```
 1                                  ******************************************************
 2                                  ** IMPROVED COPY TABLE PROGRAM FROM CHAPTER 2
 3                                  *
 4                                  ******************************************************
 5                                  *
 6                                  ******************************************************
 7                                  ** DATA SECTION
 8                                  ******************************************************
 9  0020                                    ORG $20
10  0020  C220              TABADR  FDB   $C220     ADDRESS OF FIRST TABLE
11  0022  20                SPACING FCB   $20       SPACING BETWEEN TABLES
12  0023  04                TABLEN  FCB   4         TABLE LENGTH
13  0024                    WRKCNT  RMB   1         WORKING COUNTER
14  0025                    WRKPNT  RMB   2         WORKING POINTER SAVE
15                                 *
16                                 ******************************************************
17                                 ** PROGRAM SECTION
18                                 ******************************************************
19  C100                                    ORG $C100
20                                 * INITIALIZE POINTER TO FIRST TABLE
21  C100  DE 20             START   LDX   TABADR
22  C102  DF 25                     STX   WRKPNT
23                                 * INITIALIZE WORKING COUNTER
24  C104  96 23                     LDAA  TABLEN    GET TABLE LENGTH
25  C106  97 24                     STAA  WRKCNT    STORE INTO WORKING COUNTER
26                                 * GET TABLE SPACING
27  C108  D6 22                     LDAB  SPACING
28                                 * WORKING COUNTER EQUALS ZERO?
29  C10A  7D 00 24          AGAIN   TST   WRKCNT
30  C10D  27 11                     BEQ   LAST      BRANCH ON YES
31                                 * COPY ONE ENTRY
32                                 *     GET POINTER TO FIRST TABLE
33  C10F  DE 25                       LDX   WRKPNT
34                                 *     GET ENTRY FROM FIRST TABLE
35  C111  A6 00                       LDAA  0,X
36                                 *     POINT TO SECOND TABLE
37  C113  3A                         ABX
38                                 *     PUT ENTRY INTO SECOND TABLE
39  C114  A7 00                       STAA  0,X
40                                 * COUNT DOWN THE WORKING COUNTER
41  C116  7A 00 24                   DEC   WRKCNT
42                                 * ADVANCE POINTER TO NEXT ENTRY
43  C119  DE 25                       LDX   WRKPNT
44  C11B  08                         INX
45  C11C  DF 25                       STX   WRKPNT
46  C11E  20 EA                       BRA   AGAIN     GO AROUND AGAIN
47                                 * STOP THE PROGRAM
48  C120  3F                LAST    SWI             "STOP" FOR MOTOROLA TRAINER
49  C121                            END

 Defined          Symbol Name          Value           References

    29     AGAIN                        C10A           46
    48     LAST                         C120           30
    11     SPACING                      0022           27
    21     START                        C100
    10     TABADR                       0020           21
    12     TABLEN                       0023           24
    13     WRKCNT                       0024           25     29     41
    14     WRKPNT                       0025           22     33     43     45

        Lines Assembled :  49           Assembly Errors :  0
```

Figure 3-11 Exercises 3-12 and 3-13.

Chapter 4

Program Structure and Design

Programming is a topic seldom discussed in engineering and microcomputer textbooks. Many authors assume that people automatically know how to program once they understand what computer instructions are. Such an assumption is unrealistic. Programming is a complicated process—a skill that is most easily developed with guidance from an experienced and thoughtful mentor.

Because this chapter discusses programs and programming, let's begin by defining the following words:

- *Program*—the sequence of instructions, and the associated data values, that the computer hardware uses to carry out an algorithm—the step-by-step procedure required to do something.
- *Programmer*—a person who creates new programs and modifies or maintains existing programs.
- *Programming*—the act of creating programs, which includes initial design and planning, documenting, coding into a language, testing, and debugging.

- *Software*—the programs and related information used by a computer. The word *software* is often used as a synonym for *program*.

These definitions provide us with some common ground for discussing programming. Don't confuse these definitions with job titles or department names. For example, the person employed as a programmer may have quite different responsibilities from those stated in the definition.

4.1 THE HARD COLD FACTS

Programming is usually taught by discussing the tools used to make programs. The writer assumes that the reader will see the light and understand why these tools are important. But often the light doesn't shine! So let's begin by looking at some issues related to programming before getting to the tools and techniques.

What Does Software Cost?

Software is expensive. Industry studies show that labor cost for program development is about one to one and a half hours per machine instruction. This work is performed by expensive skilled labor. Most useful programs require a minimum of several thousand instructions!

If you are a college student, you certainly believe that software is expensive. Think of the time and effort you put into your last program. You probably felt that the minor task that the program performed was not worth the effort!

Many on-the-job people have personal computers that they have never programmed to do a custom job. Even when advanced application programming languages are available, the effort required to write a custom program is not worthwhile. Usually, a program is purchased that is good enough to meet the needs. Also, a large software industry exists to supply people with software tailored to specific applications.

Don't be deluded by the highly sophisticated personal computer program that costs a few hundred dollars or less. Mass market programs selling at that price depend solely upon a large volume of sales to justify their existence. Such programs often cost many hundreds of thousands of dollars to develop. Marketing costs drive the total cost even higher.

What about Software Quality?

The quality of much software and most documentation is poor. Even after extensive testing, software frequently has bugs. Some people say that all programs have at least one bug.

Programmers use the term *bug* to describe a mistake because the word *mistake* devastates their egos. Bugs are not only costly to the supplier, but also to the customer, who relies on the software to do a job.

You might think that good testing and correction will overcome the problem of bugs. Experienced people say that *testing proves the presence of bugs, not the absence of bugs.*

Seldom does testing prove the reliability of software because not every possibility can be tested. Because testing has not been very successful, many companies now distribute preproduction copies of software to selected users. They hope to find the bugs before widespread distribution complicates the issue. Despite this effort, bugs frequently show up later.

Testing and debugging of software usually take more time than the original planning and writing. Consequently, most new software is completed later than the planned schedule. Software projects commonly require two to three times the estimated time that was allotted.

After software has been used for a while, changes are often desired or necessary. Changes can easily introduce new bugs, and the changes are often very difficult to make. Experience at modifying software has led some companies to the following rule: If more than five percent of a program must be changed, throw the program away and start over!

How could software quality be so troublesome? Besides the complexity of most software, poor quality is partly due to the way people learn about computers. They must start by learning what a computer is, what instructions are, and what programming is. These are all very detailed efforts that focus the person's attention away from the final goal. As the person learns more, the number of details increases. Unless people make a concerted effort to see the bigger picture, they can become trapped in the habit of seeing only details. The result is poor-quality software. Some people go through an entire career without making an effort to see the final goal.

Programming Is Hard Work

Some people like programming. Others don't! To those who like programming, it is the best expression of creativity. To those who dislike programming, it is tedious, boring work.

Regardless of your feelings about programming, it is a difficult and exacting job. Programs with mistakes usually are nearly worthless. The pressure to be correct is overwhelming. However, no magic will remove the work from programming.

Starting on the Right Path

Let's make your life easier. Programming can be made easier, and you can be more productive if you approach the task properly. Studies of programming and related activities suggest ways to make programming more cost effective and less work, and the programs produced more reliable. This chapter discusses many ideas that have been put into practice and have proved to be useful.

Trial and error is the least effective way to write software. Deliberate and careful design approaches using good tools are necessary for true long-term success that includes maintenance and enhancement of the software.

Two good software development techniques—called *structured programming* and *top/down design*—are widely accepted. They are the starting points for good software design and implementation. They are not the answer to all problems. They are starting points. This book will encourage the use of both of these techniques. All examples in later sections adhere to these principles.

Structured programming strongly influenced the design of most high-level programming languages developed since about 1970. Before then, people did not understand the programming problem. Since then, even the hardware architecture of computers has been influenced. Top/down design has also changed language design for the better. Many modern languages enforce or encourage these ideas in your programs.

Assembly language, as discussed earlier in this book, has total flexibility. Assembly language does nothing to enforce good programming. Only the discipline of the programmer enforces good programming. So the programmer must understand and embrace the techniques to gain the benefits that structured programming and top/down design promise.

4.2 PROGRAM DESIGN—WHAT'S IMPORTANT

Programming in any language is a human activity. Errors frustrate people, and successful working programs cause great joy. Building programs and running a computer system are usually enjoyable activities. Many people work long hours at their home computers as a hobby.

Contrary to this view is the pain of detailed design work that goes into complex software. Program coding requires very exacting work.

Some people who use hobby computers call themselves *hackers*. This name implies that they use trial and error to get their programs to work as they wish. Hackers usually do little planning or detailed design of their software.

Programmers' Goals

People usually have goals when they start to write a program. Assume that they are not programming as a hobby, but they have serious reasons to do the job. These people may be college students, working engineers, software developers, or consultants. *Programmer* is a useful name for such individuals, although the job title of programmer may imply quite different duties. Once again, the term *programmer* as used here means the person—usually persons—who designs and writes a program.

Programmers have similar goals as they set out to write programs. Different applications will affect the goals, so assume that microcomputer hardware as discussed in this book is adequate for the application. The list of possible goals is probably very long, so consider only the following five:

- *Write the shortest program.* The shortest program means the program that occupies the fewest bytes of memory and yet does the required task. The reasons for this goal are many. In product engineering or personal computer applications, the program length may be important because using less memory hardware reduces costs. College students may gain status with other students if their programs are shorter when doing the same assignment. Making the program fit existing resources could be useful if adding more memory is not practical. Many considerations including cost may limit memory size, especially in time-sharing applications.

- *Write the fastest program.* The fastest program is the one that runs in the shortest time while doing the required task. In product applications, such as automobile engine control, quick processing of complex algorithms is necessary to make the engine run correctly. Personal computers are being used for ever-larger applications, but they must quickly interact with the persons operating the computer. The fees for using some time-sharing computers are based on the run time of the program.

- *Write an easily understood program.* People, including the person writing the program, will need to understand the function of the instructions in the program. Understanding is necessary for changing the program, fixing bugs, or coping with changes in the hardware. Complex programs are more difficult to understand than simple programs. The program algorithm may affect the ease of understanding of a program, so the programmer may choose the algorithm accordingly.

- *Write an easily modified program.* Modifying a program means changing it to cope with changes in the environment. For example, a monitoring computer for several automatic manufacturing machines must monitor more machines if production must increase. The engine control computer for an six-cylinder engine will require changes if it is to control a four-cylinder engine. A personal computer program will require changes to use new features of an upgraded operating system. Each of these cases involves changes in an existing and functioning program. These examples clearly imply that an entirely new program is not necessary.

- *Meet the schedule.* Meeting a schedule is usually the goal of the programmer's supervisor or client, or of a college professor. The schedule specifies the date when the software user can do productive work with the program. A contract may dictate the schedule. Release of a new model of a product will enforce a deadline. Remember that microcomputers often are part of a product, and late release will give the competitor's products an advantage. To meet the schedule, people often forget all the other goals discussed and press on as fast as possible.

You probably can add several other goals to this list. What were your goals the last time you wrote a program?

What We Are Working With

The goals of programmers must be put into correct perspective in view of practical market conditions. The cost and performance of microcomputer hardware have changed greatly since microcomputers were first developed. Here are some accomplishments of the integrated circuit and computer manufacturers since the first hardware appeared:

- *Speed.* Microprocessor speed has increased more than a factor of 100.

- *Cost.* The cost of chips for equivalent performance has decreased more than a factor of 1000.

- *Memory size.* Usual memory sizes have increased from a few kilobytes to a few dozen megabytes.

- *Secondary storage.* Secondary storage has evolved beyond slow and small floppy disks with a capacity of a few hundred kilobytes. Hard and optical disks and CD ROMs (compact disk read only memory) have hundreds of megabytes to gigabytes of capacity and are much, much faster than the original floppy disks.

The enormous improvement in computer hardware has also increased its complexity. The complexity of software development has similarly increased. Sometimes additional computing power simplifies the programmer's job through more advanced software, but the complexity is still there.

In product engineering applications with embedded computers, the computer is part of a product and even the electronics will influence the programmer's task. Development software cannot easily hide the hardware complexity, although that is the intention.

The effect of increased complexity has been an ever-increasing cost of software development. Modification and long-term maintenance of software are large problems. The success of microcomputers has made enhancements a necessary marketing device. Competitors will be enhancing their products too!

Assessing the Goals Based on Reality

Let's evaluate each of the programmer's goals in view of the reality of the market and the technology available. Here is some discussion and some opinions about each goal:

- *Write the shortest program.* This goal emphasizes expensive human resources to reduce the cost of inexpensive hardware. Human resources are an expensive part of product development, and people write programs. Automated factories build computer hardware, so it is low in cost. Thus, this goal is of little practical value. Even if many copies of the program will be used, making deliberate efforts to write short programs is seldom practical.

  Effort toward writing short programs also encourages programming tricks to cut memory usage. Tricks usually lead to problems sometime in the lifetime of a product. In contrast, selecting a good algorithm that is particularly efficient in using memory would be practical.

- *Write the fastest program.* With the high and ever-increasing speed of microcomputer hardware, writing the fastest program is not an important goal for most applications. A big related problem, if speed is important, is knowing how to write the program for fast execution before writing begins. Similarly, determining what to optimize for speed after completing a program that is too slow is difficult. Usually, rewriting an entire program to optimize its speed is impractical anyway.

  The speed of a program must be determined under realistic control situations. One testing approach, called *profiling*, is to take data on the frequency of usage of various

parts of the program. You find out how often each part of the program runs under realistic circumstances. The usage data reveals which parts have a significant effect on the performance. You then optimize those significant parts. For example, doubling the speed of a part that accounts for one percent of the total processing time does little good. Doubling the speed of a part that accounts for 40 percent of the time makes a large overall improvement.

The realistic goal is to get a good program working, and then optimize the parts that will make a difference. This goal accomplishes overall cost-effectiveness. Deliberate effort to make a fast program from the beginning is usually not practical.

The execution speed of a program is almost independent of the length of the program. Long programs are not necessarily slower than short ones. For example, suppose that a short program is written as loops inside loops. An equivalent long program is a sequence of instructions without loops. The short program will certainly run slower than the long one.

- *Write an easily understood program.* The word *understood* implies understanding by people—including both the programmer and others. Better understanding results from simple straightforward problem solutions as opposed to complex and clever solutions.

  Never use programming tricks. For example, never use the op code of an instruction as the data value for another instruction. Clever tricks may slightly shorten or speed up a program. On the other hand, using expensive human resources to save inexpensive computer resources is not cost-effective. Therefore, an easily understood program without tricks is a principal goal of realistic programmers.

- *Write an easily modified program.* Modification of programs happens frequently both during development and after completion. Making a program easy to modify requires planning and effort during the design. However, this small effort is cost effective because it saves very large efforts when making the inevitable changes. Successful programmers focus on long-term goals and not just immediate success.

- *Meet the schedule.* Many people in the software industry accept late delivery of software as normal, because it happens so frequently. Users of personal computers have coined the term *vaporware* to describe advertised software that is not available for purchase. Good software delivered late is often not profitable.

  The best way to meet this goal is to organize carefully and plan for changes. Software written to ease understanding and modification is necessary. Forget about writing short or fast executing programs. Forget about clever programming tricks. Emphasizing the correct goals will get the job done as fast as possible.

The purpose of this section is to emphasize that you should write programs that are easy to understand and easy to modify. No significant effort should go into writing a program so that it is fast or short. This statement does not mean that silly things should be done. It means that little value is derived from spending serious effort to make short or fast programs.

This direction is somewhat contrary to human nature. Many people feel that much of the fun of programming is in finding clever solutions to programming problems. Instead, that creativity should be applied to bigger problems than to the minute details of a program.

4.3 PRACTICAL PROGRAMMING

A set of guidelines will emphasize the important goals in the creative process of programming. Guidelines cannot cover all circumstances and guarantee success—success requires good judgment. The ideas presented here are completely compatible with the goals of easily understood and easily modified programs:

- *Don't use a single resource for multiple purposes.* When an item is used for multiple purposes, compromises will usually be necessary in the use of that item. Instead, create a new resource for each independent purpose. Separate resources will sometimes require a slightly longer program, but the advantages are worth it. For example, don't use a loop counter to form addresses in a pointer. The two uses should be independent of each other; then a change in one use won't affect the other.
- *Use no intimate knowledge of the hardware configuration.* For example, do not use unimplemented bits of a register as default values; they may change later with hardware revisions. Similarly, moving program modules to new memory locations should have no effect on the correctness of the program.
- *Keep the instructions and data separated.* The data numbers should not be scattered between the instructions. Not only is understanding such a program difficult, but the program will have unnecessary instructions to branch over the data numbers.
- *Use tables.* Collect program parameters and related data together in tables or structures. If the program is designed to interact with a person, provide software that will easily modify the tables.
- *Only put constant data within the instructions.* To change data within instructions, you must have intimate understanding of the instructions in the program. Only when a change in the data also requires the program to be rewritten should data be put inside instructions. For example, immediate and indexed addressed instructions contain constants.
- *Avoid tricks.* Clever use of instructions usually leads to solutions that are not easy to enhance and alter. Tricks usually lead to greater cost.
- *Don't write self-modifying programs.* Programs that change their instructions as they run are very difficult to understand, debug, and modify. If the computer has permanent memory that can't change, as many microcomputers do, the program won't work at all if it must modify itself.

- *Use the instructions in the instruction set well.* The instruction set might be somewhat inconsistent, so you might prefer to avoid some instructions because they are rarely used. However, the design probably includes odd instructions because they make certain functions easy to do. Use all the available resources.

You should always actively strive to write good programs. Don't concentrate only on solving the problem at hand. You will find problem solving will be easier when you search for the simple, straightforward solution instead of the complex and elegant solution.

4.4 FLOWCHARTING

A tool to help organize the programmer's thoughts is valuable when designing and writing programs. Organization will aid in promoting the proper goals. Organizing your thinking with a design tool helps you to include details that you might miss otherwise.

Computer programs are algorithmic processes—they carry out large jobs one small step at a time. Most people understand algorithmic processes best when they are illustrated graphically. Several graphical tools exist, but the one most widely used is the flowchart.

A *flowchart* is a diagram made from several standard symbols connected by flow arrows. The flowchart symbols represent actions to be taken. The arrows direct the reader to follow the progress of the actions of the program.

The flowchart is both a documentation device and a design tool. At the design stage, the programmer usually sketches the flowchart on paper and changes it repeatedly as the design progresses. When the design and program coding are complete, the flowchart becomes a documentation device because it clearly illustrates what the program does and how it does it. Both the programmer and others use the flowchart to understand a program.

Flowchart Symbols

Flowcharts are made from several standard symbols. Let's consider a subset of the standard symbols and their meanings. Additional specialized symbols are available, but they often complicate the flowchart and add little in the way of clarity.

Primary flowchart symbols

The primary symbols are necessary for making most flowcharts. With them, you can make any flowchart. All other symbols are optional.

Flow lines and flow arrows. Flow lines with arrow heads guide the reader through the other symbols in the correct order. Several flow lines may come together at an intersection point, and then one line continues from that point as shown in Figure 4-1. Several flow arrows cannot leave an intersection point though, because the reader would not know which line to follow.

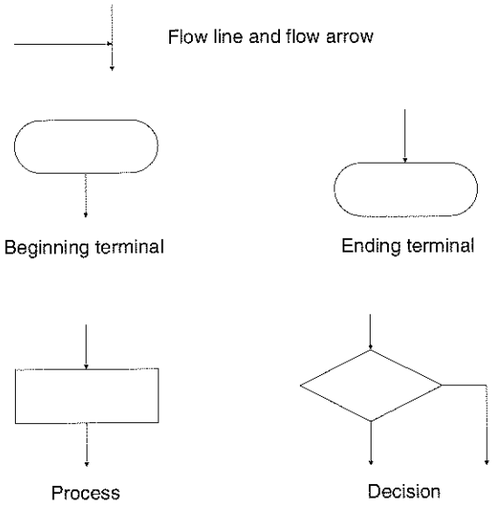

Figure 4-1 Primary flowchart symbols.

Only a single flow line should terminate on a symbol. If two or more flow lines must come together at a symbol, the flow lines should come together before the symbol with only a single flow line terminating at the symbol. This design leads to better clarity.

Terminal. The flowchart begins and ends with a unique symbol called a *terminal* as shown in Figure 4-1. The beginning terminal tells the reader where to start reading the flowchart. Put the name of the program module associated with this flowchart on the terminal symbol. The ending terminal indicates where the computer stops processing instructions in this program module. The label on the ending terminal is usually END or RETURN.

Process. The process symbol is a rectangular box that means do something or process some information. Figure 4-1 shows the process symbol. Write the name or description of the process inside the symbol. Most of the symbols in flowcharts are process symbols.

Each process box identifies a group of instructions that together carry out a particular function. This function has a distinct beginning point and a distinct ending point. Hence, the symbol has a single flow arrow entering it and a single flow arrow leaving it. This simple yet important observation, one flow arrow entering and one flow arrow leaving, is discussed further later.

Decision. The choice between two alternative flow paths is the essence of a decision. Figure 4-1 illustrates the diamond-shaped symbol for a decision. A short word description of the decision followed by a question mark identifies the symbol.

The computer makes binary choices, so the fundamental decision symbol has two flow arrows leaving it and one arrow coming to it. The paths going out usually have two opposite labels such as YES/NO, UP/DOWN, or TRUE/FALSE. These words answer the question inside the symbol. Decisions made up of more than two alternatives require several fundamental decisions.

Secondary flowchart symbols

The symbols included here are commonly encountered. However, their use is optional and usually unnecessary.

Connector. Two connectors containing the same identifier indicate the same point in a broken flow line—they connect two parts of a broken flow line. The connector symbol shown in Figure 4-2 is a small circle. Connectors usually contain a letter as an identifier. Each connector will have either one flow arrow coming into it or one flow arrow going out of it. Connectors are seldom used. You should avoid connectors whenever possible because reading broken flow lines is difficult.

Off-page connector. Figure 4-2 shows the off-page connector symbol—it connects two parts of a broken flow line that are on different sheets of paper. *The off-page connector is never needed and should never be used!* You will understand this very strong statement after reading the next two major sections.

Flowchart Example

A simple flowchart that includes all the primary symbols is shown in Figure 4-3. The symbols are not labeled, because the figure illustrates only the symbols and not a program. However, the logic of the algorithm can be understood even without labels.

Look at the figure carefully to see the individual flowchart symbols. Here are some characteristics of this flowchart that you should observe:

- The flowchart begins with a single terminator symbol.
- Each process box has a single flow arrow coming into it and a single flow arrow leaving it.
- The decision symbol has exactly two flow arrows leaving it.
- When the two flow arrows come together, they meet before the ending terminator so that only one arrow meets the ending symbol.

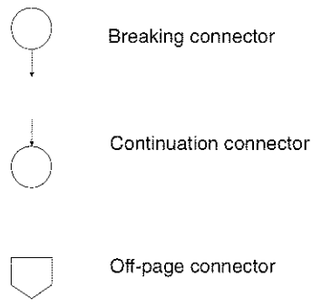

Figure 4-2 Secondary flowchart symbols.

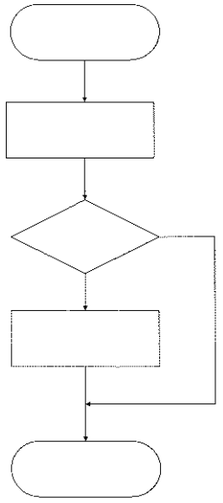

Figure 4-3 An example flowchart.

- The arrow heads clearly show the flow of the algorithm represented by the flowchart.
- The flowchart has a single ending terminator.

Using the flowchart symbols as described here for all flowcharts is good practice. Most of these characteristics should seem simple and obvious. However, when an algorithm becomes very complex, and hence its flowchart becomes complex, forgetting these simple rules is easy.

4.5 STRUCTURED PROGRAMMING

One requirement of good programming is simplicity. If programs are very complex, they are difficult to write, to modify, and to understand.

Programs consist of building blocks that flowchart symbols represent. Fortunately, we need only three symbols to make flowcharts: the terminal, the process block, and the decision. This concept seems simple enough; however, connecting these building blocks with flow arrows can be done in many ways.

Undisciplined programmers write *spaghetti* programs. The flowchart for a spaghetti program has the flow lines and boxes entangled in ways to make changes and understanding almost impossible. Figure 4-4 illustrates such a flowchart. Try to follow all the possible paths through the flowchart. Some processes have three or four different paths leading to them.

The effort to cope with spaghetti programs led to research to determine whether fundamental program building blocks exist. The research revealed only three fundamental ways of connecting the flowchart symbols. These three ways of connecting the symbols, called

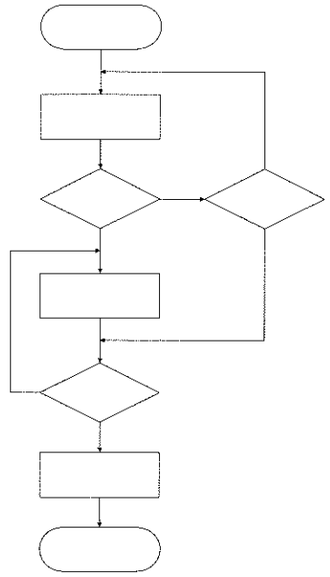

Figure 4-4 The flowchart for a spaghetti program.

program structures, are enough to build all possible programs. The discipline now called *structured programming* was the result of the research. The purest form of structured programming requires the building of programs from the three fundamental program structures only.

Experience has shown that structured programming leads to better quality software and to more cost-effective software. Of course, programs are only correctly structured if programmers make them structured, or the computer language enforces structuring. A programmer cannot make an unstructured program look structured on a flowchart. The programmer must design a structured program and then write the program by following the design. A programmer cannot write a spaghetti program, get it to work by trial and error, and then make a structured flowchart for it.

Structured programming is just one step in designing good programs and does not guarantee high-quality programs. It is just one necessary ingredient in successful software engineering.

Fundamental Program Structures

The names of the three fundamental program structures are SEQUENCE, IF-THEN-ELSE and DO-WHILE. Look at the three fundamental structures as represented by the flowchart in Figures 4-5 through 4-7. Observe that each structure has a single beginning or entry point and a single exit point. Similarly, each process box on a flowchart has a single entry point and a single exit point. Never draw any additional flow arrows entering or leaving a process box.

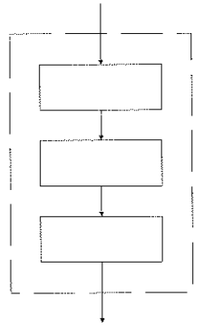

Figure 4-5 The SEQUENCE structure.

Each structure is itself a process—drawing a rectangular process box around each structure is correct. The fundamental program structure is the basic building block of programs. Structured programming advocates building programs from only these fundamental structures. The phrase *correctly structured program* describes a program built only from fundamental structures.

SEQUENCE structure

The *SEQUENCE structure* is simply several process boxes strung together one after another. Figure 4-5 shows that this series of boxes begins at one point and ends at one point. Remember this structure by saying that it does one thing after another in a sequence.

IF-THEN-ELSE structure

Figure 4-6 illustrates the *binary decision structure* called *IF-THEN-ELSE*. The decision chooses between two alternative processes. The flow arrows must merge at one point after the processes on the two sides. If these two sides do not come back together, you get spaghetti because control is transferred outside the structure.

A common form of this structure has a process that does nothing on one side. The effect is a structure, sometimes called *IF-THEN*, that does something or bypasses it.

Remember the IF-THEN-ELSE structure by saying that it tests a condition, and IF the condition is satisfied, THEN do something, otherwise do something ELSE.

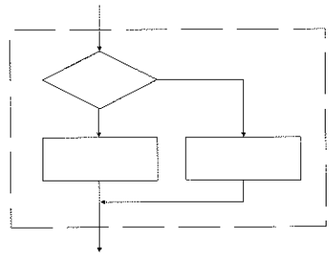

Figure 4-6 The IF-THEN-ELSE structure.

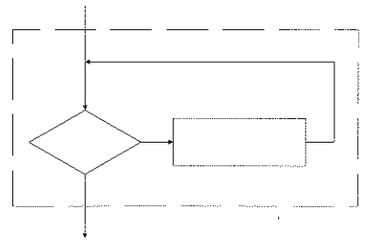

Figure 4-7 The DO-WHILE structure.

DO-WHILE structure

Figure 4-7 illustrates the iterative or *loop structure* called *DO-WHILE*. The internal process or body is repeatedly executed until the decision determines that the process should not execute again. That is, it loops until the decision exits the loop.

You can use any kind of condition for the decision. The decision may be based on a number. Alternatively, it could depend upon a complex series of input/output operations. Usually, the body of the loop will do something to alter the condition so that the loop exits eventually. If the exit never occurs, the loop is an *infinite loop*, meaning that it never ends.

If you are familiar with other kinds of loops, you can easily distinguish the DO-WHILE loop because the DO-WHILE has the decision *before* the body of the loop. The decision may cause the loop to exit without the body of the loop executing even once.

Remember the DO-WHILE structure by saying that it will test for a condition, and then DO something over and over WHILE the condition exists.

Extended Program Structures

A few other structures, called *extended structures*, meet the general requirements of structured programming. A variety of names have been created for these extra structures. You should look for the fundamental characteristics of structures of other names because they may be the same structures you already know.

Many high-level languages have some extended structures. People also write assembly language versions, although these extra structures are not necessary—the three fundamental structures will carry out all tasks. However, extra structures are sometimes more convenient for the programmer or for the language compiler.

DO-UNTIL structure

The *DO-UNTIL loop* structure shown in Figure 4-8 is similar to the DO-WHILE. The main difference is that the decision to exit the loop is placed after the body of the loop rather than before.

A serious limitation of the DO-UNTIL is that the body of the loop always executes at least once. Because the body executes before the decision, the decision cannot prevent the execution of the body at least once.

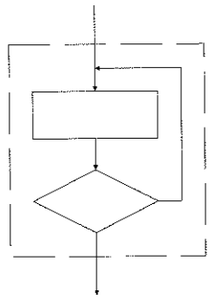

Figure 4-8 The DO-UNTIL structure.

Practical programs often need to execute the body of a loop zero times. So the DO-UNTIL loop is impractical as a general solution. The DO-WHILE loop has no such limitation; it is more flexible and adequate for all programs.

The DO-UNTIL loop can be troublesome if the programmer is not careful. If the loop decrements a numeric counter down to zero to end the loop, an initial counter value of zero causes problems. Probably the loop will execute the maximum number of times that the counter can specify—decrementing a zero value, if unsigned numbers are used, results in the largest possible value. The problem happens because the loop decrements the counter before the decision.

The usual solution to this problem is another decision before the loop that checks for this special case. The decision bypasses the loop to avoid the problem of an initial zero value. This decision adds extra code to the program that is unnecessary and confusing.

The DO-UNTIL loop does reflect normal thought to some extent. That is, people usually think about doing something first and then about making a decision based on the results. By contrast, experienced programmers think of a loop structure and how to get out of the loop. So the DO-WHILE loop seems normal to experienced programmers and odd to others.

IN-CASE-OF structure

Figure 4-9 shows the multiple decision structure with more than two alternatives called *IN-CASE-OF*. The first distinguishing feature of this structure is that one alternative for each decision is to do nothing. The second feature is that the program exits the structure after a single test is satisfied and the corresponding process is executed.

The IN-CASE-OF structure can be made from fundamental structures. Figure 4-10 shows how multiple IF-THEN-ELSE structures are made. A table lookup technique for making the IN-CASE-OF structure is also possible.

The IN-CASE-OF structure is just a program module that has a name because it occurs so often in practical programs. It does meet the basic requirement of a structure in that it has one beginning and one ending point.

Some high-level programming languages have a CASE structure to make programming a little easier. Also, the language compiler may be able to optimize the machine language for fast execution.

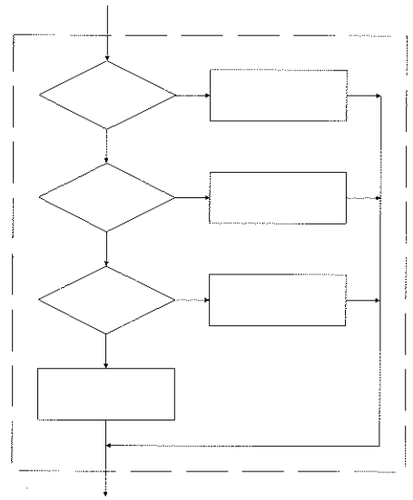

Figure 4-9 The IN-CASE-OF structure.

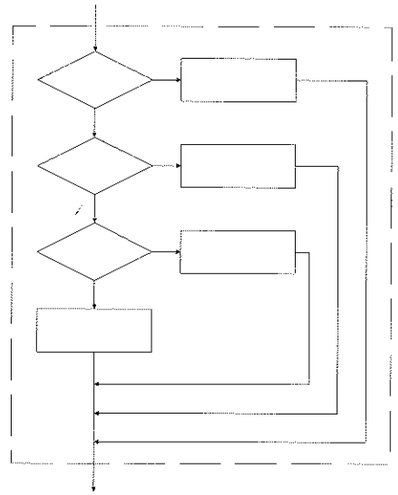

Figure 4-10 The IN-CASE-OF structure made from IF-THEN-ELSE.

Identifying Structured and Unstructured Programs

The flowchart shown in Figure 4-11 represents a structured program. Dashed lines enclose the fundamental structures. The design uses structures inside structures to build the overall flowchart and, eventually, the whole program. Notice that you can study the structure of the program without even knowing the names in the symbols.

To check a flowchart for correct structure, begin by drawing boxes over the flowchart to enclose program structures at all reasonable places. If all the boxes have only a single entry point and a single exit point, the flowchart meets a major requirement of structured programming.

Enclosing a collection of spaghetti with a box and then claiming to have a structured program is not legitimate! On the other hand, if each box contains only fundamental structures, you have met all the requirements.

Look at the unstructured spaghetti flowchart in Figure 4-4. Try to draw boxes on the flowchart that enclose program structures that each have a single entry point and a single exit point. Your boxes will reveal that the program modules have multiple entry and exit points.

Programs represented by unstructured flow charts usually contain unnecessary branch or jump instructions. Therefore, you can usually identify an unstructured program from its listing by looking at the use of the branch and jump instructions.

Some high-level programming languages have a branch statement called *GOTO*. Some companies encourage their employees to use *GOTO-less* programming to avoid jumping around in the program. Their goal is to encourage structured programming. However, those

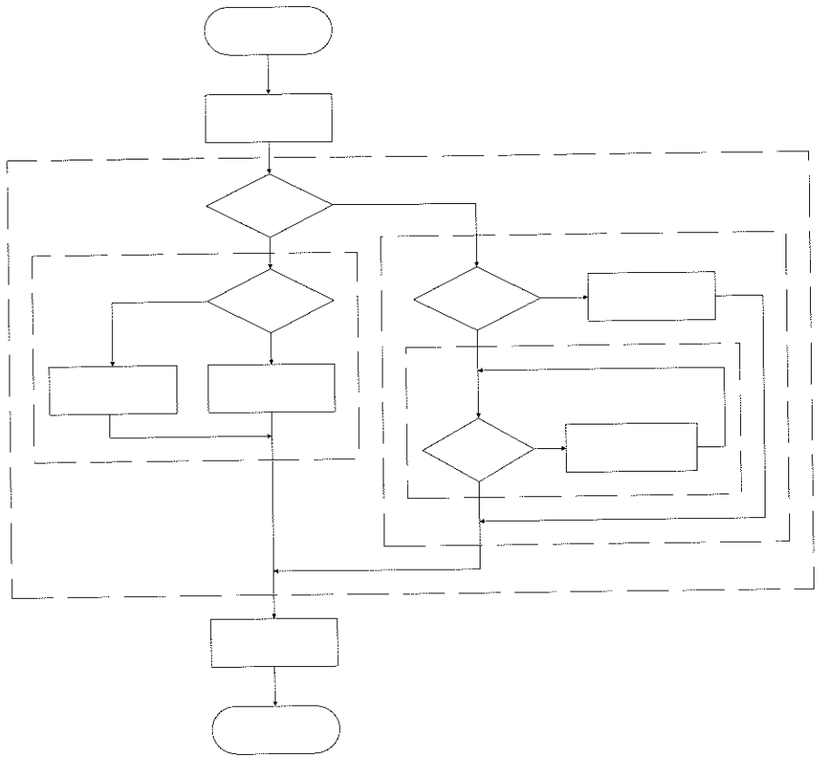

Figure 4-11 A structured flowchart made from fundamental structures.

languages with a GOTO usually require the GOTO statement to make all the fundamental structures. In contrast, other high-level programming languages lack the equivalent of a GOTO statement eliminating any chance for unstructured programs.

Making Structured Loops

If you are unaccustomed to structured programming, making structured DO-WHILE loops may seem difficult. As an example, the loop in Figure 4-12 is clearly unstructured. The loop has two exit points. The problem is caused by the need to terminate the loop prematurely when the program detects an error. The decision in the example sends control out of the loop, effectively making a second exit point.

The alternative flowchart in Figure 4-13 easily handles the premature termination using a structured design. The technique requires that the termination condition for the loop be forced. Then the loop follows its normal course to the end and remains correctly structured.

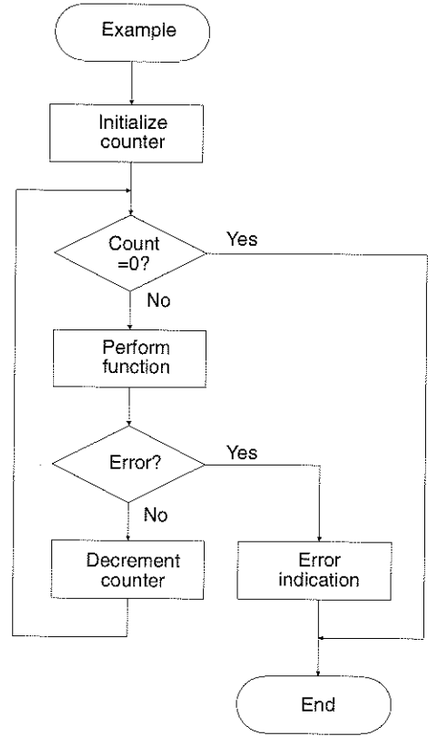

Figure 4-12 Unstructured premature loop termination.

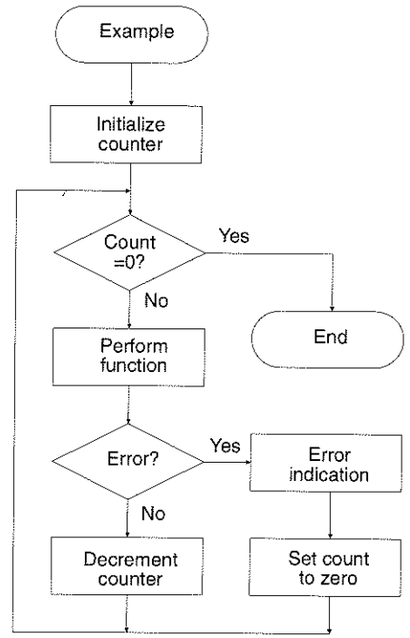

Figure 4-13 Structured premature loop termination.

Complicated loops may need to exit due to several different conditions. These loops are easily implemented in fundamental structures by using a flag that determines whether the loop will exit. Outside the loop, the flag is set to an initial value that enables the loop to execute. Then, inside the loop, various program conditions alter the flag. The loop exits based on the value of the flag.

Sometimes, the value of the flag that causes the loop to terminate is used outside the loop. Figure 4-14 illustrates using such a flag to terminate a loop. After the loop in the figure exits, the flag has the value one or two depending on the condition that caused the exit.

A Troublesome Case

One problem often disturbs people when they begin using structured programming. The problem is an error that aborts the program. Aborting a program means to prevent it from going to its end. Figure 4-15 illustrates the case of a program that detects an error that prevents it from continuing.

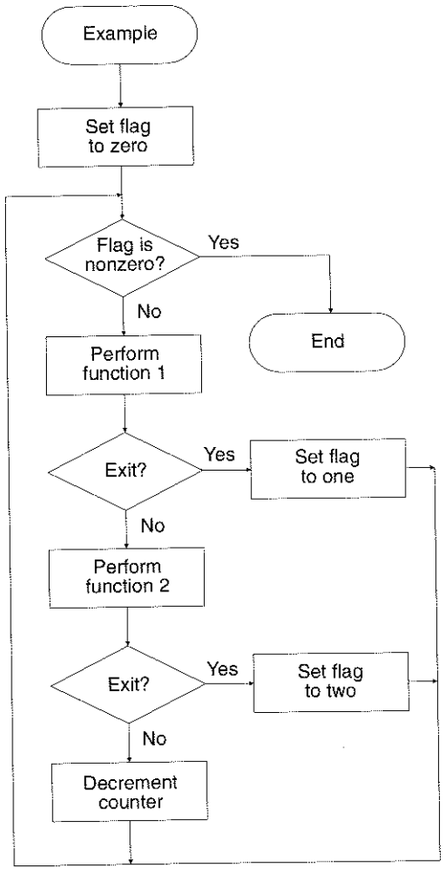

Figure 4-14 Using a flag to terminate a loop.

Aborting the program technically is a violation of structured programming because the program does not come to its normal end. However, aborting is quite practical to do, and it causes no problems. Specifically, aborting the program means to leave this execution of the program at some point and not return to continue from that point under any circumstance. Consequently, aborting the program prevents it from reaching the normal end of the program.

If a program is run again after being aborted, the new execution starts at the beginning of the program. It never resumes from the point where the program was aborted.

Starting again at the beginning is the reason that aborting a program is allowed within structured programming. Regardless of structuring, the data required by the program may be damaged by running only part of the program. However, running the entire program from the beginning will initialize the program and regenerate any data needed.

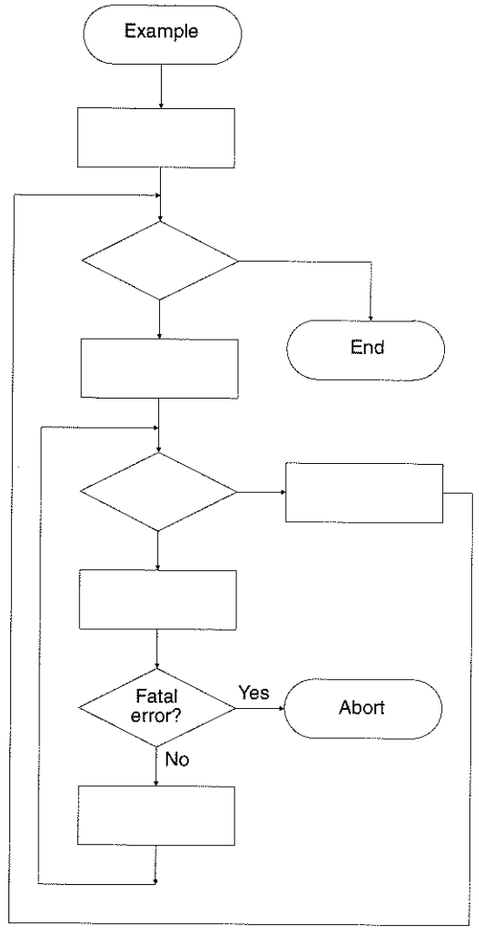

Figure 4-15 A troublesome case for structured programming.

Will You Use It?

If you have experience at writing spaghetti programs, you may decide that creating a structured program is unnecessary and nearly impossible. You will feel very limited by the requirements, which may at times seem artificial. If you persevere, you will learn that structured programming is neither impossible nor confining.

You will often discover that an unstructured program resulted from poor understanding of the problem. Experience shows that forcing a design into proper structure also forces the programmer to think carefully about the problem. When the design meets the requirements of structured programming, other problems are often solved. Thus, structuring is not just a programming tool, but it is also an aid to problem solving.

Sometimes people feel that making a structured flowchart limits what can be done in the program code. However, if you cannot solve the problem in the flowchart, you cannot solve the problem in the code.

Making Unstructured Programs Structured

Making an unstructured program correctly structured by simple changes is almost impossible. Generally, you must write an entirely new program. The logic of the unstructured program must be understood to design the structured equivalent. Therefore, illustrating an unstructured flowchart without labels and then making its equivalent is not possible. Instead, all the detail of the unstructured program must be understood.

Clearly, the only practical approach to structured programs is to design a structured program from the beginning. Then as changes are made, the structuring must be preserved. Experience will show you that this design is easier than it may seem. When a program is written in correctly structured form, changes are easier to make.

4.6 TOP/DOWN DESIGN

Top/down design is an old and simple idea that has a new name, because it is now applied to computer program design. Top/down design simply says to get the big picture first, and then consider finer and finer details as the design proceeds. The basic idea is to approach the design of a program by first identifying the major functional parts, or modules, of the program. This collection of major modules is called the *top level* of the design. Next, each of these parts is further broken into smaller parts, which together are called the *second level.* The second-level design is equivalent to the top-level module that it expands. The design continues until enough detail exists to write and document the program.

Detail is added to the design by adding more modules at lower levels. Altering the top-level modules to make more modules is not correct. Doing so would make too many modules, causing the designer to lose sight of the overall design. The result probably would be a spaghetti program.

For example, some second-level modules may be quite complex, so these modules are broken into parts on a lower third level. Again, the parts of the second level are not replaced, but each module is separately broken into parts.

Writing teachers recommend this approach for writing an English composition—another form of software. To write a composition, make an outline with the major sections identified first. Next, add subsections to the outline; the subsections are each equivalent to the major section that they are outlining. This expansion to additional detailed subsections continues until a complete outline results.

The table of contents of a book is an example of an outline of a written document. The design of the book you are reading was done this way. I decided upon the chapter contents by making the table of contents first. Then, I determined the sections of each chapter. As I wrote, smaller subsections were designed and added to a detailed table of contents, which

served as an outline. The table of contents in the book contains only the upper levels of the detailed table of contents. Some changes in the design were made, but the major modules remained the same as in the original design.

Choosing Program Modules

You must decide how to break the design of a program into practical modules. A programmer must use judgment and experience to do this successfully. You must choose functional parts of the program to be modules. Each module should do something that is immediately recognizable by someone looking at the design. Only related functions should be together in a module.

For example, a program may gather some data and then print a report. The jobs of gathering the data and printing are two separate functions. Probably, these should be thought of separately in tackling the design. You will consider details such as scaling data values for unit conversions. These details can be thought about later, after all the top-level design is finished.

Top/Down Design Using Flowcharts

Figure 4-16 illustrates the top/down approach to computer program design using flowcharts. The first step in designing a program is making a top-level flowchart that will segment the total program into its major functional sections. Next, make a second-level flowchart for each process block that requires additional design work. You continue creating additional levels for complex modules. Your judgment determines when you have enough levels. Your creativity gets expressed and your talent is demonstrated when you make judgments about the design of the program.

The lower-level flowcharts are equivalent to the higher-level modules they represent. That is, the process blocks each have a single beginning point and a single ending point. Therefore, the expanded flowchart for a block also must have a single beginning point and a single ending point.

Relationship to structured programming

The requirement for a single beginning and single ending point is the same as one requirement of structured programming. Thus, structured programming is perfectly compatible with top/down design. Top/down design requires structured programming.

Number of levels

You must determine the number of levels required. Because this process is a creative design process, we can rely only on practical experience to provide guidelines or rules. However, some justification exists for the rules.

Maximum number of levels. The first purpose of a flowchart is to display the function performed by each section of the program. The names placed on the symbols must

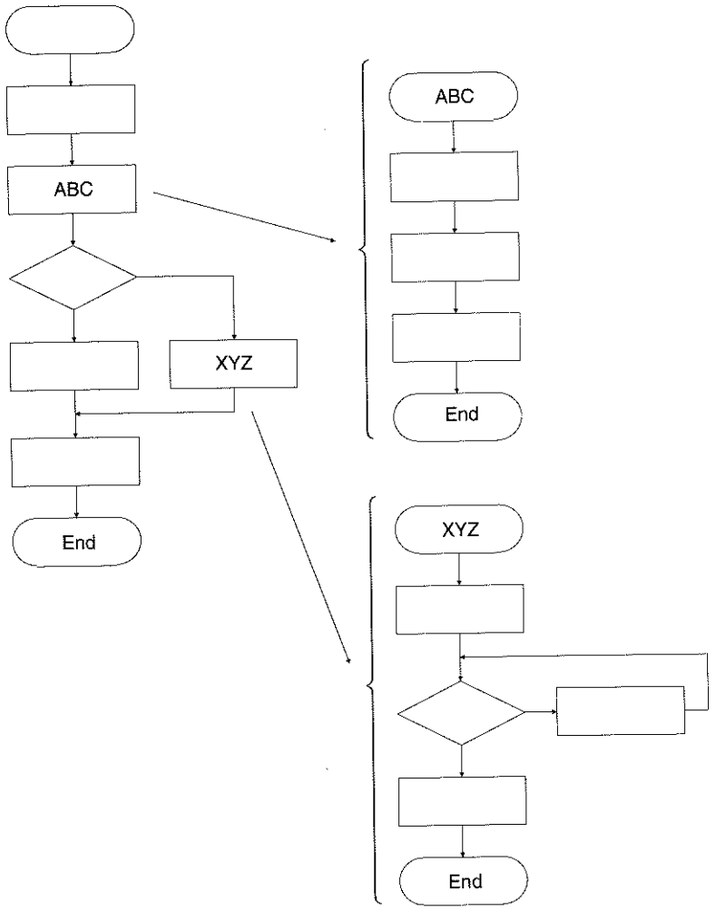

Figure 4-16 Top/down design using flowcharts.

describe these functions. The programming language statements will not appear on the flowcharts, because the language statements comprise the program implementation—not the program function. A listing of the language statements documents the program.

In other words, the flowchart should not have language statements on its symbols. Not placing language statements on the flowchart provides an effective maximum limit on the number of levels of flowcharts. Whenever the only choice is to put language statements on the flowchart, that flowchart is unnecessary.

Similarly, the flowcharts will be independent of the computer architecture and the hardware. Don't put hardware details on the flowchart.

Minimum number of levels. At the other extreme, the minimum number of levels depends on the size of each flowchart. If your flowcharts have only a few levels, you may need to make large flowcharts to provide enough detail. However, large flowcharts destroy most of the advantages of top/down design. Each flowchart should be simple enough that you understand its entire function at a glance without extensive study. Most people can understand a flowchart after a quick glance if the flowchart has no more than ten symbols.

Therefore, a rule of thumb is to limit flowcharts to ten symbols. A ten-symbol flowchart is practical because it will easily fit on a single 8 1/2 by 11-inch sheet of paper, making the documentation convenient. Furthermore, *you will never need the off-page connector symbol*, because your flowcharts will each fit on one page. An earlier section encouraged you never to use the off-page connector because your flowcharts should fit on one page.

Top/Down Design Summary

Here is a summary of the guidelines for top/down design using flowcharts:

- Make a top-level flowchart first, then make second-level flowcharts, and then other levels if needed. Then start coding. Never start writing code without designing this way first.
- Run a working program as soon as possible to check the design. Don't be concerned if incomplete or dummy modules make the function of the program nonsense.
- Design each program module using structured programming, and then code them in a structured fashion according to the flowchart.
- Put comments in the program that match the flowcharts. Put the comments in the source as you create the source module, not after it has been assembled and tested.
- Use some commenting technique to show the program structure in the listing.
- Put functional names on the flowchart symbols. Don't make any flowcharts so detailed that only language statements can be put on the flowchart.
- Make flowcharts that are independent of the computer architecture and hardware whenever possible. For example, the flowcharts should contain no mention of addressing modes or microprocessor registers.
- Limit the maximum size of each flowchart to ten symbols. This guideline limits the complexity of the flowchart, and the flowchart will fit on one sheet of paper.
- Design each flowchart to meet the requirements of structured programming.

These guidelines will help you write good programs, but they do not guarantee success. Only careful and diligent work by a knowledgeable person will result in high-quality software.

4.7 STRUCTURED TOP/DOWN ASSEMBLY LANGUAGE

Assembly language programmers have total flexibility in writing and documenting their programs. The quality of their work varies from excellent to worthless. The following two examples illustrate these extremes. The comments will help you evaluate the ideas presented. Do not assume that the approaches used in the good program are the only good ways to write and document programs. Instead, form your own opinion and use your creativity to make improvements.

A Good Program

The sample program that follows provides an illustration of the techniques of structured programming and top/down design. First, structured flowcharts demonstrate top/down format even though the program is very short and simple. Next, the assembly language program listing illustrates the use of many instructions, addressing modes, and programming techniques. The documentation on the listing is adequate for most purposes, and it is based on the assumption that the reader has knowledge of the Motorola assembly language.

Example program specifications

The following program searches a table in memory for negative numbers. It both counts and sums these negative numbers and then indicates whether all numbers in the table are negative. The table may represent the inventory of certain parts, and the negative numbers are orders that could not be filled.

The table is specified by the address of its first entry (lowest address) and the length of the table. The table contains single-byte signed numbers. The length (unsigned), negative count, and sum values are double-byte numbers to allow for large tables and large sums. The program does not test for two's complement overflows.

Flowcharts for the example

Flowcharts for this example are shown in Figure 4-17. Though this program is very simple, the top/down design idea is useful. Each flowchart defines a specific function that is independent of the other parts. The title of each flowchart matches exactly the title of the higher-level process that is being expanded.

Documentation on the assembler listing

Documenting the program structure in the listing is very useful. The programmer should use some creativity in deciding upon a format. The format used in the example is only one possibility. It was chosen primarily to make the listing match the flowcharts closely. Here are the details of the format used in the listing:

- *Major sections of the listing.* The title block, data section, and program section of the listing are labeled with headings surrounded by lines of asterisks. The asterisks

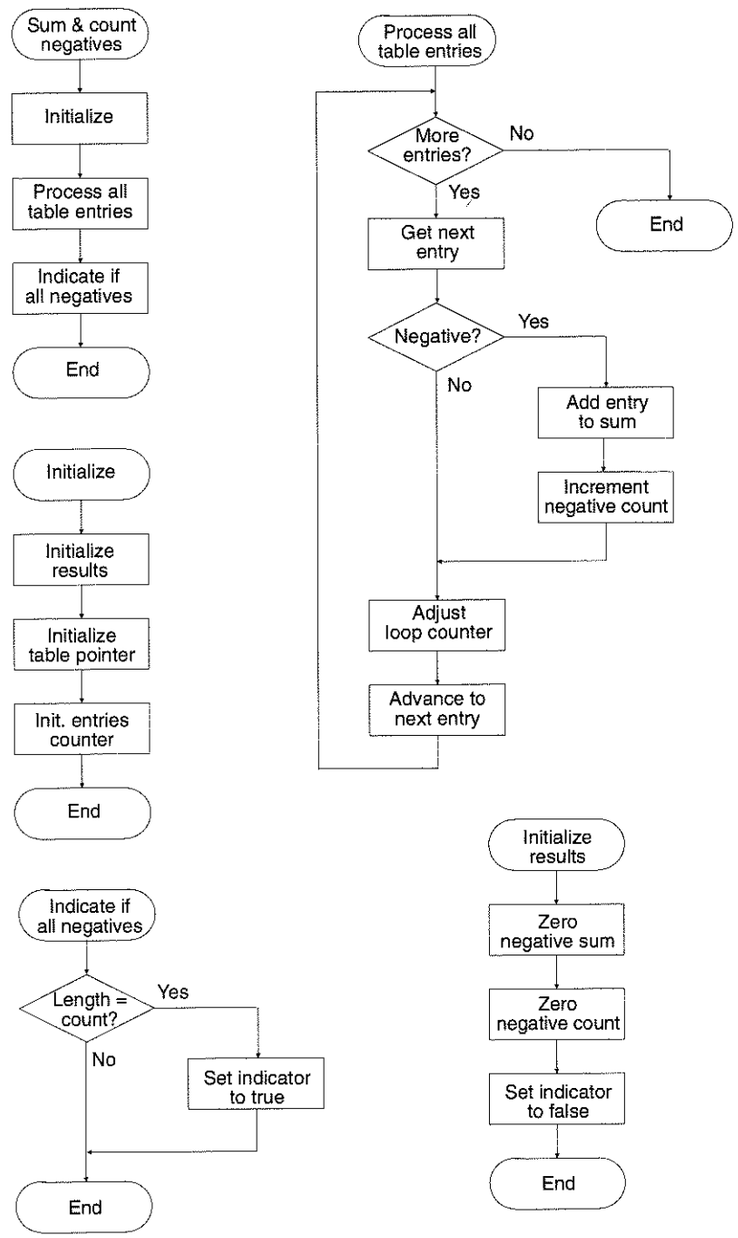

Figure 4-17 Top/down flowcharts.

make the headings easy to find. Printing each of these sections on separate pages of the listing is also common.

- *Top-level format.* The top-level flowchart titles are surrounded by dashed lines as Figure 4-18 shows on lines 36 through 38. The titles on the flowchart must closely match those on the listing. All in-line programming for this level should be between its title and the title of the next section on the listing that is at the same level. In other words, the program should not jump around on the listing with logical parts of this section within other sections. All the code for the Initialization module should be on the listing before the title for the Process All Table Entries section.

- *Second-level format.* The titles of the second-level flowcharts have headings without dashed lines as Figure 4-18 shows on lines 39, 47, and 49. These comments mark off the second-level modules. Although the headings for the second level are less prominent than those for the top level, you can easily see that the second-level modules are contained within the top-level module.

- *Third-level format.* Indenting lower-level comments is common practice as Figure 4-18 shows on lines 40, 43, and 45. Indenting makes these comments less prominent on the listing and implies that these modules are within the second-level module.

- *Implementation specific comments.* The last set of comments is at the right of the instructions as Figure 4-18 shows on lines 63 and 64. These comments are not flowchart titles; instead, they explain details of how the instructions were used in writing the program. Sometimes, such comments include specifics of a particular computer architecture and the particular programming approach used. This information is not on the flowchart, because the flowchart describes only the function of the program. Notice that the comment on line 63 was continued on line 64 with two periods that approximate an ellipsis to imply the continuation. The area to the right of instructions is also the perfect place to explain tricky uses of instructions. The explanation is on the line with the instruction, making it easy to understand.

Documentation is an area open to the creativity of the programmer. Some companies attempt to enforce consistency among programmers by mandating the format of the listing.

Editing comments

Because programs change frequently, the comments also will change frequently. You should plan for changes in your comments. The approach used in the example makes these changes easy. For example, changes often require moving sections of the program to new locations. An editor program can easily move a block of lines. If the comments are formatted as shown here, the instructions can be changed or moved without changing the comments. If a change requires that an instruction with a detailed comment at the right be moved, the instruction and its comment can be moved together easily. Not moving the specific comments with the instruction is difficult!


```
 1                          ********************************************************
 2                          ** SUM AND COUNT NEGATIVE TABLE ENTRIES
 3                          ** INDICATE IF ALL NEGATIVE
 4                          *
 5                          * THIS PROGRAM SUMS THE NEGATIVE ENTRIES IN A TABLE,
 6                          * COUNTS THE NUMBER OF NEGATIVES, AND INDICATES IF
 7                          * ALL ENTRIES ARE NEGATIVE.
 8                          *
 9                          * THE TABLE IS SPECIFIED BY ITS ADDRESS AND LENGTH.
10                          * TABLE ENTRIES AND INDICATOR ARE SINGLE-BYTES. THE
11                          * LENGTH, SUM, AND NEGATIVE COUNT ARE DOUBLE-BYTES.
12                          * TWO'S COMPLEMENT OVERFLOW CHECKING IS NOT DONE.
13                          *
14                          * AUTHOR: GENE H. MILLER
15                          *
16                          * REVISION HISTORY: VER2.1  5/28/1998
17                          *
18                          ********************************************************
19                          *
20                          *
21                          ********************************************************
22                          ** DATA SECTION
23                          ********************************************************
24  0010                            ORG $10
25  0010                    TABADR  RMB   2     PUT ADDRESS OF TABLE HERE
26  0012                    TABLEN  RMB   2     NUMBER OF TABLE ENTRIES
27  0014                    NEGSUM  RMB   2     PROGRAM PUTS SUM HERE
28  0016                    NEGCNT  RMB   2     PROGRAM PUTS COUNT HERE
29  0018                    ALLNEG  RMB   1     00=FALSE, FF=TRUE
30  0019                    WRKCNT  RMB   2     WORKING LOOP COUNTER
31                          *
32                          ********************************************************
33                          ** PROGRAM SECTION
34                          ********************************************************
35  C100                            ORG $C100
36                          *-------------------------------------------------------
37                          * INITIALIZE
38                          *-------------------------------------------------------
39                          * INITIALIZE RESULTS
40                          *     ZERO NEGATIVE SUM
41  C100  CC 00 00          START   LDD   #0000
42  C103  DD 14                     STD   NEGSUM
43                          *     ZERO NEGATIVE COUNT
44  C105  DD 16                     STD   NEGCNT
45                          *     SET INDICATOR TO FALSE
46  C107  7F 00 18                  CLR   ALLNEG
47                          * INITIALIZE POINTER TO TABLE BEGINNING
48  C10A  DE 10                     LDX   TABADR
49                          * INITIALIZE NUMBER OF ENTRIES COUNTER
50  C10C  DC 12                     LDD   TABLEN
51  C10E  DD 19                     STD   WRKCNT
52                          *-------------------------------------------------------
53                          * PROCESS ALL TABLE ENTRIES
54                          *-------------------------------------------------------
55                          * MORE ENTRIES?
56  C110  DC 19             LOOP    LDD   WRKCNT    TEST WORKING LOOP COUNTER
57  C112  27 1B                     BEQ   NEXT      BRANCH ON NO
58                          * GET NEXT ENTRY
59  C114  E6 00                     LDAB  0,X
60                          * NEGATIVE?
61  C116  2A 0D                     BPL   POSITV    BRANCH ON NO
```

Figure 4-18 Assembly language listing for a good program.

```
62                                  * ADD NEGATIVE ENTRY TO SUM
63   C118   4F                             CLRA            EXTEND SINGLE-BYTE NEGATIVE
64   C119   43                             COMA            ..TO DOUBLE-BYTE NEGATIVE
65   C11A   D3 14                           ADDD   NEGSUM
66   C11C   DD 14                           STD    NEGSUM
67                                  * INCREMENT NEGATIVE COUNTER
68   C11E   DC 16                           LDD    NEGCNT
69   C120   C3 00 01                        ADDD   #1
70   C123   DD 16                           STD    NEGCNT
71                                  * ADJUST LOOP COUNTER
72   C125   DC 19                   POSITV  LDD    WRKCNT
73   C127   83 00 01                        SUBD   #1
74   C12A   DD 19                           STD    WRKCNT
75                                  * ADVANCE TO NEXT ENTRY
76   C12C   08                              INX
77   C12D   20 E1                           BRA    LOOP
78                                  *-------------------------------------------------------
79                                  * INDICATE IF ALL NEGATIVES
80                                  *-------------------------------------------------------
81                                  * LENGTH = COUNT?
82   C12F   DC 12                   NEXT    LDD    TABLEN
83   C131   93 16                           SUBD   NEGCNT
84   C133   26 04                           BNE    LAST     BRANCH ON NO
85                                  * INDICATE TRUE
86   C135   86 FF                           LDAA   #$FF
87   C137   97 18                           STAA   ALLNEG
88   C139   3F                     LAST    SWI             "STOP" FOR MOTOROLA TRAINER
89   C13A                                   END

 Defined          Symbol Name          Value          References

    29     ALLNEG                       0018          46     87
    88     LAST                         C139          84
    56     LOOP                         C110          77
    28     NEGCNT                       0016          44     68     70     83
    27     NEGSUM                       0014          42     65     66
    82     NEXT                         C12F          57
    72     POSITV                       C125          61
    41     START                        C100
    25     TABADR                       0010          48
    26     TABLEN                       0012          50     82
    30     WRKCNT                       0019          51     56     72     74

        Lines Assembled :  89           Assembly Errors :  0
```

Figure 4-18 Continued.

A Bad Program

Sometimes people don't recognize the problems caused by poorly written and poorly documented programs. When they know every detail of a program, they have difficulty seeing other people's problems. Usually, the problems become clear to them when they study another person's program.

You should study the example bad program in Figure 4-19 to observe some bad practices you should avoid. This program does the same job as the program in Figure 4-18. Notice how simple things become very confusing!

Here are some general problems with this program and the assembler listing shown in Figure 4-19:


```
 1                                  * AN UNACCEPTABLE SUM NEGATIVE TABLE ENTRIES
 2                                  * PROGRAM WRITTEN IN VALID ASSEMBLY LANGUAGE
 3                                  * THAT WORKS CORRECTLY
 4                                  *
 5          0013                    R       EQU    $13
 6    0010                                  ORG $10
 7    0010                                  RMB    2          ADDRESS OF TABLE
 8    0012                          TL      RMB      1        TABLE LENGTH
 9    0013                                  RMB    2
10    0015                          TL1     RMB    1
11    0016                          WCNT1   RMB    1
12    0017                          WCNT    RMB    1
13    0018                          TEMP    RMB    2
14    001A                          NEG     RMB    2
15    001C                          *
16    C000                                  ORG $C000
17    C000   7F 00 13                        CLR    R          INIT RESULT TO ZERO
18    C003   96 13                           LDAA   R
19    C005   97 14                           STAA   R+1
20    C007   16                              TAB
21    C008   DD 1A                           STD    NEG        ZERO NEGCNT
22    C00A   7F C0 24                        CLR    ALL
23    C00D   DE 10                           LDX    $10        POINT TO TABLE
24    C00F   96 15                           LDAA   TL1
25    C011   D6 12                           LDAB   TL
26    C013   D7 16                           STAB   WCNT1
27    C015   97 17                           STAA   WCNT
28    C017   7D 00 16                L       TST    WCNT1      TEST LOOP COUNTER
29    C01A   26 16                   TH      BNE    XYZ        AND BRANCH
30    C01C   96 17                           LDAA   WCNT1+1
31    C01E   27 06                           BEQ    E
32    C020   20 10                           BRA    XYZ
33    C022   FF                     F0      FCB    $FF
34    C023   01                     ONE     FCB    $001
35    C024                          ALL     RMB    2
36    C026   96 12                   E       LDAA   TL
37    C028   D6 15                           LDAB   TL1
38    C02A   93 1A                           SUBD   NEG
39    C02C   26 03                           BNE    F
40    C02E   7A C0 24                        DEC    ALL
41    C031   3F                     F       SWI             "STOP"
42    C032   E6 00                   XYZ     LDAB   0,X        GET VALUE
43    C034   2B 02                           BMI    RST
44    C036   20 15                           BRA    BILL
45    C038   DB 14                   RST     ADDB   R+1        ADD TOTAL
46    C03A   D7 14                           STAB   R+1
47    C03C   F6 C0 22                        LDAB   F0
48    C03F   D9 13                           ADCB   R
49    C041   D7 13                           STAB   R
50    C043   CC 00 00                        LDD    #0
51    C046   5C                              INCB
52    C047   D3 1A                           ADDD   NEG
53    C049   D7 1B                           STAB   NEG+1
54    C04B   97 1A                           STAA   NEG
55    C04D   08                      BILL    INX             INCREMENT IND
56    C04E   D6 17                           LDAB   WCNT
57    C050   F0 C0 23                        SUBB   ONE        DECREMENT LOCATIONS
58    C053   D7 17                           STAB   WCNT1+1    17 & 18
59    C055   24 C0                           BCC    L
60    C057   7A 00 16                        DEC    WCNT1
61    C05A   20 BE                           BRA    L+3
62    C05C                                   END
```

Figure 4-19 Assembly language listing for a bad program.

- *Program structure.* You cannot determine the structure of the program by just looking at the listing. Without some study, you probably can't even tell that a loop is in the program. The few comments that are placed to the right of the instructions are little help.
- *Data organization.* The data section seems to be separated from the program section, but closer study will reveal that some data is placed within the program at lines 33 through 35. Careful scrutiny will reveal that the double-byte data numbers are not all stored with the high and then low bytes in adjacent memory locations.
- *Documentation.* The program does not have a title and other identification. The comments that are provided are almost useless because they don't relate to each other well enough. The abbreviations are confusing. Many of the user symbols are poorly named and cause confusion rather than helping document the program.

Here are some further problems listed by specific line numbers on the listing:

- *Line 5.* The address of a data location is defined by the EQU directive, which is confusing. For example, can you quickly see that the label R is on a double-byte number?
- *Lines 8 and 10.* A double-byte number is stored in two noncontiguous locations. This separation of the two bytes of the number not only is confusing, but it prevents proper use of the instructions available in the computer.
- *Lines 11 and 12.* The labeling of the two instructions is confusing because they represent a double-byte number that is incorrectly stored with the high byte at the higher address.
- *Line 13.* Two bytes are reserved and assigned a label when they are never used in the program.
- *Lines 17 through 22.* The instructions here put zero into some memory locations several different ways when one straightforward way would be much better.
- *Line 23.* Never put the numerical address in an instruction. Use a label.
- *Lines 24 through 27.* The instructions here copy the table length to the working counter in a disorganized way. First, the program gets the low byte of the double-byte number into the A accumulator and the high byte into the B accumulator; this order is backward from the normal use of double-byte numbers with the D accumulator. Then, it stores the two bytes in the opposite order; the high byte is stored first, and the low byte is stored second. These little inconsistencies make following the program difficult. Probably, a better solution would be to use the D accumulator.
- *Lines 28 through 32.* The instructions test the double-byte working counter one byte at a time in an unstructured and disorganized manner. Avoid using multiple branches if possible. Here, the number could be loaded into the D accumulator to set the condition code bits.

- *Line 33.* The number FF created here is a constant that should be in an instruction using immediate addressing.
- *Line 34.* The number is specified with three digits, which is confusing. Either one or two digits would be much better.
- *Line 35.* The RMB reserves two memory bytes, but only one of them is ever used by the program.
- *Line 40.* Understanding would be much easier if an instruction sequence stored the number FF into location ALL. The use of the DEC here makes the function unnecessarily dependent upon another part of the program and requires the reader to learn about that other part.
- *Line 41.* The program stops running due to this instruction that is not at the physical end of the program. This instruction is difficult to find while working with the program.
- *Lines 43 and 44.* The branch technique on lines 43 and 44 is poor. The two instructions could be replaced by a single BPL BILL, which would be much better.
- *Lines 50 and 51.* The number one is put into the D accumulator in a long and complicated way. The LDD instruction should load the number one.
- *Lines 53 and 54.* The instructions here store the D accumulator in a difficult way. Use the available instructions well by using the STD instruction.
- *Lines 56 through 58.* One byte of the working counter is decremented one way, and the other byte is decremented a second way. A better sequence would use the DEC instruction to decrement both bytes.
- *Line 61.* Don't use expressions for branch addresses—label the branch location instead and use the label in the branch instruction.

Of course, this example was developed to illustrate a large number of problems. Probably no practical program would ever be this bad. However, all the problems illustrated have been seen by the author in other people's programs.

4.8 LARGE-SCALE TOP/DOWN DESIGN

Teams of people write most programs. Usually the size of a program, and sometimes its complexity, is more than one person can handle in a reasonable period. The team approach is only successful if careful coordination occurs between the software written by various people. Spaghetti programs are almost impossible to make work under team circumstances. The top/down design ideas also apply to programming team management.

The Top/Down Team

A programming team will, at least, consist of a team leader and several other programmers. The team leader is an experienced person who has designed software and written programs. The team leader will have the responsibility of designing the top level of the program. This person also may design the second level of the program if it is small enough. Often this design phase will require interactions between the team leader, the team members, and the program users. Both flowcharts and written materials document the design. The documentation of the interactions between program modules is particularly important.

Team members

The team leader gives the upper level flowcharts and documents to other team members. Each person will work on one section of the program. The flowcharts contain only structured modules. Therefore, the team leader is sure that the modules will fit together later, unlike those of spaghetti programs.

Each person now makes additional levels of flowcharts to expand the design into lower levels. Each person must meet the requirements of structured programming in designing new program modules.

A problem

Suppose that the procedure as described were to continue until everyone on the team completes the design of their sections. Then they do the actual coding of their program modules. When these modules are put together to make the final program, catastrophe will strike! The modules almost certainly won't go together correctly.

Certainly some design work has been incorrect. The program specifications likely have changed. Therefore, much of the design work and coding is useless because they must now be changed.

The problem occurs because the implementation phase is too late. Detailed coding was done before the design and specifications were proved.

Top/Down Implementation

Top/down implementation helps avoid problems when the separate program modules developed by different team members are brought together into a single program. The idea is to start writing and running the program soon after starting the design. Start writing the program even before designing the lower levels.

Stubs

At first, writing a program before the design is complete seems impossible, and to some extent it is. Consequently, some program modules developed early in the project will be dummy modules called *stubs*. Stubs don't do correct functions, because they are substitutes for the real code. The actual code will be designed and written later. A stub may contain some useful code, but some stubs contain no code at all. Certainly using stubs requires the judgment of a knowledgeable programmer.

An example using stubs

As an example of the use of stubs, consider writing a program that must read some temperature sensors and print a report. The report displays the data collected from the sensors. At the least, the program will have an input section for reading the sensors and a report section for generating the report.

The input program module. The input program module does three major functions. First, it must read the temperature sensors. Second, it must use an algorithm or table lookup to correct the data for nonlinearity in the sensors. Third, it must scale the data values for useful units.

The temperature sensor program module will probably have three separate parts for these three functions. To write and test all the code for these will be time-consuming. It also may be impossible because the characteristics of the sensors may not yet be known. Also, the sensors may not yet be interfaced to the computer hardware, so they don't operate.

These problems should not be a deterrent to writing the program. Instead, the input program can contain a stub that generates phony data from the temperature sensors, a stub that linearizes the data in a phony way, and a stub to scale the data in a phony way.

The report program module. The report program module will print a report on a printer. The report will need headings, neat columns, titles, time and date, and so on. However, all the details needed on the report probably are not known. The required data to be presented may not have been decided. However, these problems should not deter the programmer from writing the report program.

The report program can be written with stubs. The report may contain headings with little information, the time and data may be phony, and the arrangement on the page may be less than desirable. However, the program can be written and made to work.

After the report program module and the input program module are both completed, they can be put together to make a working program. Any bugs can be found easily because the code is still very simple—much of it is stubs.

The demonstration. After the program modules have been completed using stubs, a complete working program can be demonstrated. The report generated will contain phony data created in the stubs in the input program. The organization of the report will also need improvements. Regardless of these details, a program has been written that can be demonstrated and evaluated. The team leader, the team members, and maybe the ultimate users of the program can see some results.

Probably this working system will do something that is not right—that is, incorrect, not just missing. Often people will not specify the desired results from a program completely. However, when they see it running, they say, "See, that is wrong." If what they see is based on stubs, changing the design is still relatively easy and little detailed work is lost. The program design can be changed, and the program can be demonstrated again.

Program testing

Running and demonstrating the program containing stubs checks the design, overall function, and initial coding of the program. The next step is to remove some stubs and replace them with correct modules. The best approach is to replace these one at a time so that finding the location of bugs will be easy. Each time a stub is replaced with actual code, a new aspect of the program will perform correctly. Eventually, the entire program will work correctly.

The top/down implementation technique tests most of the program modules to some extent as the program is built and the stubs are replaced. Testing done within the actual program, and not by a separate test program, is valuable. You are assured that each program module works in the context of the actual program. This built-in testing significantly reduces the need to write separate test programs.

As the program evolves toward the final design, it should be demonstrated to interested parties several times. If errors occur, they will be caught before further work is completed. Catching errors early reduces the effort needed to correct them. Likewise, if additional features are needed, adding them to the design is easier if the program is not yet completed.

Team member interactions

Usually, the team leader will merge new modules into the overall program and test their operation. The other team members do not need to interact with each other. Their individual parts of the program are largely independent of the other programmers' work. The reduced interaction improves the productivity of the team.

Replacing a team member will have little effect on the other team members. New people added to the team need learn only about their part of the program—they do not need to learn all the program.

A major problem of adding new people is the nonproductive time spent informing them about the work completed. Top/down design reduces this time, because the modules are designed to be independent of each other—the design specifications are developed before the modules are written.

The results

Top/down design applied to team programming has been very successful. The goal is to get a system working before coding all the program. The working program demonstrates the design and some working code even if the data is phony. When the entire project is completed, everyone will have confidence that the program works correctly; it has been working and demonstrated from the early stages of design.

The top/down approach to team management is possible only if you have correctly structured programs. Otherwise, the jumping around in spaghetti programs causes different programmers' modules to interact at an intimate level. Changing one person's module will affect another person's module; getting the program to function correctly will be difficult.

Studies of practical programming teams have shown that a combination of structured programming methods and top/down design methods leads to successful projects. Of course, these are techniques that do not exclude other approaches. Usually, some bottom/up testing of low-level modules is also done.

4.9 SMALL-SCALE TOP/DOWN DESIGN

An individual programmer also gains from the use of top/down design and top/down implementation. Don't try to design or write all of a program at once. Organize your efforts by making top/down structured flowcharts, and then code the program from the flowcharts. At first, code only the higher-level modules freely using stubs for the lower-level modules. Get your program running as soon as possible so that you can do some testing. You may even leave major portions of the program out of the design by making empty stubs. They can be added later. Likewise, you may write the program for hardware that is readily available, and then convert it later for the specific hardware required. Avoid taking on too much at once.

To use top/down design and top/down implementation effectively, you need good editor and assembler programs. The editor will make changes such as replacing a stub easy. After each change, immediately assemble the new program and run it to see the results. Your program will evolve to completion rapidly.

4.10 REVIEW

Good programming approaches have been demonstrated throughout this chapter. Examples have been used to illustrate both good and bad programs. This book may be unique in that it illustrates and labels bad programs. Of course, deciding whether a program is good or bad is a matter of judgment. Not everyone will agree. Therefore, this chapter has given reasons for the judgments that were made. When you write programs, you too will be making such judgments continually. You must decide what is important and set your goals accordingly.

The design of software is an activity that affords people the opportunity to be creative to an extent possible in few other endeavors. Their results can be elegant and beautiful problem solutions or ugly and costly catastrophes.

4.11 EXERCISES

4-1. Name the three fundamental program structures. Sketch a flowchart for each structure.

4-2. If a loop processes information before checking whether it should exit the loop, is it a DO-WHILE or a DO-UNTIL structure?

4-3. Figure 4-20 is a structured flowchart that contains only the three fundamental structures. Identify all the structures by drawing boxes around each structure.

4-4. If an assembly language program listing for a structured program has documentation as described in this chapter, what is the implication of two consecutive lines containing comments?

4-5. Consider an assembly language program listing for a program that has documentation as described in this chapter, and has a comment line followed by several lines of program code. What is implied if the first line beyond the comment does not have an address label, but the second line does have an address label?

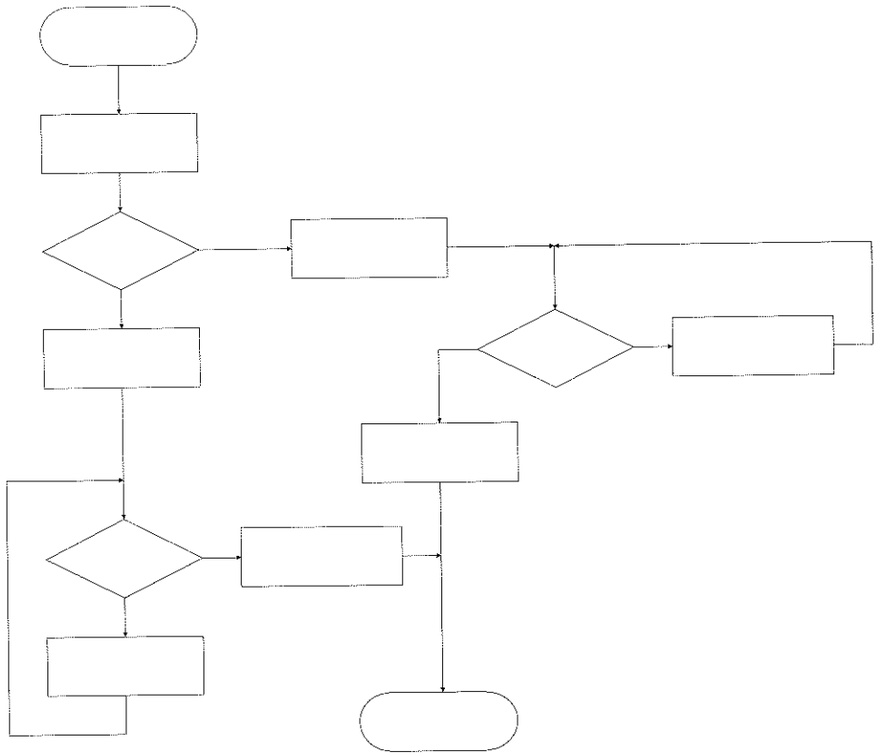

Figure 4-20 Exercise 4-3.

4-6. Make a single-level structured flowchart that represents the following algorithm:
Input a number. If the number is greater than 200_{10}, stop the program. Sum the integer numbers from 1 to (and including) the number, then print the number and the sum and stop the program.

4-7. Make top/down structured flowcharts for the following program:
Input a positive number. Print a report that lists the squares and cubes of the numbers from zero to the number read. Limit the input number to 100_{10} by printing an error message for incorrect numbers and ending the program. After the report is finished, request a YES or NO for repeating the program and respond accordingly. Include messages and headings for the report.

4-8. Make a single-level structured flowchart for the program in Exercise 3-11.

4-9. Make a single-level structured flowchart for the program in Exercise 3-14.

4-10. Make a new structured flowchart using only the three fundamental structures that is equivalent to the incorrectly structured flowchart in Figure 4-21.

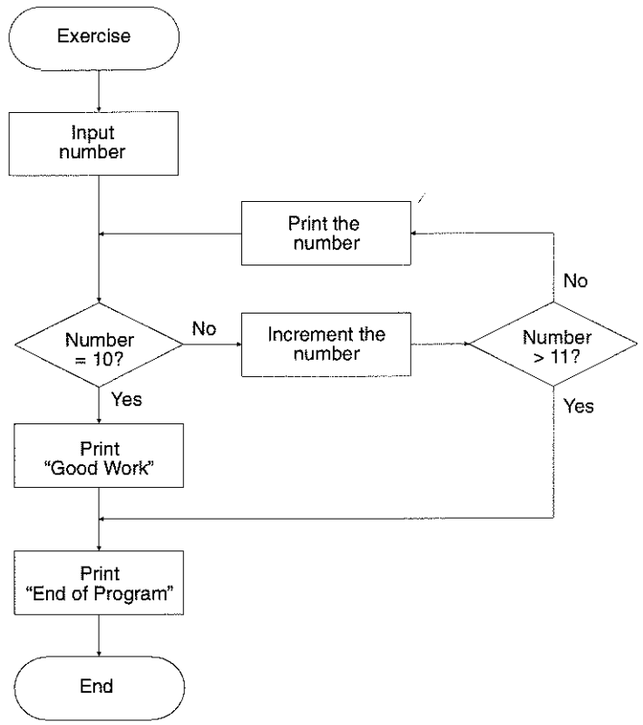

Figure 4-21 Exercise 4-10.

4-11. Find a program written in a high-level language such as Basic, Fortran, or C. Study the program to determine whether it is built from the three fundamental program structures.

4-12. Make a new structured flowchart using only the three fundamental structures that is equivalent to the incorrectly structured flowchart in Figure 4-22.

4-13. Make a new structured flowchart using the three fundamental structures and the DO-UNTIL structure that is equivalent to the flowchart in Figure 4-22.

4-14. Modify the program given in Figure 4-18 by making new flowcharts so that it sums and counts both the positive and negative entries found. Your program must be properly structured using only the three fundamental structures.

4-15. Write an assembly language program from your flowcharts developed for Exercise 4-10. Your program must follow the flowcharts, and the listing must be well documented.

4-16. Name the program structure that starts on line 13 of Figure 5-16.

4-17. Name the program structure that starts on line 38 of Figure 5-17.

4-18. Name the program structure that starts on line 44 of Figure 5-17.

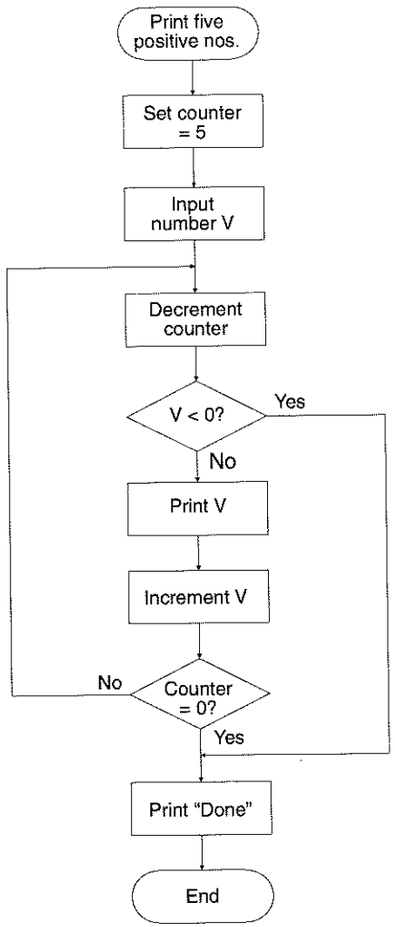

Figure 4-22 Exercises 4-12 and 4-13.

4-19. Name the program structure that starts on line 51 of Figure 5-18.

4-20. Name the program structure that starts on line 56 of Figure 5-18.

Chapter 5

Advanced Assembly Language Programming

The more complicated Motorola 68HC11 instructions that were avoided in previous chapters are introduced in this chapter. Many of the more complex instructions were designed to solve particular programming problems. This chapter discusses both the problems and the use of the more complicated instructions to solve the problems. All examples presented use assembly language, so memory diagrams are usually omitted.

5.1 MORE INDEXING

The 68HC11 has a Y index register that was avoided in previous chapters. The function of the Y index register is the same in all ways to the function of the X index register. The instruction set has a parallel instruction using the Y index register for every instruction that in any way uses the X register.

The 68HC11 has many instructions that relate to the index registers. Therefore, many instruction op code numbers are needed for these instructions.