X	X	X		X		 Rotate Accumulator A or B left through Carry flag.
	RORA RORB		1	2				X	X	X		X		 Rotate Accumulator A or B right through Carry flag.

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status								Operation Performed	
					E	F	H	I	N	Z	V	C		
Stack (Continued)	PULS } PULU }	reg, list	2	5+										<p>Test post byte and load registers from specified stack as follows:</p> <p>Condition:</p> <p>b0 = 1: [CC] ← [SP], [SP] ← [SP] + 1 b1 = 1: [A] ← [SP], [SP] ← [SP] + 1 b2 = 1: [B] ← [SP], [SP] ← [SP] + 1 b3 = 1: [DP] ← [SP], [SP] ← [SP] + 1 b4 = 1: [X(HI)] ← [SP], [SP] ← [SP] + 1 [X(LO)] ← [SP], [SP] ← [SP] + 1 b5 = 1: [Y(HI)] ← [SP], [SP] ← [SP] + 1 [Y(LO)] ← [SP], [SP] ← [SP] + 1 b6 = 1: [U(HI)] or [S(HI)] ← [SP], [SP] ← [SP] + 1 [U(LO)] or [S(LO)] ← [SP], [SP] ← [SP] + 1 b7 = 1: [PC(HI)] ← [SP], [SP] ← [SP] + 1 [PC(LO)] ← [SP], [SP] ← [SP] + 1</p> <p>Pull all, none, or any subset of registers from the specified stack, except for the Pointer to that Stack. Status register bits are determined by byte pulled from Stack.</p> <p>Execution time increases by one cycle for each byte pulled.</p>

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status										Operation Performed		
					E	F	H	I	N	Z	V	C					
Interrupt	CWAI	data8	2	20													<p>[CC] ← [CC] ^ data8. This may clear CC bits. E ← 1 [S] ← [S] - 1, [[S]] ← [PC(LO)] [S] ← [S] - 1, [[S]] ← [PC(HI)] [S] ← [S] - 1, [[S]] ← [U(LO)] [S] ← [S] - 1, [[S]] ← [U(HI)] [S] ← [S] - 1, [[S]] ← [Y(LO)] [S] ← [S] - 1, [[S]] ← [Y(HI)] [S] ← [S] - 1, [[S]] ← [X(LO)] [S] ← [S] - 1, [[S]] ← [X(HI)] [S] ← [S] - 1, [[S]] ← [DP] [S] ← [S] - 1, [[S]] ← [B] [S] ← [S] - 1, [[S]] ← [A] [S] ← [S] - 1, [[S]] ← [CC]</p> <p>Stores all registers in Hardware Stack and waits for an interrupt. When non-masked interrupt occurs, vectors to corresponding interrupt service routine. Note that a fast interrupt (FIRQ) service routine will be entered with all registers saved, but RTI will restore them correctly since CWAI sets E flag. CWAI does not float the system busses.</p>

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status							Operation Performed	
					E	F	H	I	N	Z	V		C
Interrupt (Continued)	RTI		1	6/15									<p>Pull registers from Hardware Stack in accordance with value of E flag in Status register.</p> <p>If E = 0, pull the subset:</p> <p>ICC ← [(SI), (S) ← (S) + 1 PC(HI) ← [(SI), (S) ← (S) + 1 PC(LO) ← [(SI), (S) ← (S) + 1</p> <p>If E = 1, pull the full complement of registers:</p> <p>ICC ← [(SI), (S) ← (S) + 1 IA ← [(SI), (S) ← (S) + 1 IB ← [(SI), (S) ← (S) + 1 IDP ← [(SI), (S) ← (S) + 1 IX(HI) ← [(SI), (S) ← (S) + 1 IX(LO) ← [(SI), (S) ← (S) + 1 IY(HI) ← [(SI), (S) ← (S) + 1 IY(LO) ← [(SI), (S) ← (S) + 1 IU(HI) ← [(SI), (S) ← (S) + 1 IU(LO) ← [(SI), (S) ← (S) + 1 IP(HI) ← [(SI), (S) ← (S) + 1 IP(LO) ← [(SI), (S) ← (S) + 1</p> <p>Status register bits are as removed from the Hardware Stack.</p>
	SWI		1	19									<p>Save all registers in the Hardware Stack and transfer control to interrupt subroutine. Vectors are in:</p>
	SWI2		2	20									FFFF and FFFB for SWI
	SWI3		2	20									FFFF and FFF5 for SWI2 FFF2 and FFF3 for SWI3
													E ← 1
													(S) ← (S) - 1, [(SI) ← PC(LO)]
													(S) ← (S) - 1, [(SI) ← PC(HI)]
													(S) ← (S) - 1, [(SI) ← IU(LO)]
													(S) ← (S) - 1, [(SI) ← IU(HI)]
													(S) ← (S) - 1, [(SI) ← IY(LO)]
												(S) ← (S) - 1, [(SI) ← IY(HI)]	
												(S) ← (S) - 1, [(SI) ← IX(LO)]	
												(S) ← (S) - 1, [(SI) ← IX(HI)]	

Appendix A. A Summary of the 6809 Instruction Set (Continued)

Type	Mnemonic	Operand(s)	Bytes	Cycles	Status							Operation Performed	
					E	F	H	I	N	Z	V		C
Interrupt (Continued)	SYNC		1	2									<p> $[S] \leftarrow [S] - 1$, $[SI] \leftarrow [DP]$ $[S] \leftarrow [S] - 1$, $[SI] \leftarrow [B]$ $[S] \leftarrow [S] - 1$, $[SI] \leftarrow [A]$ $[S] \leftarrow [S] - 1$, $[SI] \leftarrow [CC]$ $[PC] \leftarrow$ interrupt vector Note that the SWI disables the regular and fast interrupts, whereas SWI2 and SWI3 do not affect either one. Stop processing instructions. Float system busses and wait for an interrupt. When an interrupt occurs, resume processing as follows: a. if interrupt is enabled, transfer control to the service routine. b. if interrupt is disabled, continue execution at next instruction in sequence. </p>
Status	ANDCC	data8	2	3									<p> $[CC] \leftarrow [CC] \wedge$ data8 Logically AND immediate data with contents of status register. Used to clear bits of status register by logically AND-ing them with '0's. </p>
	ORCC	data8	2	3									<p> $[CC] \leftarrow [CC] \vee$ data8 Logically (Inclusive) OR immediate data with contents of status register. Used to set bits of status register by logically OR'ing them with '1's. </p>
No Operations	BRN LBRN NOP	disp8 disp16	2 5 2	3 4 1									<p> Branch never. This is a No operation. Long branch never. This is a No operation. No operation. </p>

B

Summary of 6809 Indexed and Indirect Addressing Modes

Type	Form	Non-indirect		Cycles +	Bytes +	Indirect		Cycles +	Bytes +
		Assembler Form	Post-Byte Op-code			Assembler Form	Post-Byte Op-code		
Constant Offset from R	No Offset	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5-Bit Offset	n,R	ORRnnnnn	1	0		Defaults to 8-bit		
	8-Bit Offset	nn,R	1RR01000	1	1	[nn,R]	1RR11000	4	1
	16-Bit Offset	mmnn,R	1RR01001	4	2	[mmnn,R]	1RR11001	7	2
Accumulator Offset from R	A — Register Offset	A,R	1RR00110	1	0	[A,R]	1RR10110	4	0
	B — Register Offset	B,R	1RR00101	1	0	[B,R]	1RR10101	4	0
	D — Register Offset	D,R	1RR01011	4	0	[D,R]	1RR11011	7	0
Auto Increment/ Decrement R	Increment by 1	,R+	1RR00000	2	0		Not allowed		
	Increment by 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	Decrement by 1	,-R	1RR00010	2	0		Not allowed		
	Decrement by 2	,--R	1RR00011	3	0	[,--R]	1RR10011	6	0
Constant Offset from PC	8-Bit Offset	label,PCR	1XX01100	1	1	[label,PCR]	1XX11100	4	1
	16-Bit Offset	label,PCR	1XX01101	5	2	[label,PCR]	1XX11101	8	2
Extended Indirect	16-Bit Address	—	—	—	—	[mmnn]	10011111	5	2

R = X, Y, U, or S RR: 00 = X 10 = U
 XX = Don't Care 01 = Y 11 = S

Note: This chart conforms to Motorola nomenclature; their use of square brackets [] indicates to the assembler that the addressing mode is indirect — thus, their use of [] differs from the use in Appendix A.

C-2 6809 Instruction Codes, Memory Requirements, and Execution Times

Address Mode	Inherent		Immediate		Direct		Extended		Indexed/Indirect		Relative		notes		
	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes		label or displacement	
Operand Form				data8 or data16	adr8	adr16			See Appendix B						
Instruction Mnemonic	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes	Object Code	No. of Cycles	No. of Bytes
CMPS	11 8C	5	4	11 9C	7	3	11 BC	8	4	11 AC	7+	3+			
CMPU	11 83	5	4	11 93	7	3	11 B3	8	4	11 A3	7+	3+			
CMPX	8C	4	3	9C	6	3	BC	7	3	AC	6+	2+			
CMPY	10 8C	5	4	10 9C	7	3	10 BC	8	4	10 AC	7+	3+			
COM				03	6	2	73	7	3	63	6+	2+			
COMA	43	2	1												
COMB	53	2	1												
CWAI	19	2	1	3C	20	2									
DAA															
DEC	4A	2	1	0A	6	2	7A	7	3	6A	6+	2+			
DECA	4A	2	1												
DECB	5A	2	1												
EOA	88	2	2	98	4	2	B8	5	3	AB	4+	2+			
EOB	C8	2	2	D8	4	2	F8	5	3	E8	4+	2+			
EXG	1E	8	2	0C	6	2	7C	7	3	6C	6+	2+			2
INC	4C	2	1												
INCA	5C	2	1												
INCB				0E	3	2	7E	4	3	6E	3+	2+			
JMP				9D	7	2	BD	8	3	AD	7+	2+			
LBCC															
LBGS															
LBGL															
LBGE															
LBGT															
LBHI															
LBHS															
LBLE															
LBLO															
LBLS															
LBLT															
LBMM															
LBME															
LBPL															
LBRA															
LBRI															
LBRS															
LBVS															
LDA	86	2	2	96	4	2	B6	5	3	A6	4+	2+			1
LDB	C6	2	2	D6	4	2	F6	5	3	E6	4+	2+			1
LDD	CC	3	3	DC	5	2	FC	6	3	EC	5+	2+			

C-4 6809 Instruction Codes, Memory Requirements, and Execution Times

Address Mode	Inherent:		Immediate		Direct		Extended		Indexed/Indirect		Relative		notes				
	Object Code	No. of Cycles	No. of Bytes	data8	data16	Object Code	No. of Cycles	No. of Bytes	adr16	Object Code	No. of Cycles	No. of Bytes		label or displacement	Object Code	No. of Cycles	No. of Bytes
Operand Form																	
Instruction Mnemonic																	
STU						DF	5	2			FF	6	3		EF	5+	2+
STX						9F	5	2			BF	6	3		AF	5+	2+
STY						10 9F	6	3			10 BF	7	4		10 AF	6+	3+
SUBA						80	2	2			B0	5	3		A0	4+	2+
SUBB						C0	2	2			F0	5	3		E0	4+	2+
SUBD						83	4	3			83	7	3		A3	6+	2+
SWI	3F	19	1														
SWZ	10 3F	20	2														
SW3	11 3F	20	2														
SYNC	13	2	1														
TFR						1F	7	2									
TST	4D	2	1			0D	6	2			7D	7	3		6D	6+	2+
TSTA	5D	2	1														
TSTB																	

Note 1: The cycle count in parentheses applies if the branch is taken.

Note 2: The immediate data in this instruction's object code is always an encoded register specification. See the description of this instruction in Chapter 22 for details.

Note 3: A PSH or PUL instruction requires 5 cycles plus one cycle for each byte pushed or pulled.

D

6809 Instruction Object Codes in Numerical Order

The following symbols and abbreviations appear in this appendix:

adr8	8-bit address
adr16	16-bit address
data8	8-bit data
data16	16-bit data
dd	8-bit data
dd dd	16-bit data
label	The destination of a Jump or Branch
mm	8-bit displacement in the object code
mm nn	16-bit displacement in the object code
pp	post byte for indexed and indirect addressing
qq	8-bit address
ssqq	16-bit address

6809 Instruction Object Codes in Numerical Order

Object Code ¹	Instruction ^{2, 3}	Addressing Mode
00 qq	NEG adr8	Base page (direct)
03 qq	COM adr8	Base page (direct)
04 qq	LSR adr8	Base page (direct)
06 qq	ROR adr8	Base page (direct)
07 qq	ASR adr8	Base page (direct)
08 qq	ASL adr8 / LSL adr8	Base page (direct)
09 qq	ROL adr8	Base page (direct)
0A qq	DEC adr8	Base page (direct)
0C qq	INC adr8	Base page (direct)
0D qq	TST adr8	Base page (direct)
0E qq	JMP adr8	Base page (direct)
0F qq	CLR adr8	Base page (direct)
10 21 mm nn	LBRN label	Relative
10 22 mm nn	LBHI label	Relative
10 23 mm nn	LBLE label	Relative
10 24 mm nn	LBHS label / LBCC label	Relative
10 25 mm nn	LBLO label / LBCC label	Relative
10 26 mm nn	LBNE label	Relative
10 27 mm nn	LBEC label	Relative
10 28 mm nn	LBVC label	Relative
10 29 mm nn	LBVS label	Relative
10 2A mm nn	LBPL label	Relative
10 2B mm nn	LBMI label	Relative
10 2C mm nn	LBGE label	Relative
10 2D mm nn	LBTL label	Relative
10 2E mm nn	LBGT label	Relative
10 2F mm nn	LBLE label	Relative
10 3F	SWI2	Inherent
10 83 dd dd	CMPD data 16	Immediate
10 8C dd dd	CMPY data 16	Immediate
10 8E dd dd	LDY data 16	Immediate
10 93 qq	CMPD adr8	Base page (direct)
10 9C qq	CMPY adr8	Base page (direct)
10 9E qq	LDY adr8	Base page (direct)
10 9F qq	STY adr8	Base page (direct)
10 A3 pp ¹	CMPD indexed forms	Indexed / indirect
10 AC pp ¹	CMPY indexed forms	Indexed / indirect
10 AE pp ¹	LDY indexed forms	Indexed / indirect
10 AF pp ¹	STY indexed forms	Indexed / indirect
10 B3 ss qq	CMPD adr16	Extended (direct)
10 BC ss qq	CMPY adr16	Extended (direct)
10 BE ss qq	LDY adr16	Extended (direct)
10 BF ss qq	STY adr16	Extended (direct)
10 CE dd dd	LDS data 16	Immediate

Note 1. The post byte may be followed by two bytes, one byte, or no byte. See Appendix B and the discussion of the post byte in Chapter 3 for more details. Appendix E lists all possible post bytes and the operand forms that produce them.

Note 2. Some instructions have two mnemonics. In each such case, we show both forms, separated by a slash (/).

Note 3. Appendix B displays the "indexed forms" for operands in the indexed and indirect addressing modes.

Note 4. In the instructions EXG and TFR, the processor interprets the second byte (the immediate data) as designating the source and destination registers.

Note 5. In the instructions PSHS, PULS, PSHU, and PULU, the processor interprets the second byte (the immediate data) as designating which registers are to be included in the transfer of data to or from the stack.

6809 Instruction Object Codes in Numerical Order (Continued)

Object Code ¹	Instruction ^{2, 3}	Addressing Mode
10 DE qq	LDS adr8	Base page (direct)
10 DF qq	STS adr8	Base page (direct)
10 EE pp ¹	LDS indexed forms	Indexed / indirect
10 EF pp ¹	STS indexed forms	Indexed / indirect
10 FE ss qq	LDS adr16	Extended (direct)
10 FF ss qq	STS adr16	Extended (direct)
11 3F	SWI3	Inherent
11 83 dd dd	CMPI data16	Immediate
11 8C dd dd	CMPS data16	Immediate
11 93 qq	CMPI adr8	Base page (direct)
11 9C qq	CMPS adr8	Base page (direct)
11 A3 pp ¹	CMPI indexed forms	Indexed / indirect
11 AC pp ¹	CMPS indexed forms	Indexed / indirect
11 B3 ss qq	CMPI adr16	Extended (direct)
11 BC ss qq	CMPS adr16	Extended (direct)
12	NOP	Inherent
13	SYNC	Inherent
16 mm nn	LBRA label	Relative
17 mm nn	LBSR label	Relative
19	DAA	Inherent
1A dd	ORCC data8	Immediate
1C dd	ANDCC data8	Immediate
1D	SEX	Inherent
1E dd	EXG data8	Register ⁴
1F dd	TFR data8	Register ⁴
20 mm	BRA label	Relative
21 mm	BRN label	Relative
22 mm	BHI label	Relative
23 mm	BLS label	Relative
24 mm	BCC label / BHS label	Relative
25 mm	BCS label / BLO label	Relative
26 mm	BNE label	Relative
27 mm	BEQ label	Relative
28 mm	BVC label	Relative
29 mm	BVS label	Relative
2A mm	BPL label	Relative
2B mm	BMI label	Relative
2C mm	BGE label	Relative
2D mm	BLT label	Relative
2E mm	BGT label	Relative
2F mm	BLE label	Relative
30 pp ¹	LEAX indexed forms	Indexed / indirect
31 pp ¹	LEAY indexed forms	Indexed / indirect
32 pp ¹	LEAS indexed forms	Indexed / indirect
33 pp ¹	LEAU indexed forms	Indexed / indirect
34 dd	PSHS data8	Register ⁵
35 dd	PULS data8	Register ⁵
36 dd	PSHU data8	Register ⁵
37 dd	PULU data8	Register ⁵
39	RTS	Inherent (Stack)
3A	ABX	Inherent
3B	RTI	Inherent (Stack)
3C dd	CWAI data8	Immediate
3D	MUL	Inherent
3F	SWI	Inherent
40	NEGA	Accumulator
43	COMA	Accumulator
44	LSRA	Accumulator
46	RORA	Accumulator

D-4 6809 Instruction Object Codes

6809 Instruction Object Codes in Numerical Order (Continued)

Object Code ¹	Instruction ^{2, 3}	Addressing Mode
47	ASRA	Accumulator
48	ASLA / LSLA	Accumulator
49	ROLA	Accumulator
4A	DECA	Accumulator
4C	INCA	Accumulator
4D	TSTA	Accumulator
4F	CLRA	Accumulator
50	NEGB	Accumulator
53	COMB	Accumulator
54	LSRB	Accumulator
56	RORB	Accumulator
57	ASRB	Accumulator
58	ASLB / LSLB	Accumulator
59	ROLB	Accumulator
5A	DECB	Accumulator
5C	INCB	Accumulator
5D	TSTB	Accumulator
5F	CLRB	Accumulator
60 pp ¹	NEG indexed forms	Indexed / indirect
63 pp ¹	COM indexed forms	Indexed / indirect
64 pp ¹	LSR indexed forms	Indexed / indirect
66 pp ¹	ROR indexed forms	Indexed / indirect
67 pp ¹	ASR indexed forms	Indexed / indirect
68 pp ¹	ASL / LSL indexed forms	Indexed / indirect
69 pp ¹	ROL indexed forms	Indexed / indirect
6A pp ¹	DEC indexed forms	Indexed / indirect
6C pp ¹	INC indexed forms	Indexed / indirect
6D pp ¹	TST indexed forms	Indexed / indirect
6E pp ¹	JMP indexed forms	Indexed / indirect
6F pp ¹	CLR indexed forms	Indexed / indirect
70 ss qq	NEG adr16	Extended (direct)
73 ss qq	COM adr16	Extended (direct)
74 ss qq	LSR adr16	Extended (direct)
76 ss qq	ROR adr16	Extended (direct)
77 ss qq	ASR adr16	Extended (direct)
78 ss qq	ASL adr16 / LSL adr16	Extended (direct)
79 ss qq	ROL adr16	Extended (direct)
7A ss qq	DEC adr16	Extended (direct)
7C ss qq	INC adr16	Extended (direct)
7D ss qq	TST adr16	Extended (direct)
7E ss qq	JMP adr16	Extended (direct)
7F ss qq	CLR adr16	Extended (direct)
80 dd	SUBA data8	Immediate
81 dd	CMPA data8	Immediate
82 dd	SBCA data8	Immediate
83 dd dd	SUBD data16	Immediate
84 dd	ANDA data8	Immediate
85 dd	BITA data8	Immediate
86 dd	LDA data8	Immediate
88 dd	EORA data8	Immediate
89 dd	ADCA data8	Immediate
8A dd	ORA data8	Immediate
8B dd	ADDA data8	Immediate
8C dd dd	CMPX data16	Immediate
8D mm	BSR label	Relative
8E dd dd	LDX data16	Immediate
90 qq	SUBA adr8	Base page (direct)
91 qq	CMPA adr8	Base page (direct)
92 qq	SBCA adr8	Base page (direct)
93 qq	SUBD adr8	Base page (direct)

6809 Instruction Object Codes in Numerical Order (Continued)

Object Code ¹	Instruction ^{2, 3}	Addressing Mode
94 qq	ANDA adr8	Base page (direct)
95 qq	BITA adr8	Base page (direct)
96 qq	LDA adr8	Base page (direct)
97 qq	STA adr8	Base page (direct)
98 qq	EORA adr8	Base page (direct)
99 qq	ADCA adr8	Base page (direct)
9A qq	ORA adr8	Base page (direct)
9B qq	ADDA adr8	Base page (direct)
9C qq	CMPX adr8	Base page (direct)
9D qq	JSR adr8	Base page (direct)
9E qq	LDX adr8	Base page (direct)
9F qq	STX adr8	Base page (direct)
A0 pp ¹	SUBA indexed forms	Indexed / indirect
A1 pp ¹	CMPA indexed forms	Indexed / indirect
A2 pp ¹	SBCA indexed forms	Indexed / indirect
A3 pp ¹	SUBD indexed forms	Indexed / indirect
A4 pp ¹	ANDA indexed forms	Indexed / indirect
A5 pp ¹	BITA indexed forms	Indexed / indirect
A6 pp ¹	LDA indexed forms	Indexed / indirect
A7 pp ¹	STA indexed forms	Indexed / indirect
A8 pp ¹	EORA indexed forms	Indexed / indirect
A9 pp ¹	ADCA indexed forms	Indexed / indirect
AA pp ¹	ORA indexed forms	Indexed / indirect
AB pp ¹	ADDA indexed forms	Indexed / indirect
AC pp ¹	CMPX indexed forms	Indexed / indirect
AD pp ¹	JSR indexed forms	Indexed / indirect
AE pp ¹	LDX indexed forms	Indexed / indirect
AF pp ¹	STX indexed forms	Indexed / indirect
B0 ss qq	SUBA adr16	Extended (direct)
B1 ss qq	CMPA adr16	Extended (direct)
B2 ss qq	SBCA adr16	Extended (direct)
B3 ss qq	SUBD adr16	Extended (direct)
B4 ss qq	ANDA adr16	Extended (direct)
B5 ss qq	BITA adr16	Extended (direct)
B6 ss qq	LDA adr16	Extended (direct)
B7 ss qq	STA adr16	Extended (direct)
B8 ss qq	EORA adr16	Extended (direct)
B9 ss qq	ADCA adr16	Extended (direct)
BA ss qq	ORA adr16	Extended (direct)
BB ss qq	ADDA adr16	Extended (direct)
BC ss qq	CMPX adr16	Extended (direct)
BD ss qq	JSR adr16	Extended (direct) ¹
BE ss qq	LDX adr16	Extended (direct)
BF ss qq	STX adr16	Extended (direct)
C0 dd	SUBB data8	Immediate
C1 dd	CMPB data8	Immediate
C2 dd	SBCB data8	Immediate
C3 dd dd	ADDD data16	Immediate
C4 dd	ANDB data8	Immediate
C5 dd	BITB data8	Immediate
C6 dd	LDB data8	Immediate
C8 dd	EORB data8	Immediate
C9 dd	ADCB data8	Immediate
CA dd	ORB data8	Immediate
CB dd	ADDB data8	Immediate
CC dd dd	LDD data16	Immediate
CE dd dd	LDU data16	Immediate
D0 qq	SUBB adr8	Base page (direct)
D1 qq	CMPB adr8	Base page (direct)

6809 Instruction Object Codes in Numerical Order (Continued)

Object Code ¹	Instruction ^{2, 3}	Addressing Mode
D2 qq	SBCB adr8	Base page (direct)
D3 qq	ADDD adr8	Base page (direct)
D4 qq	ANDB adr8	Base page (direct)
D5 qq	BITB adr8	Base page (direct)
D6 qq	LDB adr8	Base page (direct)
D7 qq	STB adr8	Base page (direct)
D8 qq	EORB adr8	Base page (direct)
D9 qq	ADCB adr8	Base page (direct)
DA qq	ORB adr8	Base page (direct)
DB qq	ADDB adr8	Base page (direct)
DC qq	LDD adr8	Base page (direct)
DD qq	STD adr8	Base page (direct)
DE qq	LDU adr8	Base page (direct)
DF qq	STU adr8	Base page (direct)
E0 pp ¹	SUBB indexed forms	Indexed / indirect
E1 pp ¹	CMPB indexed forms	Indexed / indirect
E2 pp ¹	SBCB indexed forms	Indexed / indirect
E3 pp ¹	ADDD indexed forms	Indexed / indirect
E4 pp ¹	ANDB indexed forms	Indexed / indirect
E5 pp ¹	BITB indexed forms	Indexed / indirect
E6 pp ¹	LDB indexed forms	Indexed / indirect
E7 pp ¹	STB indexed forms	Indexed / indirect
E8 pp ¹	EORB indexed forms	Indexed / indirect
E9 pp ¹	ADCB indexed forms	Indexed / indirect
EA pp ¹	ORB indexed forms	Indexed / indirect
EB pp ¹	ADDB indexed forms	Indexed / indirect
EC pp ¹	LDD indexed forms	Indexed / indirect
ED pp ¹	STD indexed forms	Indexed / indirect
EE pp ¹	LDU indexed forms	Indexed / indirect
EF pp ¹	STU indexed forms	Indexed / indirect
F0 ss qq	SUBB adr16	Extended (direct)
F1 ss qq	CMPB adr16	Extended (direct)
F2 ss qq	SBCB adr16	Extended (direct)
F3 ss qq	ADDD adr16	Extended (direct)
F4 ss qq	ANDB adr16	Extended (direct)
F5 ss qq	BITB adr16	Extended (direct)
F6 ss qq	LDB adr16	Extended (direct)
F7 ss qq	STB adr16	Extended (direct)
F8 ss qq	EORB adr16	Extended (direct)
F9 ss qq	ADCB adr16	Extended (direct)
FA ss qq	ORB adr16	Extended (direct)
FB ss qq	ADDB adr16	Extended (direct)
FC ss qq	LDD adr16	Extended (direct)
FD ss qq	STD adr16	Extended (direct)
FE ss qq	LDU adr16	Extended (direct)
FF ss qq	STU adr16	Extended (direct)

E

6809 Post Bytes in Numerical Order

Post Byte	Operand Form ¹						
00	0,X	37	-9,Y	6E	14,S	B4	[,Y]
01	1,X	38	-8,Y	6F	15,S	B5	[B,Y]
02	2,X	39	-7,Y	70	-16,S	B6	[A,Y]
03	3,X	3A	-6,Y	71	-15,S	B8	[nn,Y]
04	4,X	3B	-5,Y	72	-14,S	B9	[mmnn,Y]
05	5,X	3C	-4,Y	73	-13,S	BB	[D,Y]
06	6,X	3D	-3,Y	74	-12,S	BC	[nn,PC] ³
07	7,X	3E	-2,Y	75	-11,S	BD	[mmnn,PC] ³
08	8,X	3F	-1,Y	76	-10,S	BF	[mmnn]
09	9,X	40	0,U	77	-9,S	C0	,U+
0A	10,X	41	1,U	78	-8,S	C1	,U++
0B	11,X	42	2,U	79	-7,S	C2	,-U
0C	12,X	43	3,U	7A	-6,S	C3	,--U
0D	13,X	44	4,U	7B	-5,S	C4	,U
0E	14,X	45	5,U	7C	-4,S	C5	B,U
0F	15,X	46	6,U	7D	-3,S	C6	A,U
10	-16,X	47	7,U	7E	-2,S	C8	nn,U
11	-15,X	48	8,U	7F	-1,S	C9	mmnn,U
12	-14,X	49	9,U	80	,X+	CB	D,U
13	-13,X	4A	10,U	81	,X++	CC	nn,PC ²
14	-12,X	4B	11,U	82	,-X	CD	mmnn,PC ²
15	-11,X	4C	12,U	83	,--X	D1	[,U++]
16	-10,X	4D	13,U	84	,X	D3	[,--U]
17	-9,X	4E	14,U	85	B,X	D4	[,U]
18	-8,X	4F	15,U	86	A,X	D5	[B,U]
19	-7,X	50	-16,U	88	nn,X	D6	[A,U]
1A	-6,X	51	-15,U	89	mmnn,X	D8	[nn,U]
1B	-5,X	52	-14,U	8B	D,X	D9	[mmnn,U]
1C	-4,X	53	-13,U	8C	nn,PC ²	DB	[D,U]
1D	-3,X	54	-12,U	8D	mmnn,PC ²	DC	[nn,PC] ³
1E	-2,X	55	-11,U	91	[,X++]	DD	[mmnn,PC] ³
1F	-1,X	56	-10,U	93	[,--X]	DF	[mmnn]
20	0,Y	57	-9,U	94	[,X]	E0	,S+
21	1,Y	58	-8,U	95	[B,X]	E1	,S++
22	2,Y	59	-7,U	96	[A,X]	E2	,-S
23	3,Y	5A	-6,U	98	[nn,X]	E3	,--S
24	4,Y	5B	-5,U	99	[mmnn,X]	E4	,S
25	5,Y	5C	-4,U	9B	[D,X]	E5	B,S
26	6,Y	5D	-3,U	9C	[nn,PC] ³	E6	A,S
27	7,Y	5E	-2,U	9D	[mmnn,PC] ³	E8	nn,S
28	8,Y	5F	-1,U	9F	[mmnn]	E9	mmnn,S
29	9,Y	60	0,S	A0	,Y+	EB	D,S
2A	10,Y	61	1,S	A1	,Y++	EC	nn,PC ²
2B	11,Y	62	2,S	A2	,-Y	ED	mmnn,PC ²
2C	12,Y	63	3,S	A3	,--Y	F1	[,S++]
2D	13,Y	64	4,S	A4	,Y	F3	[,--S]
2E	14,Y	65	5,S	A5	B,Y	F4	[,S]
2F	15,Y	66	6,S	A6	A,Y	F5	[B,S]
30	-16,Y	67	7,S	A8	nn,Y	F6	[A,S]
31	-15,Y	68	8,S	A9	mmnn,Y	F8	[nn,S]
32	-14,Y	69	9,S	AB	D,Y	F9	[mmnn,S]
33	-13,Y	6A	10,S	AC	nn,PC ²	FB	[D,S]
34	-12,Y	6B	11,S	AD	mmnn,PC ²	FC	[nn,PC] ³
35	-11,Y	6C	12,S	B1	[,Y++]	FD	[mmnn,PC] ³
36	-10,Y	6D	13,S	B3	[,--Y]	FF	[mmnn]

Note 1: See Appendix B for addressing modes which the operand forms represent.

Note 2: May appear in source listing in the form label,PCR.

Note 3: May appear in source listing in the form [label,PCR].

Index

- A register. See Accumulator A
- ABA, 22-1
- Absolute addresses, 10-17. *See also* base page direct addressing, extended direct addressing
- Absolute loader, 2-17
- ABX, 3-39, 22-2-3
 - difference from accumulator offset addressing, 3-29
 - difference from LEAX instruction, 22-2-3
- Access, 17-12, 17-14
- Access time, 12-10, 21-5
- Accumulator, 3-4, 4-1-3
 - A, 3-4
 - B, 3-4
 - D, 3-4, 3-10
 - differences between A and B, 3-4, 4-3, 8-5
- Accumulator offset addressing mode, 3-28-29, 4-9, 7-6, 9-14-15, 19-14
 - difference from ABX instruction, 3-29
- Accumulator offset indirect addressing mode, 3-29-31, 9-14, 12-13, 12-14
- Accuracy, 8-3-4, 8-6
- ACIA. *See* 6850 ACIA or 6551 ACIA
- Acknowledgment from a 6820 PIA, 13-8, 13-9, 13-10
- Active transitions on a 6820 PIA control line, 13-4, 13-7-9, 13-38-39, 15-8
- ADC, 5-8, 8-1, 8-3, 8-5, 8-7, 22-3-4
- A/D converters, 13-13-46
- ADD, 4-2, 5-6, 8-5, 8-16, 15-27, 22-4-22-5
 - execution diagrams, 3-9, 3-12, 3-14, 3-16, 3-21-24, 3-27, 3-33, 22-4-5
- ADDD, 4-7-8, 8-4, 8-5, 22-5-6
 - execution diagrams, 3-10, 3-34, 22-5-6
- Adding Carry flag to Accumulator, 5-9
- Adding entry to list example, 9-2-3
- Addition:
 - BCD, 8-4-6, 18-11-12
 - binary, 4-2, 4-7-8, 8-2-4, 18-4-5, 18-10-11
 - decimal, 8-4-6, 18-11-12
 - 8-bit, 4-2, 18-10-11
 - multiple-precision, 8-2-6, 18-4-5, 18-11-12
 - 16-bit, 4-7-8
- Address arrays, 3-29-30, 3-34-36, 9-14, 12-13-14
- Address field, 2-1, 2-2, 2-10-12, 3-6, 3-45, 3-48-50
 - general description of options, 2-10-12
 - options in standard 6809 assembler, 3-48-50
- Address register, 5-4. *See also* index register X, index register Y, stack pointer S, stack pointer U
- Addressing bit in 6820 PIA control register, 13-3, 13-7
- Addressing modes. *See also* base page direct addressing, extended direct addressing, extended indirect addressing, immediate addressing, indexed addressing modes, indirect addressing
 - general description, 3-6
 - 6809, 3-7
 - specific descriptions, 3-7-38
 - symbols, 3-49
- Alarms, 15-1
- Alphabetizing strings, 6-1
- Analog-to-digital (A/D) converters, 13-44-47
- AND, 4-3-4, 13-10, 13-30-31, 15-30, 22-6-7
 - clearing bits, 13-10, 13-31, 15-30, 22-7
 - masking, 4-3-4, 13-30
 - testing bits, 13-13, 13-30-31, 22-7
- ANDCC, 3-5, 8-3, 15-5, 18-4, 22-6, 22-8-9
 - masks for clearing individual flags, 22-8
- Apostrophe indicating ASCII character, 3-49, 6-5, 6-8
- Architecture of 6809 CPU, 3-3-5
- Argument lists, 11-3-8
- Arithmetic, 8-1-20
 - add-ons, 8-20
 - algorithms, 21-5, 21-6
 - high-speed, 8-8, 8-12, 8-20
 - references, 21-6
 - tables, 4-8-11
- Arithmetic and logical expressions, 2-12, 3-50, 7-11
- Arithmetic processing units, 8-20
- Arithmetic shift, 4-3, 6-10, 8-1, 8-7, 8-12, 22-11
- Array, 3-20, 3-31-32, 5-3, 8-7-8. *See also* data structures
 - base address, 3-20, 3-28, 4-9
 - index, 3-6, 3-20, 4-9
 - multi-dimensional, 8-7-8
 - of addresses, 3-29-30, 3-34-36, 9-13-14, 12-13-14
 - one-dimensional, 3-20, 5-4
 - processing, 3-31-3-32, 5-4
- ASCII character code, 6-1-2
 - assembler format, 3-49, 6-5, 6-8
 - binary conversion program, 7-8-10
 - comparison with BCD, 6-2, 6-8
 - decimal conversion program, 7-6-7
 - FCC directive, 3-46-47
 - hexadecimal conversion program, 7-6-7
 - letter offset, 7-2
 - 7-bit version, 6-1
 - table, 6-2
 - validity checking, 7-7
- ASCII code table, 6-2
- ASCII strings, entry of, 2-12, 6-8. *See also* FCC directive
- ASCII to decimal conversion example, 6-1, 7-6-7
- ASCII to EBCDIC conversion, 3-28-29
- ASL, 4-3, 5-14, 8-12, 8-15, 13-14, 13-31, 22-9-10
 - multiplying by small integers, 7-8, 9-13
 - testing bit 6, 13-14
 - testing bit 7, 6-10, 13-14
- ASR, 22-10-11
- Assembler directives, 2-1, 2-5-10, 2-12-14
 - standard 6809 assembler, 3-46-48
- Assembler-related errors, 19-15-16
- Assemblers, 1-5-8, 2-1-18
 - address field, 2-1, 2-2, 2-10-12
 - advantages, 1-5, 1-6
 - applications, 1-13
 - arithmetic and logical expressions, 2-12
 - choice, 1-7
 - comments, 2-14
 - conditional assembly, 2-12-13
 - definition, 1-5
 - delimiters, 2-2-3
 - directives, 2-1, 2-5-10, 2-12-14
 - disadvantages, 1-7-8

- Assembler (Continued)
 - error messages, 2-16
 - errors from use, 19-15-16
 - features, 1-6-7
 - field structure, 2-1-3
 - formats, 2-2
 - inputs and outputs, 1-6
 - labels, 2-3-4, 2-10, 4-7
 - location counter, 2-11-12, 4-6
 - macros, 2-13-14
 - operations codes, 1-4, 2-4-5
 - pseudo-operations, 2-1, 2-5-10, 2-12-14
 - rules, 1-6
 - standard 6809 version, 3-45-50
 - symbol table, 2-7
 - types, 2-15
- Assembly-time arithmetic, 2-12, 3-50, 7-10
- Asterisk in standard 6809 assembler:
 - before a line of comments, 3-45
 - current value of location counter, 3-49, 4-6
- Asynchronous input/output:
 - documentation of programs, 18-1-2, 18-5-7
 - handshake, 12-5-7
 - interrupt-driven, 15-28-30
 - 6820 PIA, 13-49-52, 15-29-30
 - 6850 ACIA, 14-1-6, 15-28-29
 - TTY procedures, 13-48
 - UARTs, 13-52
- Autodecrement, 3-31-36, 5-3, 6-6, 19-12
 - by 1, 3-32-33
 - by 2, 3-32, 3-33-36
 - compatibility with stack storage, 3-32, 10-5
 - execution diagram, 3-34
 - indirect, 3-34-36
 - initialization, 3-32, 3-35, 5-3, 19-13
- Autoincrement, 3-31-36, 5-2-3, 5-7, 6-6, 15-20, 19-12
 - alternative to DUL instruction, 22-59
 - by 1, 3-31-36, 5-6, 13-31-32
 - by 2, 3-32, 3-34-36, 5-17
 - compatibility with stack loading, 3-32, 10-6, 13-31
 - execution diagrams, 3-33, 3-35
 - indirect, 3-34-36
 - initialization, 3-32, 3-34, 5-6
- Automatic saving of registers, 15-4-5, 15-17
- Automatic (strobe) modes on a 6820 PIA, 13-7-10, 13-26, 13-27, 13-29, 13-40, 13-43
- Auxiliary carry flag. *See* Half-carry flag
- B Register. *See* Accumulator B
- Backwards branches, 4-6, 22-23
- Base address, 3-20, 3-26, 3-28, 3-30, 4-9, 4-10, 7-4
- Base page. *See* direct page
- Base page direct addressing, 3-4, 3-11-13, 4-1-2, execution diagrams, 3-11, 3-12
- Base register, 3-16, 3-17, 3-18, 3-19
- BASIC computer language, 1-9, 1-111
- Baud, 13-47
- Baud rate generator, 12-15
- Baudot character code, 6-1
- BCC, 22-11. *See also* BHS
 - after CMP instruction, 4-5-6, 9-10, 19-12, 19-17.
 - testing bit 0, 13-14
 - testing bit 7, 5-14, 13-14
- BCD representation, 7-8, 8-4-6. *See also* decimal numbers
 - addition, 8-4-6
 - counting, 8-6
 - subtraction, 8-5-6
- BCD-to-binary conversion example, 7-8
- BCS, 22-12. *See also* BLO
 - after CMP instruction, 4-6, 9-5, 9-11, 19-17
- BEQ, 22-12
 - checking for FF (hex), 13-19
 - checking for zero, 5-14, 19-22
 - comparing values, 4-5-6, 6-3
- BGE, 5-12, 22-13
- BGT, 5-12, 22-13-14
- BHI, 22-14-15
 - after CMP instruction, 4-6, 9-11, 19-7, 19-12, 19-18
- BHS, 22-15-16. *See also* BCC
 - use for clarity, 19-17
- Bidirectional capability of 6820 PIA, 13-37-38
- Binary machine language programs, 1-2-3
- Binary notation for masks, 4-3
- Binary numbers, 3-49, 4-3, 7-8-10
 - use of % sign to designate, 3-49, 4-3
- Binary rounding, 8-15
- Binary search, 9-4
- Binary-to-ASCII conversion example, 7-8-10
- Binary-to-hexadecimal conversion table, 1-3
- Bit-by-bit operations, 4-3
- BIT, 13-31, 14-6, 22-16-17
- Bit length, 8-3-4, 8-6
- Bit manipulation:
 - clearing bits, 4-4, 13-10, 13-22, 13-43, 15-6, 15-30, 22-7
 - complementing bits, 22-35. *See also* EOR
 - setting bits, 6-10, 13-10, 13-22, 13-43, 15-5, 15-13, 15-30, 22-56
 - testing bits, 13-13-14, 13-30-31, 15-8-9, 22-7
- Bit numbering, 3-3, 3-18
- Bit patterns for instructions, 1-2, 3-8, 3-18, 4-9, B-1
- Bit rate (for TTY), 13-48
- Bit rate generator, 12-15
- Blank code (in ASCII), 6-5, 6-8
- Blanking leading zeros, 6-7-8, 13-23-26
- BLE, 5-13, 22-17-18
- BLO, 22-18. *See also* BCS
 - use for clarity, 19-17
- Block. *See* array, data structures
- BLS, 22-18-19
 - after CMP instruction, 4-6, 7-3, 9-12, 19-12
- BLT, 5-13, 22-19-20
- BMI, 22-20
 - signed operations, 5-11
 - testing bit 6, 15-8-9
 - testing bit 7, 5-11, 5-15, 13-11, 15-8, 19-5
- BNE, 22-20-21
 - checking for carry after INC, 8-15
 - comparing values, 4-5, 6-6
 - loop control, 5-6, 5-7, 19-22
- Boldface type, 1-1
- Bootstrap loader, 2-17
- Borrow, 3-8, 4-5, 8-1, 8-6. *See also* Carry flag, subtraction
- Bottom-up design, 17-26
- BPL, 22-21
 - signed operations, 5-11
 - testing bit 7, 5-11, 13-11, 13-14, 13-39, 13-47, 15-9
- BRA, 3-37-8, 5-15, 22-21-23
- Brackets around addresses, 3-15, 3-26, 3-45
- Branch instructions, 2-3, 3-36-3-38, 4-6-7. *See also* relative addressing

- Breakpoint, 19-2-5, 19-8
 - clearing, 19-4
 - correcting return address, 15-14, 19-3
 - example of use, 19-25-26
 - inserting, 19-3
 - precautions, 19-5
 - return address, decrementing of, 15-14, 19-3
 - setting, 19-4-5
 - software interrupt instructions, use of, 19-3-5
- BRN, 22-23
- BSR, 10-1, 10-7, 10-17, 22-23-24
- Bubble sort, 9-10
- Buffer, 9-5, 15-11, 15-19-20, 15-22-23
- Buffered interrupts, 15-11, 15-19-20, 15-22-23
- BVC, 22-25
- BVS, 22-25
- Byte disassembly example, 4-4-5
- Byte-length data, 3-46
- C flag. *See* Carry flag
- Calculating relative offsets, 4-6, 5-6, 5-9-10, 5-13, 5-15
- Calculator chips,
- Calendar time, 15-25-27
- Call by name, 11-13
- Call by value, 11-13
- Call instruction. *See* BSR, JSR
- Carriage return character, 6-2, 6-4
- Carry (C) flag; 3-4, 3-5, 4-2, 4-3, 4-5, 5-8, 8-1, 8-6
 - adding to accumulator, 5-8
 - arithmetic use, 8-1, 8-3
 - clearing of, 8-3, 22-26
 - decimal adjust, 8-5, 8-6
 - definition, 3-4
 - effect of CMP, 4-5, 9-12, 19-12
 - effect of MUL, 3-4, 22-52
 - equality case, 9-11, 19-12, 19-17
 - instructions with no effect, 3-5, 8-3, 22-34, 22-28
 - inverted borrow, 8-6
 - parallel/serial conversion, 13-52
 - position in CCR, 3-3
 - serial/parallel conversion, 13-47
 - setting of, 8-3, 13-51, 22-64
 - shifts, 4-3, 8-12
 - subtraction, 8-5-6
- Case structure, 17-17, 17-18
- CBA, 22-26
- CCR. *See* condition code register
- Centering data reception, 12-8, 13-30, 13-48
- Changing the return address, 11-4, 11-5, 11-6, 11-8, 19-3
- Changing values in the stack, 15-13-15
- Character codes, 6-1-2
- Character manipulation, 6-1, 6-5, 6-8, 6-12
- Character strings, entry of, 2-12, 6-8. *See also* FCC directive
- Checking an ordered list example, 9-3-4
- Checklist, 19-10-11, 19-17-18, 19-22-24
- Checksum, 5-15, 8-12
- Circular shift. *See* ROL, ROR instructions
- Classes of data for testing, 20-3
- CLC, 3-44, 8-3, 22-26
- Cleaning up the stack, 11-8, 11-11, 11-13
- Clear condition codes, 22-8-9
- Clearing bits, 4-4, 13-10, 13-22, 13-43, 15-6, 15-30, 22-7
- Clearing breakpoints, 19-4
- Clearing Carry flag, 8-3, 22-26
- Clearing flags, 3-5, 8-3, 15-5, 15-7, 22-8-9. *See also* ANDCC instruction
- Clearing memory example, 4-4
- Clearing 6820 PIA interrupt flags, 13-3, 13-7, 13-11, 13-39, 13-40, 15-8, 15-16, 15-17, 15-19, 15-24, 19-16
- CLF, 15-5, 22-26
- CLI, 15-5, 22-26
- CLIF, 15-5, 22-26
- Clock interrupts, 15-24-24
- CLR, 4-4, 5-8, 6-4, 13-8, 13-11, 22-27
- CLV, 22-27
- CMPA, B, 22-28-29
 - branch instructions, 4-6
 - comparison with SUB, 7-7
 - confusion in use, 19-12
 - direction, 19-13
 - effect on Carry flag, 4-5, 5-4, 19-12
 - effect on Zero flag, 4-5, 6-4, 19-12
 - input instruction, 13-11
 - signed numbers, 5-12
 - unsigned numbers, 4-5, 5-12, 19-12
- CMPD, X, Y, U, S, 22-28, 22-29-30
 - checking an address register, 5-3, 7-10, 19-7
 - length of operation codes, 10-16
- Code conversion, 7-1, 7-7
 - ASCII to binary, 7-10-11
 - ASCII to decimal, 7-7-8
 - ASCII to EBCDIC, 3-28-29
 - BCD to binary, 7-8
 - decimal to ASCII, 6-1
 - decimal to binary, 7-8
 - decimal to seven-segment, 7-3-5, 13-24-25, 18-11, 19-17, 19-20, 20-1
 - hexadecimal to ASCII, 7-2-4
- Coding, IV-3, 17-2
- COM, 22-30-31
- Combining control information, 13-31-32
- Commas in operand field, 3-45
- Comments, 2-2, 2-15, 18-2-7
 - delimiters in 6809 assembler, 3-45
 - examples, 18-4-7
 - guidelines, 2-14-15, 18-2-4
 - questions that comments should answer, 18-5
- Common-anode display, 13-22, 13-23
- Common-cathode display, 13-22, 13-23
- Common programming errors, 19-11-16
- Communications with interrupt service routines, 15-10-11, 15-18-19, 15-29-21
- Comparison instructions. *See* CMPA, CMPD
- Compiler, 1-9, 1-10
- Complementary binary form, 13-45
- Computer program, 1-2
- Condition code. *See* flag
- Condition Code Register, 3-3, 15-5, 15-6, 15-14. *See also* ANDCC, ORCC
 - bit assignments, 3-3
 - flags, 3-4-5
- Conditional assembly, 2-12-13
- Condition branch instructions, 4-7, 9-4, 19-12, A-10-11
- Consecutive structure, 17-16
- Constant offset from base register, 3-17-23, 3-28, 5-12, 9-7, 9-8, 9-9, 15-14-15
 - comparison with accumulator offset, 3-28
 - from hardware stack pointer, 10-7, 15-14-15
 - short offset modes, 3-20, 3-21-23, 5-13
 - zero offset mode, 3-20, 3-21
- Constant offset from base register indirect, 3-25-27, 9-8-9, 10-17, 11-4

- Constant offset from program counter, 3-23-25, 10-17
 - comparison with program relative addressing, 3-37
- Control characters, 6-2, 6-4
- Control information, 13-30-32
- Control lines on 6820 PIA, 13-1, 13-3, 13-4, 13-6-8
 - use independent of parallel data port, 13-40, 13-43, 15-24
- Control ports, 12-2-4
- Control register:
 - in 6820 PIA, 13-3-10
 - in 6850 ACIA, 14-1, 14-3-5
- Converters:
 - A/D, 13-43-47
 - D/A, 13-40-43
 - microprocessor-compatible, 13-42
 - successive approximation, 13-44
- Cost of redesign, 21-5
- counter, 13-26-28
- Counting down, 5-6, 22-38
- Counting 1 bits, 6-10, 13-51
- Credit verification terminal example. *See also* verification terminal 16-9-13, 17-5-10, 17-13-14, 17-22-25, 17-29-31.
- Cross-assembler, 2-5
- Cross-compiler, 1-13
- Cross-reference, 2-16
- Current value of location counter, 2-11-12, 3-49, 4-6
- CWAI, 15-6, 15-17, 22-31-22-32
- D register. *See* Double accumulator
- DAA, 22-32-33
 - after series of additions, 8-14
 - effect, 8-5
 - example programs, 8-4-6, 8-12-14, 8-16
 - hexadecimal to decimal conversion, 7-3
 - use after certain instructions, 8-5-6
- D/A converter, 13-40-43
- Darlington transistor, 13-22
- Data accepted flag, 15-21
- Data accepted signal, 13-9
- Data-address confusion, 2-11, 4-3, 19-13, 19-19
- Data direction register, 13-1, 13-3, 13-6-7
 - addressing, 13-6
 - establishing directions, 13-7
- DATA directive, 2-6-7, 2-10, 11-3. *See also* FCB, FCC, FDB directives
- Data flowchart, 17-4
- Data ready flag, 14-4, 15-18
- Data ready signal, 12-6, 13-8, 13-9
- Data structures, 3-20, 3-25-26, 9-5-9, 17-2, 17-31-32. *See also* array, lookup table
 - design of, 17-31-32
 - selection, 17-32
 - use, 3-20, 3-25-26, 9-6-9, 17-31-32
- Data transfer example, 4-1-2
- Debouncing switches, 13-14-15
- Debugging, IV-4, 19-1-27
 - assembler-related errors, 19-15-16
 - branches, 19-11
 - checklists, 19-10-11
 - code conversion example, 19-17-20
 - common programming errors, 19-11-15
 - definition, IV-4
 - examples, 19-17-26
 - interrupt-driven programs, 19-16
 - loops, 19-11
 - programming errors, 19-11-15
 - review, 20-5
 - sorting example, 19-21-26
 - tools, 19-1, 19-2-10
- Debugging examples, 19-17-26
- DEC (decrement by 1) instruction, 22-33-34
 - clearing bit 0, 15-12, 15-13, 15-30
 - effect on Carry flag (none), 8-6, 22-34
 - loop control, 5-5, 5-6, 22-38
- Decimal accuracy, 8-3, 8-6
- Decimal addition example, 8-4-6, 18-11-12
- Decimal arithmetic, 8-1, 8-4-6, 8-12-15, 8-16, 18-11-12
 - addition, 8-4-6, 18-11-12
 - rounding, 8-16
 - self-checking digit calculation, 8-12-15
 - subtraction, 8-5-6, 8-17
- Decimal default in address field, 2-10, 3-48, 19-15
- Decimal digits in ASCII, 6-1, 6-8
- Decimal numbers:
 - accuracy, 8-3, 8-6
 - addition example, 8-4-6, 18-11-12
 - arithmetic, 8-1, 8-4-6, 8-12-15, 8-16, 18-11-12
 - comparison with ASCII numbers, 6-2, 6-8
 - counting, 8-6
 - decrement, 8-6
 - increment, 8-6, 8-16
 - rounding, 8-16
 - subtraction, 8-5-6, 8-17
- Decimal subtraction, 8-5-6, 8-17
- Decimal to ASCII conversion, 6-1
- Decimal to binary conversion example, 7-9
- Decimal to hexadecimal conversion table, 1-3
- Decimal to seven-segment conversion example:
 - debugging, 19-17-20
 - documentation, 18-11
 - program example, 7-3-6
 - subroutine, 13-26, 18-11
 - testing, 20-1
- Decisions, 17-12, 17-14
- Decoder, 7-1, 13-23-24, 13-26-27
- Decoding commands, 9-3
- Default values, 3-12, 3-13, 3-48, 3-49, 19-15
 - addressing modes, 3-12, 3-13, 3-49, 19-15
 - direct page, 3-12, 3-48
 - errors, 19-15
 - number base in address field, 2-10, 3-48, 19-15
- Defensive programming, 2-3, 2-4, 2-7, 2-12, 2-15
- Defining inputs, 16-1
- Defining names, 2-7. *See also* EQU directive
- Defining outputs, 16-2
- Definition lists, 2-7, 18-8-9
- Degenerate (trivial) cases, 9-11, 19-12, 19-24
- Delay constant (for 1 ms delay), 12-11, 12-12
- Delay routines, 12-9-12, 13-6, 15-24-27
 - using real-time clock, 15-24-27
- Delimiters in standard assembler, 2-2-3, 3-45, 6-8
 - rules for use, 2-2-3
- Demultiplexer, 12-2, 12-3
- DES, 22-34
- Design decisions, limiting effects of, 17-12, 17-14
- Design of programs, 17-1-32. *See also* program design
- DEX, 3-41, 6-5, 22-34
- DEY, 22-34
- Digital-to-analog (D/A) converters, 13-40-43
- Direct addressing. *See also* base page
 - direct addressing
 - base page, 3-11-13, 4-1, 4-2, 21-2

- definition, 3-6
- execution diagrams, 3-11, 3-12, 3-13, 3-14
- extended, 3-13-14
- special meaning to 6809 manufacturers, 3-11
- Direct memory access (DMA), 12-8
- Direct page, use of, 3-12, 4-1, 4-2, 5-9, 21-2
- Direct page (DP) register, 3-4, 3-11, 3-12, 3-40, 7-8
 - default value, 3-12, 3-48
 - effect of Reset, 3-40
 - loading of, 7-8, 22-36, 22-73
 - SETDP directive, 3-48
- Directives, 2-1, 2-5-10, 2-12-14, 3-46-48. *See also* pseudo-operations
- Disabling interrupts, 15-2, 15-11-13, 15-14-15
- Disassembly table, D-2-6
- Displays, 13-20-29
 - multiple, 13-22-29
 - multiplexing of, 13-26-29
 - seven-segment, 13-22-29
 - single, 13-20-22
- Division, 8-1, 8-8-12
 - by power of 2, 8-1, 8-15
- DMA, 12-8
- DMA controller, 12-8
- Do-forever structure, 17-17
- Do-until structure, 17-16, 17-17
- Do-until structure, 17-16, 17-17
- Do-while structure, 17-16, 17-17, 17-19, 17-26
 - examples, 17-19
 - flowchart, 17-17
- Documentation, 13-11, 13-32, 8-1-14
 - comments, 2-15, 18-2-7
 - definition, IV-4
 - flowcharts, 18-7
 - input/output routines, 13-11
 - library forms, 18-9-12
 - memory maps, 18-7-8
 - package, 18-12-13
 - parameter and definition lists, 18-8-9
 - production software, 18-13
 - self-documenting programs, 18-1-2
 - status and control, 13-32
 - structured programs, 18-7
- Dollar sign, 3-9, 3-48
- Double accumulator, 3-3, 3-4. *See also* ADDD, CMPD and SUBD instructions
 - clearing of, 5-8
 - errors in use, 19-15
 - instructions, 3-4, 3-10, 3-33-34, 4-8, 22-6, 22-69
 - organization, 3-4, 4-8
 - shifting, 8-12
 - use in MUL instruction, 7-8, 8-7
- Double buffering of interrupts, 15-11, 15-20, 15-22-23
- Double-byte shifts, 8-12
- Double counting of switches, avoiding of, 13-15
- Doubling a binary number, 4-3, 5-15
- Doubling a decimal number, 8-14-15
- Doubly linked lists, 9-9
- Downward growth of stack, 10-5-6, 11-2
- DP register, 3-4, 3-11, 3-12, 7-8. *See also* page register
- Driver programs, 17-11. *See also* I/O driver
- Dummy operations on 6820 PIA, 13-9-11, 15-9, 15-17, 15-21
- Dump, 19-5-8
- Dynamic allocation, 10-17
- E flag, 3-5, 15-3-4, 15-6
 - meaning, 3-5, 15-4
 - position in CCR, 3-3
 - use, 15-6
- EBCDIC character code, 3-28, 6-1, 6-10
- Edge-sensitive interrupt (NMI), 15-7, 15-12
- Editing strings of digits, 6-8
- Effective address, 3-7, 3-14, 3-15, 4-9, 5-5-6, 6-5.
 - See also* addressing modes, indexed addressing modes, LEA instruction
- 8-bit summation example, 5-5-5-7, 18-10-11
- Empty state, 9-9
- Emulation, 20-2
 - in-circuit, 20-2
- Enabling and disabling interrupts. 15-11-14, 15-17, 15-31, 19-15. *See also* CLF, CLI, CWAI, SEF, SEI
 - automatic by CPU, 15-3
 - during a service routine, 15-14-15
 - errors, 19-16
 - instructions, 15-5-6, 15-17
 - restoring state after disable, 15-13
 - 6820 PIA, 15-13-14, 15-30
 - 6850 ACIA, 14-3-4, 15-29
 - when to disable, 15-11-12
 - when to enable, 15-12
- Encoder, 13-14, 13-38
- END directive, 2-9, 3-48
- Endless loop instruction, 17-17, 19-3
- Entire (E) flag, 3-5, 15-4, 15-6
- Entire state of processor, 15-4-5, 15-13-14
 - diagram, 15-5
 - indexed offsets, 15-14
- ENTRY directive, 2-9
- EOR, 5-16, 122-34-35
- Equal elements, 9-11-12
- Equality, checking for, 4-5
- EQUATE (EQU) directive, 2-7, 2-10
- Error-correcting codes, 6-10, 12-8
- Error-detecting codes, 6-10, 12-8
- Error exit from a service routine, 15-14
- Error handling, 16-3, 16-5, 16-8, 16-12, 17-14
 - recovery, 17-14
- Error messages, 2-16-17, 19-15-16
- Errors, 19-11-16
- Even parity generation example, 6-9-10, 13-52
- Examples:
 - format, II-1
 - interrupts, 15-5
 - standard memory addresses, "Guidelines for Examples" point (unnumbered) 7 under "Guidelines for Examples"
 - subroutines, 10-3
- Execution time:
 - delay routine, 12-11-12
 - division program, 8-12
 - indexed addressing modes, B-1
 - instructions, E-2-5
 - interrupt-related operation, 15-8
 - multiplication program, 8-8
 - reduction of, 21-4
 - searching methods, 9-5
- EXG, 22-35-37
 - jump and link, 10-17-18
 - loading direct page register, 7-7, 22-36
 - register codes, 22-37
 - uses, 22-36
- Expanding program stubs, 17-27-30

- Expressions, 2-12, 3-50, 7-11
- Extend addressing, 3-13-14
 - special meaning to 6809 manufacturers, 3-13
- Extend indirect addressing, 3-14-16, 15-16
 - post byte value (9F hex), 3-15
- EXTERNAL directive, 2-9
- External reference, 2-9
- Extra factor of 2 in relative address calculations, 3-36-37, 4-6-7, 22-22
- F flag, 3-5, 15-3-5, 15-15
 - position in CCR, 3-3
- Factor of 6 in decimal addition, 8-5
- Fast interrupt, 15-3-6, 15-10, 15-30
 - difference from regular interrupt, 15-6
 - during execution of CWAI instruction, 15-6
- Fast interrupt disable (F) flag, 3-5, 15-3-5, 15-15
- FCB directive, 3-46, 4-9, 7-6, 11-3-5, 11-7
- FCC directive, 3-46-47, 6-8, 11-3
- FDB directive, 3-46-47, 11-3-5, 11-7
- Field structure, 2-1-2, 3-45
- Fields in post-byte, 3-17-18, B-1
- FIFO, 9-5
 - 5-bit offset, 3-21-22, 5-13
 - 5357 A/D converter, 13-44-47
- Finding non-blank character example, 6-5-6
- FIRQ input, 15-3-6, 15-10, 15-30
- Fixed format, 2-2, 6-5
- Flags, 3-3-5, 8-3. *See also* ANDCC, condition code register, ORCC
 - bit assignments in CCR, 3-3
 - clearing, 3-5, 8-3, 22-8-9
 - effects of instructions, 3-4-5, A-3-19
- setting, 3-5, 8-3, 22-55-56
- Flexibility, 5-5, 9-6-7
- Flowcharting, 17-2-10, 18-7
 - advantages, 17-2-3
 - data, 17-4
 - disadvantages, 17-3-4
 - general version, 17-4
 - limitations, 17-3-4
 - programmer's version, 17-4
 - switch and light system, 17-4-5
 - switch-based memory loader, 17-5-6
 - symbols, 17-3
 - use in documentation, 18-7
 - verification terminal, 17-5-10
- Forcing direct or extended addressing, 3-12-13, 3-49
- Forcing 8-bit or 16-bit offsets, 3-49
- Format for storing 16-bit data and addresses, 3-46, 4-8, 10-5
- FORTRAN, 1-8-10
- FORTRAN-line (do) loops, 5-1-2, 17-16, 19-12
- Forward reference, 2-16
- Framing, 13-30
- Framing error, 13-48, 13-50, 14-5
- Free format, 2-2, 6-5
- Frequently used instructions, 3-1, 3-2
- General parameter passing techniques, 11-3-13
- General interrupt service routines, 15-30-31
- Global variable, 2-14
- H (half-carry) flag, 3-5, 8-5-6
 - effects of instructions, 8-5-6
 - need for, 8-5
 - position in CCR, 3-3
- Half-carry flag. *See* H flag
- Halt instruction. *See* CWAI, SYNC instructions
- Halving a decimal number, 8-15
- Hand assembly, 1-6
- Handshake
 - A/D converter, 13-43-47
 - definition, 12-5
 - diagrams, 12-6-7
 - 6820 PIA operating modes, 13-8-9
 - software equivalent, 15-11
- Hardware stack pointer, 3-4, 5-6, 10-3, 10-5-7, 10-17, 13-31-32, 15-3-5, 15-13-15. *See also* stack pointer S
- Hardware/software tradeoffs, 7-2, 12-9, 13-38, 13-52-53, 15-28, 21-5
- Hashing, 9-3
- Headings, 18-3
- Hexadecimal conversion table, 1-3
- Hexadecimal loader, 1-4-5
- Hexadecimal programs, 1-3-5
- Hexadecimal to ASCII conversion example, 7-2-3, 10-4-7
- High-level language, 1-8-14
- High-speed I/O devices, 12-2, 12-8
- Hold in the stack, 11-8-11
- Housekeeping directives, 2-5, 2-9-10
- Human factors, 3-33, 16-3-4.
- Human interaction, 6-8, 16-3, 16-6, 16-8, 16-12
- I flag, 3-5, 15-3, 15-5-6, 15-14
 - position in CCR, 3-3
 - saving and restoring, 15-13
- Identifying a key by scanning, 13-32 to 13-38
- If-then-else structure, 17-16, 17-19, 17-25
 - examples, 17-19
 - flowchart, 17-16
- Illegal indexed addressing modes, 3-17, 3-35
- Immediate addressing, 3-6, 3-9-11, 4-3, 4-9, 5-9
 - definition, 3-6
 - execution diagrams, 3-9-11
 - instructions lacking the mode, 3-11
- Implied memory address, 3-20, 3-32. *See also* autodecrement, autoincrement, zero offset indexed addressing mode
- Implied subtraction, 7-7. *See also* CMP instructions
- INC, 22-37-38
 - checking for FF, 13-19
 - counting, 6-4, 22-38
 - effect on Carry flag, 8-6, 22-38
 - rounding, 8-15
 - setting bit 0, 15-13, 15-30
 - 16-bit version, 8-15
- In-circuit emulation, 20-2
- Independence, 17-1, 17-12, 17-14
- Index, 3-6, 3-20, 4-9-10, 7-4-5
- Index calculations, 4-10, 8-7-8
- Index register X, 3-4, 4-9, 5-6-7, 22-29
 - preference over Y, S, and U registers, 5-7, 21-3
- Index register Y, 3-4, 5-6
- Indexed addressing modes, 3-16-36
 - accumulator offset, 3-28-29
 - accumulator offset indirect, 3-29-31
 - autodecrement, 3-31-34
 - autodecrement indirect, 3-34-36
 - autoincrement, 3-31-34
 - autoincrement indirect, 3-34-36
 - comparison with 6800 indexing, 3-40
 - constant offset from base register, 3-19-23
 - constant offset from program counter, 3-23-25
 - constant offset indirect, 3-25-28

- extended indirect, 3-14-16
- general description, 3-16-19
- illegal modes, 3-17, 3-35
- memory requirements, B-1
- notation, 3-19
- post-byte, 3-17-18, E-1
- 6800 indexing, comparison with, 3-40
- summary, B-1
- time requirements, B-1
- unimplemented modes, 3-17
- Indirect indexed addressing, 3-17, 3-25-27
- Inefficiency of high-level languages, 1-11
- Information-hiding principle, 17-14
- Inherent addressing, 3-6, 3-8
- Initial values in RAM, 2-9
- Initializing variables, 5-1, 5-6, 6-4, 19-11-12
- Input handshake, 12-5-6
- Input/output
 - categories, 12-2
 - chips, 12-14-15
 - comparison to memory, 12-1
 - examples, 13-12-52, 14-5-6
 - instructions, 13-10-11
 - interrupt-driven examples, 15-15-30
- I/O device table, 3-29-31, 3-34-35, 12-13-14
- I/O driver, 9-5
- INS, 22-39
- Inserting elements in linked lists, 9-7
- Instruction execution times, C-2-5
- Instruction set, 1-1, 3-1, 3-7, 3-38, 22-1, A-2
- Instruction tables
 - execution times, C-2-5
 - extensions from 6800 instructions, 3-39, 3-43-44
 - frequently used, 3-2
 - generalizations of 6800 instructions, 3-43
 - identical to 6800 instructions, 3-39, 3-42
 - implementations of missing 6800 operation codes, 3-44
 - memory usage, C-2-5
 - new instructions, 3-43-44
 - object codes in numerical order, D-2-6
 - occasionally used, 3-2
 - operation code mnemonics, 3-39
 - post-bytes in numerical order, E-1
 - seldom used, 3-3
 - summary, 3-38-39, A-2-19
- Instructions
 - bit patterns, 1-2, 3-18, 4-3
 - definition, 1-2-2
 - effects on flags, 3-4-5, A-3-19
 - execution times, C-2-5
 - extensions from 6800 instructions, 3-39, 3-43-44
 - frequently used, 3-2
 - generalizations of 6800 instructions, 3-43
 - identical to 6800 instructions, 3-39, 3-42
 - implementations of missing 6800 operation codes, 3-44, 6-5, 22-1
 - input/output, 13-10-11
 - interrupt-related, 15-6-7
 - memory usage, C-2-5
 - new, 3-43-44
 - object codes in numerical order, D-2 to D-6
 - occasionally used, 3-2
 - operation code mnemonics, 3-39, C-2-5
 - recommended use, 3-1
 - seldom used, 3-3
 - summary, 3-38-39, A-2-19
- Instructions that read and write memory, 13-11, 14-3, 14-5, 19-14
- Interchanges, 9-11, 19-22-23
- Intermediate carry flag. *See* Half-carry flag
- Interpolation, 21-3
- Interpreter, 1-12
- Interrupt disable flag, 3-5, 15-3, 15-5-6, 15-13-14
- Interrupt overhead, 15-15, 15-27
- Interrupt-related instructions, 15-6-7
- Interrupt response, 15-2-4
- Interrupt service routines, 15-8-11, 15-13-33
 - communications with main program 15-10-11
 - debugging, 19-16
 - general versions, 15-30-31
 - keyboard, 15-17-20
 - printer, 15-20-23
 - real-time clock, 15-23-28
 - startup, 15-15-17
 - subroutine use, 15-13
 - teletypewriter, 15-28-30
 - transfer of control, 15-2
- Interrupt vectors, 15-2, 15-4, 15-10
 - table, 15-10
- Interrupts
 - ACIA (6850), 15-8
 - advantages, 15-1
 - breakpoints, 19-3
 - characteristics, 15-1-2
 - disabling, 15-2, 15-11-15
 - disadvantages, 15-3
 - enabling, 15-2, 15-11-13, 15-17
 - executing general subroutines, 15-13
 - initialization, 15-12-13
 - instructions, 15-5-6
 - nonmaskable, 15-2, 15-7
 - overhead, 15-15, 15-27
 - PIA (6820), 15-7-8
 - polling systems, 15-2, 15-8-9
 - priority systems, 15-2, 15-24, 15-30-31
 - response, 15-2-4
 - service routines, 15-8-11, 15-13-33
 - 6809 system, 15-2-7
 - vectored systems, 15-2, 15-10
- Inverse, 4-11. *See also* ones complement
- Inverting decision logic, 19-12, 19-17, 19-20, 19-22
- INX, 3-41, 3-44, 6-5, 19-13, 22-39
- INY, 22-39
- IRQ input, 15-3, 15-5, 15-10, 15-14-15
- JMP, 3-11, 5-14, 9-13-14, 10-6, 19-15, 22-39-40
- JSR, 22-41-42
 - examples of, 10-6, 11-5, 11-10
 - execution time, 10-7
 - function, 10-1
 - lack of immediate addressing, 3-11
 - operation, 10-5-6
 - return address, 10-5-6, 11-3-4
- Jump and link instruction, 10-17-18, 22-36. *See also* EXG instruction
- Jump instructions. *See also* BRA, condition branch, EXG, JMP, JSR, RTS, RTI, SWI, and TFR
 - differences in addressing, 9-4, 22-40
 - EXG instruction, 22-36
 - jump table, 9-2-4
 - lack of immediate addressing, 3-11
 - meaning, 2-3
 - TFR, 22-73
- Jump table, 9-2-4, 22-40. *See also* case structure

- Keyboard interface, 9-14, 13-8-9, 13-22-40
 - encoded keyboard, 13-8-9, 13-38-40
 - unencoded keyboard, 13-32-38
- Keyboard interrupts, 15-17-20
- Keyboard scan, 13-34-38
- Labels
 - choice of, 2-4
 - definition, 2-3, 4-7
 - field, 2-2-3
 - first character, 3-46
 - in jump instructions, 2-3-4
 - meaning, 2-3, 2-7, 2-9
 - reasons for use, 2-3, 2-4, 4-7
 - selection rules, 2-4
 - 6809 assembler, 3-45-46
 - space after, 3-45
 - truncation, 2-4, 3-46
 - with assembler directives, 2-6-10, 3-48, 7-6
- Lamp test, 13-23-25
- Language level, choice of, 1-12-14
- Latching, 2-1-3, 13-1, 13-39-40
 - by 6820 PIA, 13-1, 13-39-40
- LBCC, 22-42
- LBCS, 22-42
- LBEQ, 22-42
- LBGE, 22-42-43
- LBGT, 22-43
- LBHI, 22-43
- LBHS, 22-43
- BLE, 22-43
- LBLO, 22-44
- LBS, 22-44
- BLT, 22-44
- LBMI, 22-44
- LBNE, 22-44-45
- LBPL, 22-45
- LBRA, 5-10, 22-45-46
- LBRN, 22-46
- LBSR, 22-47
- LBVC, 22-47
- LBVS, 22-47
- LDA,B, 4-1-2, 22-47-8
 - effects on flags, 5-11
 - execution diagrams, 3-29, 3-31, 3-35
- LDD, 4-8, 22-47-49
- LDS, 4-10, 10-3, 22-47
 - execution diagram, 3-11
- LDU, 4-10, 9-7, 10-16, 11-5, 11-7, 22-47
- LDX, 4-9-10, 5-6, 9-7, 9-15, 22-47
- LDY, 4-10, 5-7, 9-7, 10-16, 22-47
- LEA. *See also* indexed addressing modes, 6800
 - operation codes
 - addressing uses, 6-5, 11-4, 11-8, 19-3, 19-6, 21-3, 22-50
 - arithmetic uses, 6-5, 8-3, 8-6, 8-15, 11-8, 21-3, 22-51
 - cleaning up the stack, 10-17, 11-8, 11-11
 - description, 22-49-51
 - space allocation, 10-17, 11-8, 11-10, 19-6
- LED, 13-20-29
 - control of, 13-20-21
 - interface, 13-20-21
 - turn-on time, 13-21
- Length of registers, 3-3, 4-9
- Letter offset, 7-3
- Letters and numbers, confusing of, 2-4
- Letters, seven-segment codes for, 13-23, 13-26
- Library routines, 2-4, 10-1, 18-9-12
- Lightface type, 1-1
- Limited state of processor, 15-3, 15-4, 15-14. *See also*
 - E flag, fast interrupt, RTI instruction diagram, 15-4
 - indexed offsets, 15-14
- Linear structure, 17-16
- Link editor, 2-17
- Link register, 10-17-18, 22-36. *See also* EXG instruction
- Linked lists, 9-7-9
- Linking directives, 2-9
- Linking loader, 2-17
- List processing, 9-1-9
- Loader program, 2-17
- Local variable, 2-14
- Location counter, 2-8-9, 2-11-12, 3-49, 7-6
- Logic analyzer, 19-9-10
- Logical I/O device, 12-13-14
- Logical shift, 4-4, 8-14-15. *See also* LSR
- Long branch instructions, 3-37, 5-10, 22-42-47
- Lookup tables
 - arithmetic applications, 4-8-11
 - code conversion, 3-28-29, 7-1, 7-4-6, 13-26
 - I/O device assignment, 3-30, 3-34, 12-13
 - jump, 9-12-14, 22-40
 - keyboard, 9-14, 13-37-38
 - reducing execution time, 21-4
 - saving memory, 21-3
 - tradeoffs involved, 4-10-11
 - uses, 4-11
- Loops, 5-1-4. *See also* do-while structure
 - debugging, 19-11
 - decreasing execution time, 5-14, 6-4, 6-6, 21-4
 - execution time, 5-1
 - flowcharts, 5-2-3
 - sections, 5-1
 - structures, 17-16-17, 17-19, 17-21-22, 17-26
- LSL, 22-51. *See also* ASL instruction
- LSR, 22-51
 - digit shift, 4-4
 - division by 2, 8-15
 - normalization, 4-4, 13-31
 - testing bit 0, 13-14, 13-17, 14-5
- Machine-independence, 1-8-9
- Machine language, 1-1-5, 1-12
- Macro, 2-13-14, 19-13
- Macroassembler, 2-16
- Maintenance, 13-43, 1V-4, 18-13
- Maintenance manual, 18-13
- Major changes in systems, 21-4-5
- Majority logic, 12-8, 13-30
- Manual mode, 13-8, 13-10, 13-43, 13-47
- Manufacturer's mnemonics, 1-5
- Mark state on teletypewriter line, 13-48-49
- Mask, 4-3, 13-13, 13-14, 13-30-31, 13-35
 - by bit position, 13-14
 - notation, 4-3
- Maskable interrupt, 15-2-4
- Masking example, 4-3-4
- Master reset of 6850 ACIA, 14-3-5, 15-29
- Matrix keyboard, 13-33-34
- Maximum count in loops, need for, 6-4
- Maximum value
 - example program 5-10-12
 - subroutine, 10-10-11
- Medium-speed I/O devices, 12-2, 12-5-8
- Memory dump, 19-6-8
- Memory map, 18-7-8

- Memory-mapped I/O, 3-5, 13-10-11
- Memory operations, 4-3-4, 5-9. *See also* single-operand instructions
- Memory, special features of, 12-1
- Memory/time tradeoffs, 4-10-11, 21-4
- Memory usage, decreasing, 21-2-3
- Meta-assembler, 2-16
- Microassembler, 2-16
- Microprocessor analyzer, 19-9-10
- Microprocessor-compatible converters, 13-42
- Microprogramming, 2-16
- Millisecond delay program 12-9-12
- Minus sign, 3-19
- Missing 6800 operation codes, 3-41, 3-44
- Mnemonic, 1-4, 2-4-5
 - definition, 1-4
 - manufacturer's, 1-5
 - 6809 set, 3-39
 - standard, 1-5
- Mnemonic operation codes for 6809, 3-39
- Modifying programs for your microcomputer, 10-3, 13-13, 15-16
- Modular programming, 17-11-14
 - advantages, 17-11
 - disadvantages, 17-11-12
 - information-hiding principle, 17-14
 - principles, 17-12
 - review, 17-14
 - switch and light system, 17-12
 - switch-based memory loader, 17-13
 - verification terminal, 17-13-14
- Module, 17-11-12
- Monitor programs, 1-5, 4-5, 7-3, 19-4
 - breakpoints, 19-4
 - interrupt handling, 15-16
 - return of control, 4-2, 15-6, 22-71
 - stack pointer, 10-3
- Motorola standard assembler, 3-22, 3-45-50
- MUL, 7-9, 8-7, 22-51-52
 - effect on flags, 22-52
- Multibyte data, updating of, 15-12
- Multidimensional arrays, 8-7-8
- Multiple origins, reasons for, 3-47. *See also* ORG directive
- Multi-position switches, 9-14, 13-16-20
- Multiple-precision arithmetic, 8-1-6
- Multiple-precision binary addition example
 - commenting, 18-4-5
 - program example, 8-2-4
 - subroutine, 10-15-16, 11-6-8, 11-12-13
- Multiplexing displays, 13-25-29
- Multiplexing I/O devices, 12-2-5
- Multiplication, 7-9, 8-1, 8-7-8
 - by power of 2, 8-1, 8-14-15
- Multiplication example, 8-7-8
- Multiplying by small decimal numbers, 7-9
- N flag, 3-3, 3-5, 4-5, 5-12, 5-15, 13-11, 13-31
- Names, use and choice of, 2-4, 2-7, 2-10, 18-2, 18-8
- National 5357 A/D converter, 13-44-47
- NEG, 22-53-54
- Negative elements example, 5-9-10
- Negative flag, 3-5
 - BIT instruction, 13-31
 - meaning, 3-5, 4-5, 5-11
 - position in CCR, 3-3
 - signed numbers, 5-12, 5-15
 - testing bit 7, 13-11
 - use, 3-5, 13-11, 13-31
- Negative logic, 13-20, 13-23
- Negative numbers, 5-9-10
- Nested subroutines, 10-6, 10-17-18
- Nibble, 4-4
- NMI input, 15-2-3, 15-7-8, 15-10, 15-12
- Nonmaskable interrupt input. *See* NMI input
- NOP, 22-54-55
- Normalization example, 5-14-16
- Number base, choice of, 2-11-12, 3-48-49, 4-3
- Number sign, 3-9, 3-49, 4-3
- Number systems, default choice, 3-48, 19-15
- Number systems, identification of, 2-10, 3-48-49
- Numbers and letters, confusing of, 2-4
- Nybble, 4-4
- Object code
 - alphabetical order, C-2-5
 - definition, 1-2, 1-6
 - disassembly table, D-2-6
 - numerical order, D-2-6
 - post-bytes, 3-18, B-1, E-1
- Object program, 1-2, 1-6
- Occasionally used instructions, 3-2
- Octal programs, 1-3
- Offset, 3-16-17, 5-13
- Ones complement, 4-11, 4-14, 22-30-31
- Ones complement program, 4-11
- One-pass assembler, 2-16
- One-shot, 12-9
- Operand field, 2-2, 2-10-12, 3-6, 3-45, 3-48-50
- Operands, number of, 2-4-5
- Operation code, 2-1-2, 2-4-6
 - alphabetical list for 6809, 3-39, C-2-5
 - numerical order, D-2-6
- Operation interaction, 16-3, 16-6, 16-8, 16-12-13
- OR, 22-55-56
 - use in setting bits, 6-10, 13-10, 13-22, 13-31, 13-43, 15-13-14, 15-30
- ORCC, 3-5, 8-3, 13-52, 15-5, 22-55-56
 - masks for setting individual flags, 22-56
- Order of instruction execution, 5-7
- Ordered list example, 9-4-5
- ORIGIN (ORG) assembler directive, 2-8, 3-47, 4-10
- Output control assembler directives, 2-9-10
- Output handshake, 12-5, 12-7
- Output ready signal, 12-7, 13-40
- Overflow, 5-12
- Overflow (V) flag, 3-3, 35-5, 4-5, 5-12
 - definition, 3-5
 - position in CCR, 3-3
 - signed numbers, 5-12
- Overrun error, 13-52, 14-2
- P register. *See* condition code register, flags
- Parallel I/O, 13-16, 13-22, 13-32, 15-17-23
- Parallel interface devices, 12-14-15. *See also* 6820 Peripheral Interface Adapter
- Parallel interface standards, 12-14
- Parallel/serial conversion, 13-48, 13-51-52
- Parameter lists, 18-8-9
- Parameters, 10-2, 10-7, 10-9, 10-11, 10-14, 10-16
 - general passing techniques, 10-2, 11-3-13
 - types, 11-14
- Parentheses, 3-50
- Parentheses around addresses, 5-6
- Parity, 6-9-10, 12-8, 13-48, 13-52
- Parity generation, 6-9-10, 12-8, 13-52
- PASCAL, 1-8, 1-11

- Passing parameters
 - argument lists, 11-3-8
 - definition, 10-2
 - general methods, 11-3-13
 - in registers, 10-2, 10-7, 10-9, 10-11, 10-14, 11-3
 - in the stack, 10-2, 10-7, 11-8-13
- Pattern comparison
 - program example, 6-11-13
 - subroutine, 10-12-14
- PCR notation, 3-19, 3-25, 3-49, 10-17, 22-50
- Pending interrupts, 15-8
- Percentage sign, 3-49, 4-3
- Peripheral interface adapter. *See* 6820 Peripheral Interface Adapter
- Peripheral ready signal, 12-5, 12-7, 13-8, 13-11, 14-4
- Physical I/O device, 12-13
- PIA. *See* 6820 Peripheral Interface Adapter
 - Plus sign, 3-19
- Pointer, 5-3, 5-6
- Polling, 12-5, 13-11, 15-8-9
 - definition, 12-5
 - 6820 PIAs, 13-11, 15-8-9
 - of 6850 ACIAs, 15-8
- Polling interrupts, 15-2, 15-9-10, 15-29
 - disadvantages, 15-9
- Portability, 1-7, 1-9
- Position, determination of, 3-23-24, 10-17, 22-51
- Position-independent code, 3-23, 10-2, 10-17, 22-51
- Positive logic, 7-4, 13-20, 13-23
- Post-byte, 3-17-18, 3-40, 4-9, 5-13
 - bit definitions, 3-18
 - extended indirect addressing, 3-15
 - extra time and memory requirements, B-1
 - fields, 3-17-18
 - information contained in it, 3-17
 - list in numerical order, E-1
 - position in instruction, 3-17, 3-18
 - purpose, 3-17
- Post bytes, meanings in numerical order, E-1
- Postincrement, 3-17
- Power fail interrupt, 15-2, 15-7, 15-24
- Precedence rules, 3-50
- Predecrement, 3-17
- Prefix byte, 6-13
- Preindexed addressing, 3-17
- Printer interrupts, 15-20-23
- Priority interrupt controller (6828), 15-10
- Priority interrupts, 15-2, 15-6, 15-9, 15-24, 15-31
- Priority register, 15-2, 15-31
- Problem definition, 16-1-14, 17-32-33
 - definition, IV-3
 - examples, 16-4-13
 - factors, 16-1-4
 - review, 16-14, 17-33
 - switch and light system, 16-4-6
 - switch-based memory loader, 16-6-9
 - verification terminal, 16-9-13
- Procedure-oriented language, 1-8
- Processing requirements, 16-2, 16-5, 16-8, 16-12
- Production software, documentation of, 18-13
- Program, 1-2
- Program counter, 2-3, 3-4, 4-6, 10-1-2, 1-5-7
 - determination of current value, 3-23, 10-17, 22-51
- Program design, 17-1-34
 - definition, IV-3, 17-1
 - flowcharting, 17-2-10
 - modular programming, 17-11-14
 - principles, 17-1-2
 - review, 17-32-33
 - structured programming, 17-15-26
 - top-down design, 17-26-31
- Program logic manual, 18-13
- Program loops, 5-1-3
 - debugging, 19-11, 19-17, 19-22
 - reducing execution time, 6-4, 21-4
 - time/memory tradeoffs, 21-2, 21-4
- Program relative addressing, 3-23-25, 3-36-38
- Program speed, increasing, 21-4
- Program stub, 17-26-28
- Programmable I/O devices, 12-14-15. *See also* 6820 Peripheral Interface Adapter
- Programmable timer, 12-9
- Programmed I/O, 15-1
- Programmer's flowchart, 17-4
- Programming mode of 6809 processor, 3-3
- Programming time, division of, IV-3
- Pseudo-operations, 2-1, 2-5-10, 2-12-14, 3-46-48, 11-5, 3-46-48, 11-5
 - definition, 2-1, 2-5
 - general description, 2-5-10, 2-12-14
 - mixing with instructions, 11-5
 - standard 6809 assembler, 3-46-48
- PSH, 22-57-58
 - bit assignments for registers, 11-2-3
 - examples, 11-5, 11-10
 - operation, 11-1-3
 - order for storing registers, 11-2
 - passing parameters, 11-5, 11-8
 - reducing memory usage, 21-2
 - saving interrupt status, 15-13
 - saving registers, 10-9, 12-11
 - temporary storage, 9-12
- PUL, 22-58-59
 - alternative that affects flags, 22-59
 - bit assignments for registers, 11-2-3
 - examples, 11-5-7, 11-11
 - operation, 11-1-3
 - order for loading registers, 11-2
 - passing parameters, 11-5, 11-7, 11-11
 - replacement of RTS, 11-6, 11-11, 22-59, 22-63
 - restoring interrupt status, 15-13
 - restoring registers, 10-9, 11-6, 11-11, 12-11
 - temporary storage, 9-12
- Pull order for registers, 11-2
- Pushbutton, 13-12-16
- Push order for registers, 11-2
- Queue, 9-5-9
- RAM, initialization of, 2-9, 19-11-12
- Random test cases, 20-1, 20-3-4
- Range
 - 8-bit signed offset, 3-23, 3-36, 4-6, 5-8
 - 5-bit signed offset, 3-20, 5-12
 - offset from program counter, 3-36, 4-6, 5-8-9
- RDRF flag (in 6850 ACIA), 14-3-5
- Read-only bits, 11-3, 13-7
- Read-only memory, execution from, 2-9, 5-2, 11-3, 19-4, 19-15
 - argument lists, 11-3
 - breakpoints, 19-5
 - errors, 19-15
- Read strobe from 6820 PIA, 13-7-10
- Read/write (R/W) signal, use in addressing 6850 ACIA, 14-1-2

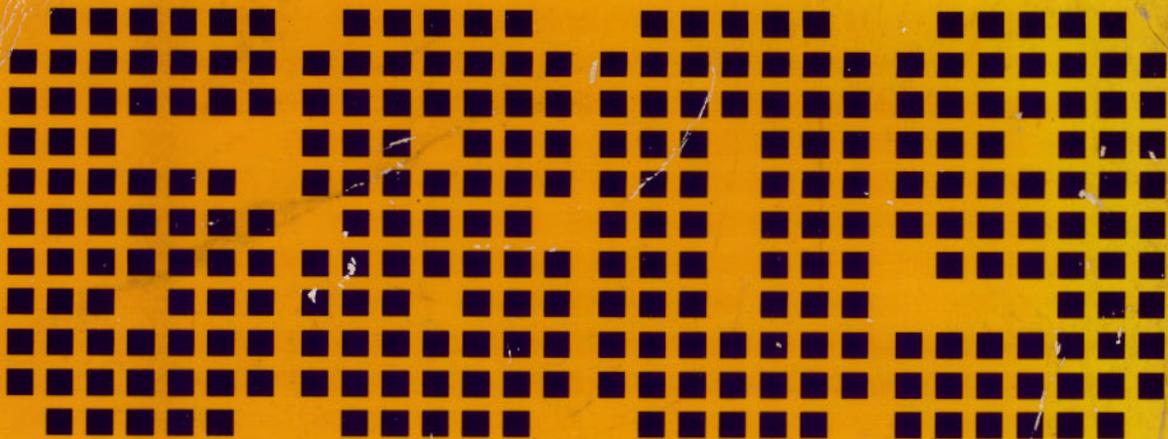
- Real-time clock, 15-23–28
 - clock time, 15-25
 - definition, 15-24
 - frequency, 15-24
 - high-frequency, 15-24, 15-28
 - priority, 15-24
 - service routines, 15-25
 - service time, 15-27–28
 - synchronization, 15-24
- Receive routine, 13-48–51, 14-5, 15-28–30
- Recovery from lethal errors, 17-14
- Recursive subroutine, 10-3
- Redesign of programs, IV-4, 21-1–6
- Reentrant programs, 10-2
 - definition, 10-2
 - examples, 10-7, 10-9, 10-11, 10-14, 11-6–7, 11-11
 - stack usage, 10-17
 - standard subroutines, 15-31
- Register addressing, 3-6, 3-8, 11-1–2
- Register designations after operation codes, 3-45
- Register dump, 19-5–6
- Registers, 3-3–4
 - coding in EXG, TFR instructions, 22-37
 - diagram, 3-3
 - indexed offsets in the stack, 15-14
 - order in the stack, 15-4–5, 15-14
 - preference in use, 21-3
 - pull order, 11-2
 - push order, 11-2
- Regular interrupt, 15-3, 15-4, 15-8, 15-15
- Relative addressing, 3-6, 3-36–38, 4-6, 5-8
- Relative offsets, calculation of, 4-6–7, 5-6–7, 5-9–10, 5-12, 5-15, 22-22–23
- Relocatable programs, 3-36, 10-2, 10-7, 10-11
- Relocating loader, 2-3, 2-17, 10-2
- Relocation constant, 2-3, 10-2
- Reorganizing programs to save execution time, 5-14, 6-4, 6-6, 21-4
- Repeat-until structure, 17-16
- RESERVE assembler directive, 2-8–10. *See also* RMB directive
- Reset, 15-8–9
 - address, 3-47, 15-9
 - effect on flags, 15-2, 15-15
 - effect on interrupt system, 15-2–3, 15-12, 15-15
 - 6820 PIA, 13-3, 13-39, 15-8, 15-15
 - 6850 ACIA, 14-3–5
 - vector, 15-10
- Resident assembler, 2-16
- Resuming programs after a breakpoint, 19-3, 19-5
- Return address, 10-1–2, 10-5–7
 - adjustment after breakpoint, 19-3
 - adjustment past argument lists, 11-4–6, 11-8
 - changing in stack, 15-14–15
 - in user stack, 22-59
 - restoring by EXG, 10-18
 - restoring by PUL, 11-6, 11-11, 22-59, 22-63
 - restoring by RTI, 15-6, 22-61–62
 - restoring by RTS, 10-6
 - saving by CWA1, 15-6, 22-31
 - saving by EXG, 10-17–18, 22-36
 - saving by interrupt response, 15-3–4
 - saving by JSR, 10-5–7
 - saving by PSH, 22-59
 - saving by SWI, 15-6, 22-70–71
 - user stack, 22-59
- Return instruction, 10-1. *See also* RTI, RTS
- Returning control to the operating system, 15-6, 22-71
- Revisions, documentation of, 18-3–4
- RMB directive, 3-47
- ROL, 22-59–60
 - double-length shifts, 8-12
 - serial I/O, 13-48–49
 - testing bits 6 or 7, 13-14
- Rollover, 13-38
- ROM simulator, 20-2
- ROR, 22-60–62
 - serial I/O, 13-48–49
- Rotate instructions. *See* ROL, ROR
- Rounding
 - after MUL, 22-52
 - binary, 8-15
 - decimal, 8-16
- RS-232 interface, 12-14
- RTI, 15-6, 16-13, 15-17, 15-19, 21-3, 22-61–62
 - reenabling interrupt status, 15-1
- RTS, 22-62–63
 - changing return address, 11-4
 - effect, 10-1–2
 - execution time, 10-7
 - multiple exits, 10-14
 - operation, 10-6
 - reducing execution time, 21-3
 - replacement by PUL, 11-6, 11-11, 22-63
- Run-time package, 1-11
- S flag. *See* Negative flag, Sign flag
- S register. *See* stack pointer S
- Sampling inputs, 12-8, 13-30
- Saving and restoring interrupt status, 15-13, 15-30
- SBA, 22-63
- SBC, 8-1, 22-63–64
- Searching examples, 9-1–5
- Searching methods, 9-4
- SEC, 8-3, 22-64
- SEF, 15-6, 22-65
- Segments, labeling of, 7-3, 13-23
- SEI, 15-6, 22-65
- SEIF, 15-6, 22-65
- Seldom used instructions, 3-3
- Self-assembler, 2-16
- Self-checking numbers
 - program example, 8-12–15
 - testing, 20-4
- Self-compiler, 1-11
- Self-documenting programs, 18-1–2
- Semaphore, 15-11
- Separating status information, 13-30–31
- Sequential execution of instructions, 1-2, 17-16, 19-14
- Sequential structure, 17-16
- Serial input/output, 13-48–52, 14-1–6, 15-28–30
 - interrupt-driven version, 15-28–30
 - LSI devices, 12-15, 13-52–53
 - order of transmitting bits, 13-48
 - 6850 ACIA, 14-1–6
 - standard interfaces, 12-14
 - teletypewriter I/O, 13-47–52
 - UARTs, 13-52
- Serial interface devices, 12-15, 14-1–6. *See also* 6850 ACIA
- Serial interface standards, 12-14
- Serial output from 6820 PIA, 13-10, 13-43, 13-47, 15-29
- Serial to parallel conversion, 13-48, 13-52, 15-28
- Set conditions codes, 3-5, 8-3, 13-52, 15-5, 22-55
- SETDP assembler directive, 3-48
- Setting bits, 6-10, 13-10, 13-43, 15-5, 15-30, 22-56

- Setting breakpoints, 19-3-5
- Setting directions in 6820 PIA, 13-1, 13-6-7
- Setting flags, 3-5, 8-3, 13-52, 15-5, 22-55-56
 - in the stack, 15-14, 15-15
- Setting 6820 PIA status bits, 13-3-4, 13-7-9, 13-11, 13-38-39, 13-44, 15-8
- Seven-segment code, 7-3-5, 13-22-24
 - conversion program, 7-3-5, 18-11
 - decimal digits, 7-3, 13-25
 - letter and symbols, 13-26
- Seven-segment displays, 7-3, 7-5, 13-22-29, 18-11, 19-17-20, 2-10
 - labeling of segments, 13-23
- 7447 seven-segment decoder/driven, 13-23-24
- 74148 priority encoder, 13-17
- SEV, 22-65
- SEX, 22-65-66
- Sharing of information (among modules), 17-12, 17-14
- Shift instructions. *See also* ASL, ASR, LSR, ROL, ROR instructions
 - double-length, 8-12
 - effects of, 19-13
 - Shift left example, 4-2-3
- Short data fields, processing of, 13-30-32
- Short (5-bit) offset indexed addressing mode, 3-20, 5-8
- Short relative branches, 5-8
- Sign extension, 5-12. *See also* SEX instruction
- Sign (negative) flag, 3-3, 4-5, 5-12, 5-14, 13-11, 13-31
- Signature analyzer, 16-3
- Signed numbers, comparison of, 5-12
- Signed offsets, 3-20, 3-23, 3-28-29, 3-36-37, 3-40, 4-6, 5-12, 22-2. *See also* ABX instruction
- Sigetics NE5018 D/A converter, 13-40-43
- Simulator program 19-8-9
- Single-bit errors, 6-10
- Single-entry, single-exit structures, 17-15-16
- Single-operand instructions, 3-8
 - accumulator, 3-8
 - application to memory, 4-3-4, 5-8
 - execution, 4-4
 - immediate addressing, lack of, 3-11
 - indexed addressing, 7-11
 - read-only memory or registers, 14-3, 19-14-15
 - use in I/O, 13-11, 14-3, 19-14
- Single-step mode, 19-2
 - example of use, 19-18-19
 - limitations, 19-2
- 16-bit addresses or data, storage of, 3-48, 4-8
- 16-bit decrement in memory, 15-14, 19-3
- 16-bit increment in memory, 8-15
- 16-bit instructions, 3-10, 3-13-14, 4-8, 4-10
 - autodecrement, 3-33-34
 - base page direct addressing, 3-11
 - double accumulator instructions, 4-8, 4-11, 5-8
 - extended direct addressing, 3-13
 - immediate addressing, 3-9-10
 - moving addresses, 9-6
 - operations on index registers and stack pointers, 4-10
 - transfers, 7-8
- 16-bit ones complement example, 4-11
- 16-bit registers, 3-3-4, 4-8-10, 5-6, 10-16
 - choice of, 5-6, 10-16, 21-3
 - transfer to or from stack, 11-2, 22-57
- 16-bit shift, 8-12
- 16-bit summation example, 5-6-8
- 6502 compatibility, 3-45, 10-5
- 6522 Versatile Interface Adapter (VIA), 12-15
- 6551 Asynchronous Communications Interface Adapter (ACIA), 12-15
- 6800 compatibility, 3-38-44
 - addressing modes, 3-40
 - differences, 3-41, 3-43
 - flags, 3-40, 22-50
 - indexing, 3-40
 - instructions, 3-41-44, 6-5, 8-3, 10-5, 22-1
 - object code, 3-38, 3-42
 - registers, 3-40
 - similarity, 3-38
 - stack pointer, 3-41, 10-5, 22-75
- 6800 operation codes, 3-41-44, 6-45, 8-3, 10-5, 22-1. *See also* CLCL, CLI, DEX, INX, SEC, SEI
- 6801 compatibility, 3-44, 8-12, 22-3
- 6820 Peripheral Interface Adapter (PIA), 12-15, 13-1
 - addresses, 13-3
 - automatic strobe mode, 13-7-10, 13-26-27, 13-29, 13-40, 13-43
 - B port drive, 13-21-22
 - bidirectional capability, use of, 13-37-38
 - block diagram, 13-2
 - clearing status bits, 13-3, 13-7, 13-11, 13-39, 15-9
 - control lines, 13-3-4, 13-6-10
 - control register, 13-3-10
 - data direction register, 13-1, 13-3, 13-6-7
 - differences between port A and port B, 13-7, 13-21
 - disabling interrupts, 15-11-12, 15-30
 - documentation problems, 13-11, 13-43
 - dummy operations, 13-9, 15-9, 15-17, 15-21, 15-24
 - enabling interrupts, 15-12-13, 15-30
 - examples of initialization, 13-8-20
 - general description, 13-1
 - indexed offsets for registers, 13-3, 13-16
 - initialization, 13-6-10
 - internal addressing, 13-3
 - interrupts, 15-7-8, 15-11-13
 - interrupt inputs, control of, 13-7, 15-7-8
 - latching, 13-1, 13-39-40
 - manual mode, 13-8, 13-10, 13-43, 13-47
 - output strobes, 13-7-10, 13-26-29, 13-43
 - pending interrupts, 15-8
 - polling, 15-9-10
 - read strobe, 13-7-10
 - registers, 13-1, 13-3
 - reset, 13-3, 13-39, 15-8
 - setting status bits, 13-3-4, 13-7-9
 - transferring data, 13-10-11
 - write strobe, 13-7, 13-9, 13-26-29, 13-43
- 6821 Peripheral Interface Adapter (PIA), 12-15, 13-1
- 6828 Priority Interrupt Controller, 15-10
- 6840 Programmable Timer, 15-24
- 6844 DMA Controller, 12-8
- 6846 Multifunction Support Device (ROM/IO/Timer), 12-9, 15-24
- 6850 Asynchronous Communications Interface (ACIA), 12-15, 14-1-6
 - addressing, 14-1, 14-3
 - block diagram, 14-2
 - control register, 14-1, 14-3-5
 - features, 14-3-4
 - initialization, 14-5, 15-29
 - interrupt mode, 15-28-29
 - interrupts, 15-8-9
 - master reset, 14-3-5, 15-29
 - polling
 - power-on reset, 14-5
 - read-only registers, 14-3

- receive routine, 14-5
- register contents, 14-3-4
- reset, 14-3-5, 15-29
- status register, 14-3, 14-5
- transmit routine, 14-6
- write-only registers, 14-3-4
- Slow I/O devices, 12-2-5
- Software delay routines, 12-9-12, 13-46
- Software development
 - coding, IV-3
 - debugging, 19-1-27, 20-5
 - documentation, 18-1-14
 - flowchart, IV-2
 - measuring progress, IV-3
 - problem definition, 16-1-14
 - program design, 17-1-34
 - redesign, 21-1-6
 - stages, IV-1-4
 - testing, 20-1-6
- Software handshake, 15-11
- Software/hardware tradeoffs, 7-1, 12-9, 13-38, 13-52-53, 15-27, 21-5
- Software interrupt instructions, 22-70-71
 - availability of SWI2 to end user, 15-6
 - breakpoint, 19-3-5
 - debugging use, 19-3-5
 - decrementing return address after breakpoint, 15-14, 19-3
 - effects, 15-6
 - reducing memory usage, 21-3
 - transferring control to operating system, 15-6, 22-71
 - trap, 15-6
 - uses, 15-6, 19-3-5, 22-71
 - vectors, 15-10
- Software simulator, 19-8-9, 20-2
- Sorting example
 - debugging, 19-21-26
 - program, 9-10-12
 - testing, 20-4
- Sorting methods, 9-11-12
- Source code, 1-6
- Square brackets indicating indirection, 3-15, 3-45, 3-49
- Space state, 13-48, 13-49
- Spaces in 6809 assembler statements, 3-45
- Spaces in strings of characters, 6-5, 6-8
- Speed, increasing of, 21-4
- STA,B, 3-11, 4-1-2, 22-66-67
- Stack
 - changing values saved in, 15-13-15
 - interrupts, 15-3-5
 - parameter passing, 10-2, 10-7, 11-8-13
 - subroutine return addresses, 10-1, 10-5, 22-59
 - temporary storage, 9-12, 10-9, 10-17, 11-8, 13-31
- Stack pointer, 3-3-4, 11-1-2
- Stack pointer S, 3-4
 - argument lists, 11-3-8
 - contents, 10-6
 - difference from stack pointer U, 3-4, 22-58
 - indexed offsets for registers, 15-13-15
 - initialization, 10-3, 10-5
 - interrupts, 15-3-5
 - JSR instruction, 10-5-7, 22-41-42
 - parameter passing, 10-2, 10-7, 11-8-13
 - return address, 10-6
 - RTI instruction, 10-6-7, 22-61-62
 - RTS instruction, 10-6-7, 22-62-63
 - subroutine use, 10-5-7
 - SWI instructions, 15-6, 15-10, 19-3-5, 22-70-71
 - temporary storage, 9-12, 10-9, 10-17, 11-8, 13-3, 15-13, 15-19, 15-30
 - usual assignments, 5-6
- Stack pointer U, 3-3, 3-4
 - difference from stack pointer S, 3-4, 22-58
 - index register, 3-17, 3-19, 3-23, 3-35, 5-7, 8-3, 19-6
 - passing parameters, 10-16, 11-2, 11-5-7, 11-11
 - subroutine linkages, 22-59
- Stack values, changing, 15-13115
- Standard mnemonics, 1-5
- Standard 6809 assembler (from Motorola), 3-45-50
 - address field, 3-48-50
 - addressing mode notation, 3-49
 - arithmetic and logical expressions, 3-50
 - automatic optimization of constant offset mode, 3-22
 - delimiters, 2-3, 3-45
 - field structure, 3-45
 - labels, 3-46, 3-48
 - pseudo-operations, 3-46-48
- Start bit, 12-5, 13-48, 13-49
- Start bit interrupt, 15-29-30
- Starting address of any array or table, 3-20, 3-28, 3-30, 4-9-10, 7-5-6, 8-7
- Starting code, searching for, 13-30
- Startup interrupt, 15-15-17
- Statement, 2-1
- Static allocation of temporary storage, 10-17
- Status flag. *See* condition code register flags
- Status information, 13-29-31
 - favored bit positions, 13-31
- Status register
 - 6809 CPU. *See* condition code register
 - 6850 ACIA, 14-3, 14-5
- STD(S,U,X,Y), 22-66-68
 - effects, 4-8
 - moving addresses, 9-6, 11-4, 11-6, 11-8
 - two-byte operation codes, 10-16, 22-68
- Stop bit, 12-5, 13-48, 13-52
- Storage requirements
 - ASCII, 6-2, 6-8
 - BCD, 6-2, 6-8, 7-8
- Stray interrupts, clearing of, 13-39, 13-47, 15-21
- String comparison, 6-11-13, 10-12-14
- String editing, 6-5-10
- String length
 - program example, 6-3-4
 - subroutine, 10-7-9, 11-4-6, 11-8-11
- Strings of characters, 6-1-16
- Strobe, 12-5, 13-38
- Strobe signal from 6820 PIA, 13-7, 13-9-10, 13-26-29, 13-40
- Structured programming, 17-15-26, 18-7
 - advantages, 17-19
 - disadvantages, 17-20
 - review, 17-25
 - rules, 17-26
 - structures, 17-15-19
 - switch and light system, 17-21
 - switch-based memory loader, 17-21-22
 - terminators, 17-26
 - use in documentation, 18-7
 - verification terminal, 17-22-25
 - when to use, 17-20
- Structured testing, 20-2-3
- Structures. *See* data structures, structured programming
- Stubs, 17-26-28
- SUB, 7-7, 8-5-6, 8-17, 22-68-69

- SUBD, 4-14, 8-5, 22-68-69
- Subroutine documentation, 10-3
 - examples, 10-5, 10-9, 10-11, 10-14, 10-16, 11-5, 11-7, 11-10, 11-13
 - library examples, 18-9-12
- Subroutine linkages
 - hardware stack, 10-5-6, 22-41
 - index register, 10-17-18, 22-36
 - user stack, 22-59
- Subroutine library, 10-1, 18-9-12
- Subroutines, 10-1-20, 15-13, 18-9-12
 - comparison with macros, 2-3
 - delay, 12-10-12
 - documentation, 10-3, 18-9-12
 - errors in use, 19-13
 - examples, 10-4-16, 11-4-13, 18-10-12
 - instructions, 10-1-2, 10-5-6
 - interrupt service routines, use by, 10-2, 15-13
 - library, 10-1, 18-9-12
 - linkages, 10-5-6, 10-17-18, 22-36, 22-41, 22-59
 - macros, comparison with, 2-3
 - nesting, 10-17
 - parameters, 10-2, 11-4
 - passing parameters, 10-2, 10-7, 11-3-13
 - position-independence, 10-2, 10-17
 - reducing execution time, 21-5
 - reducing memory usage, 21-2, 21-3
 - reentrancy, 10-2
 - relocatability, 10-2
 - specifying parameters, 11-14
 - storage allocation, 10-17
 - types, 10-2
- Successive approximation A/D converter, 13-44
- Summation example, 5-4-9, 18-10-11
- SWI, 4-2, 15-6, 15-10, 19-3-5, 21-3-4, 22-70-71.
 - See also* software interrupt instructions
 - vector, 15-10
- Switch and light system example;
 - error handling, 16-5-6
 - flowchart, 17-4-5
 - inputs, 16-4
 - modularization, 17-12
 - outputs, 16-5
 - problem definition, 16-4-6
 - structured program, 17-21
 - top-down design, 17-27-28
- Switch-based memory location example
 - error handling, 16-8
 - flowcharts, 17-5-6
 - inputs, 16-6, 16-8
 - modularization, 17-13
 - operating interaction, 16-8-9
 - outputs, 16-8
 - problem definition, 16-6-9
 - processing requirements, 16-8
 - structured program 17-21-22
 - top-down design, 17-28-29
- Switch bounce, 13-14-15
- SWI2, 22-70-71. *See also* software interrupt instructions
 - avoidance in packaged software, 22-71
 - vector, 15-10
- SWI3, 22-70-71. *See also* software interrupt instructions
 - vector, 15-10
- Switches, 13-12-20
- Symbol table, 2-7
- Symbols for flowcharting, 17-3
- SYNC, 15-6-7, 22-71
 - diagram, 15-7
- Synchronizing with I/O devices, 12-5, 12-8, 13-29-30
- Synchronizing with real-time clock, 15-23-24
- Synchronous I/O, 12-8
- Syntax, 1-10
- TAB, 22-72
- Table, lookup, 3-28-31, 4-8-11, 7-1, 7-4-6, 9-1, 9-3, 9-5, 12-13-14. *See also* lookup tables
- Table of squares example, 4-8-11
- TAP, 22-72
- TBA, 22-72
- TDRE flag in 6850 ACIA, 14-3-4, 14-6
- Teletypewriter data format, 13-49
- Teletypewriter interface, 13-48-53, 14-1-6
- Teletypewriter interrupt, 15-28-30
- Teletypewriter output routine
 - commenting, 18-5-7
 - self-documentation, 18-1-2
- Testing, 20-1-6
 - aids, 20-2
 - code conversion example, 20-1
 - data, selection of, 20-3
 - definition, IV-4
 - debugging, relationship with, 20-1
 - examples, 20-1, 20-4
 - review, 20-5
 - rules, 20-2-4
 - self-checking numbers example, 20-5
 - sorting example, 20-4
 - special cases, 20-3
 - structured testing, 20-2-3
 - test data, selection of, 20-3
 - tools, 20-2
- Testing bits, 13-13-14, 13-30-31, 15-8-9, 22-7
- Testing examples, 20-1, 20-4
- TFR, 22-72-73
 - direct page register, loading of, 7-7, 22-73
 - effects, 7-7
 - examples of use, 4-4, 7-7, 19-6
 - jump instruction, 22-73
 - loading direct page register, 7-7, 22-73
 - order of operands, 19-12
 - program counter, determination of current value, 10-17
 - register codes, 22-77
 - restrictions, 7-7, 22-73
 - uses, 22-73
- Time/generalizability tradeoffs, 11-3
- Time budgets for delay routines, 12-11-12
- Time constants, 12-11-12
- Time/memory tradeoffs, 4-10-11, 21-2-4
- Time-wasting routines, 12-9-12, 13-49, 15-23-27
- Timing loop, 12-9-23, 13-49
- Timing methods, 12-9-12
 - choice, 12-9
 - real-time clock, 15-23-27
- Top-down design, 17-26-31
 - advantages, 17-27
 - disadvantages, 17-27
 - procedure, 17-31
 - review, 17-31
 - stubs, 17-26-28
 - switch and light system, 17-27-28
 - switch-based memory loader, 17-28-29
 - verification terminal, 17-29-31

- TAP, 22-74
 Trace, 19-5
 Transmission errors, 12-8
 Transparent delay routine, 12-10
 Traps, 15-7. *See also* software interrupt instructions
 Trial subtraction in division, 8-9
 Trivial cases, 5-12, 9-3, 9-11, 19-12, 19-24, 20-1, 20-3
 Truncation of labels, 2-4, 3-46
 TST, 22-74-75
 interrupt clearing, 15-9, 15-22
 memory test, 5-10, 8-15
 6800 version, difference from, 3-41
 status checking, 13-11, 13-14, 19-5
 TSX, 22-75
 Two-byte operation codes, 3-10-11
 comparison instructions, 7-11, 22-29
 load instructions, 3-10-11, 5-6, 6-13, 10-16
 long conditional branches, 5-10
 minimizing use, 21-3
 prefix byte, 6-13
 store instructions, 10-16
 Two-dimensional arrays, 8-7-8
 Two-pass assembler, 2-16
 Twos complement, 4-16
 Twos complement overflow, 5-12. *See also* overflow flag
 TXS, 22-75
 U register. *See* stack pointer U
 UART, 13-52-53. *See also* 6551 ACIA, 6850 ACIA
 Unconditional branches, 5-14. *See also* BRA, JMP, LBRA instructions
 Unsigned numbers, comparison of, 4-5-6, 5-12
 Unsigned offsets in addressing. *See* ABX instruction
 Until-do structure, 17-16-17
 User stack in subroutine linkages, 22-59
 User stack pointer. *See* stack pointer U
 User's guide, 18-13
 V (Overflow) flag, 3-3, 3-5, 4-5, 5-12. *See also* overflow flag
 Validity checks on ASCII characters, 7-8
 Variable offsets in indexed addressing, 3-28-31.
 See also accumulator offset addressing mode, accumulator offset indirect addressing mode
 Varied length of instructions, 1-6, 2-11, 4-6
 Vector, 15-2, 15-10
 6809 table, 15-10
 Vectored interrupts, 15-2, 15-10
 Verification terminal example
 error handling, 16-12-13
 flowcharts, 17-5-10
 inputs, 16-11
 modularization, 17-13-14
 operator interaction, 16-12-13
 outputs, 16-11-12
 problem definition, 16-9-13
 processing requirements, 16-12
 structured program, 17-22-25
 top-down design, 17-29-31
 WAI, 22-75. *See also* CWAI, SYNC instructions
 While-do structure, 17-16, 17-17, 17-19
 Word-length (16-bit) data, 3-46, 3-47. *See also* FDB directive
 Write strobe from 6820 PIA, 13-7, 13-9, 13-26-29, 13-43
 X register. *See* index register X
 Y register. *See* index register Y
 Z (Zero) flag, 3-5
 BIT, effect of, 13-31
 carry recognition, 8-15, 13-19
 CMP, effect of, 4-5, 6-4, 19-12
 definition, 3-5
 equality checking, 4-5
 LD, effect of, 5-10, 9-3
 loop control, 5-5-6
 position in CCR, 3-3
 rounding, use in, 8-15
 ST, effect of, 9-3
 Zero offset indexed addressing mode, 3-21-22. *See also* constant offset indexed addressing mode
 Zero offset, omission of, 3-19, 3-22



ASSEMBLY LANGUAGE PROGRAMMING LANCE LEVENTHAL

While everyone's been talking about new 16-bit microprocessors, the 6809 has emerged as the important new device. Information included here will allow you to take full advantage of the 6809's unique design.

This is a comprehensive book. It covers 6809 assembly language programming in unmatched detail. The entire instruction set is presented and fully explained. The book contains many fully debugged, practical programming examples with solutions in both object code and source code. Discussion of assembler conventions, I/O devices, and interfacing methods is also included.

If you've never before programmed in assembly language, this book will teach you how. If you're an experienced programmer, you'll find this book an invaluable reference to the 6809 instruction set and programming techniques.

ISBN 0-931988-35-7



Parry Books
\$28.50

005.
265
MOT
7



6809 ASSEMBLY LANGUAGE PROGRAMMING

LEVENTHAL