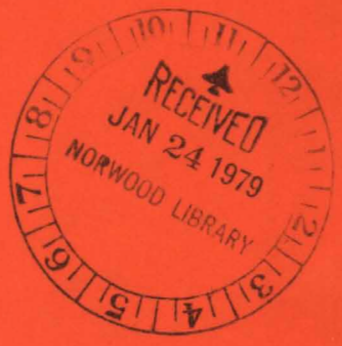
Installation Management



An Introduction to Structured Programming in FORTRAN

IBM

GC20-1790-0

An Introduction to Structured Programming in FORTRAN

This text, intended for programmers, describes and illustrates the use of structured programming. The technique and its supporting practices are generally described in one chapter. A second chapter illustrates the implementation of the technique in FORTRAN and is followed by a chapter presenting two sample programs. A knowledge of FORTRAN is assumed.

Preface

This text describes and illustrates the use of structured programming, a recently formalized programming style in which the structure of a program is made as clear as possible. Intended for programmers, the publication consists of three chapters:

1. An expository chapter describes the technique, its supporting practices, and its use. General suggestions on getting started are also included.
2. A reference chapter illustrates the implementation of the technique in FORTRAN. This chapter may be used as a starting point for establishing an installation's own structured programming guidelines.
3. A third chapter contains two sample programs written according to the techniques presented in the earlier chapters.

Familiarity with programming concepts is necessary for the expository chapter, and knowledge of FORTRAN is needed for the reference and sample program chapters.

Structured programming is also discussed in the following IBM publications:

Improved Programming Technologies – An Overview (GC20-1850), *Structured Programming (Independent Study Program)* Textbook (SR20-7149), Workbook (SR20-7150), *An Introduction to Structured Programming in COBOL* (GC20-1776), *An Introduction to Structured Programming in PL/I* (GC20-1777).

First Edition (July 1977)

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address comments concerning the contents of this publication to IBM Corporation, Technical Publications/Systems, Dept. 824, 1133 Westchester Avenue, White Plains, New York 10604.

Contents

List of Illustrations

Chapter 1: An Overview of Structured Programming

This chapter contains various items of background information about structured programming that should be useful. The topics include:

Definitions
A summary of the potential advantages of structured programming
The relationship of structured programming to other improved programming technologies
A sketch of the theoretical foundation of structured programming
The basic control logic structures
Additional control logic structures
The GO TO question
Segmentation
Indentation
Documentation considerations
Efficiency considerations
Getting started in structured programming

After reading the material in this language-independent overview, the programmer will be ready to study the reference material in Chapter 2 to see how structured programming can be implemented in FORTRAN, and to see some of the ideas illustrated in the two sample programs in Chapter 3.

Definitions

Structured programming is a style of programming in which the structure of a program (that is, the interrelationship of its parts) is made as clear as possible by using just three control logic structures:

1. Simple *sequence* of functions
2. *Selection* of functions (IFTHENELSE)
3. Loop control, or *iteration*

These three control logic structures can be combined to produce programs to handle any information processing task. Statements controlled by the selection and loop structures are intended to make obvious the scope of influence of the structure.

A structured program is composed of *segments*, which may range from a few statements up to about a page of code. Each segment has just one entry and one exit. Such a segment, assuming it has no infinite loops and no unreachable code, is called a *proper program*. When proper programs are combined using the three basic control logic structures (sequence, selection, and iteration), the result is also a proper program.

An important characteristic of a structured program is that it can be read in sequence, from top to bottom, without a great deal of the skipping around through the program that is typical of other programming styles. This feature is important because full comprehension of what a function does is easier if all the statements that influence its action are physically close by. Top-down readability is one consequence of using only three control logic structures, and of avoiding the GO TO statement except in very special circumstances, such

as the simulation of control logic structure in a programming language that lacks it.

A program written according to these principles not only *has* a structure, it clearly *exhibits* that structure.

Potential Advantages

A program written in this style tends to be much easier to understand than programs written in other styles. Easier understandability facilitates code checking, partly because structured programming concentrates on one of the most error-prone factors in programming, the logic. As a result, the program testing and debugging time may be reduced.

An easy-to-read program composed of well-defined segements tends to be simpler, faster, and less expensive to maintain. These benefits derive in part from the fact that since the program is to a significant extent its own documentation, the documentation is always up to date; this is seldom true with conventional methods.

Structured programming offers these benefits, but it should not be thought of as a panacea. Program development is still a demanding task requiring skill, effort, and creativity.

Relationship of Structured Programming to Other Improved Programming Technologies

Structured programming is compatible with and supportive of other improved programming technologies, although distinct from them. Other technologies and the relationship of structured programming to them may be sketched briefly.

Top-down program development involves writing and testing the highest level segments of a program first, in contrast to the more common method in the past, bottom-up development. This approach has the benefits of giving the critical top segments the most testing, of giving earlier warning of problems with the interfaces between segments, and of spreading the debugging and testing over a greater part of the development cycle.

Structured programming and top-down program development both emphasize the importance of segments that interact in precisely understood ways. Both involve looking at a program as a hierarchy of segments that are related to each other in a tree-like fashion.

Hierarchy plus Input-Process-Output (HIPO) is an approach to functional specification and documentation of programs. Each function is designed using a HIPO diagram, in which inputs and outputs are listed and the processing that is to be carried out is specified. A visual table of contents diagram points to the HIPO diagrams in the package and therefore shows the functions and subfunctions to be carried out by the various parts of a program, and the relationship between them. At the detailed design level, it also shows the hierarchy of segments.

Structured programming, as the term is used in this publication, refers primarily to the coding phase rather than the design phase of the program development cycle. HIPO is one good way to approach the design task, and

one that is complementary to structured programming. For additional information on *HIPO*, see *HIPO – A Design Aid and Documentation Technique* (GC20-1851).

A *structured walk-through* is a review session in which the orginator of program design material or code explains it to colleagues. The intent is to detect errors and deviations from standards and to ensure understandability. Errors are corrected after the walk-through as early in the process as possible, when they are least expensive to correct.

Structured programming, with its emphasis on easy readability of programs, increases the effectiveness of structured walk-throughs.

A *development support library* consists of a machine-readable library that contains the current versions of all project programming data. It also consists of external library binders that contain current listings of all library members and archives consisting of recently superseded listings. Besides providing easy accessibility of materials, this library helps assure that the latest versions of programs are always used.

Structured programming, with its insistence on segmentation of programs, fits in well with development support libraries, although such libraries are useful with any style of programming.

The *chief programmer team concept* involves programming with teams of at least three members: chief programmer, backup programmer, and program librarian. The team may also include other programmers, nonprogramming analysts, and end users. The chief programmer is responsible for the design and coding of all programs produced by the team, either writing or personally checking every piece of code. The program librarian maintains the development support library.

Structured programming is well suited to chief programmer team methods, since it facilitates one key element, that of code review by the chief programmer.

Structured Programming Theory

The Structure Theorem

The structure theorem states that any *proper program* can be written using only the control logic structures of *sequence, selection* (IFTHENELSE), and *iteration*.

A *proper program* is defined as one that meets the following requirements:

1. It has exactly one entry point and exactly one exit point for program control.

2. Paths from the entry to the exit lead through every part of the program; therefore, the program has no infinite loops and no unreachable code. This requirement is, not a restriction but simply a statement that the structure theorem applies only to meaningful programs.

The three basic control logic structures are defined as follows:

Sequence is simply a formalization of the idea that unless otherwise stated, program statements are executed in the order in which they appear in the program. Although this is true of all commonly used programming languages, the fact is not always realized that sequence is a control logic structure. In flowchart terms, sequence is represented by one function after the other, as shown in Figure 1. *A* and *B* are anything from single statements up to complete subprograms; the concern is only with the abstract idea of a proper program, regardless of its size and internal complexity. *A* and *B* must both be proper programs in the sense just defined (one entry and one exit). The combination of *A* followed by *B* is also a proper program, since it also has one entry and one exit. This concept can be shown pictorially, as in Figure 2, where the outer box indicates that the combination of *A* followed by *B* can be treated as a single unit for control purposes.

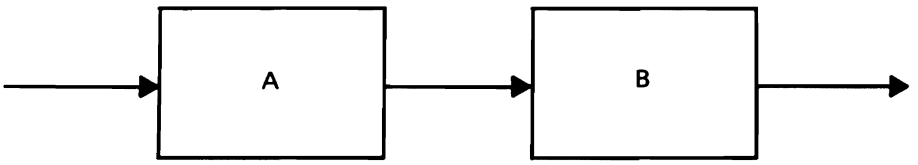


Figure 1. Flowchart for the control logic structure *sequence*

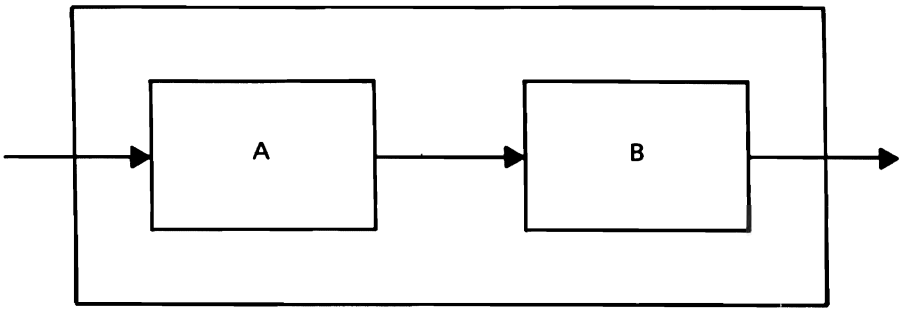


Figure 2. Two proper programs in sequence

Selection is the choice between two actions based on a *predicate*; this structure is called IFTHENELSE. The usual flowchart notation for selection is shown in Figure 3, where *p* is the predicate and *A* and *B* are the two functions.

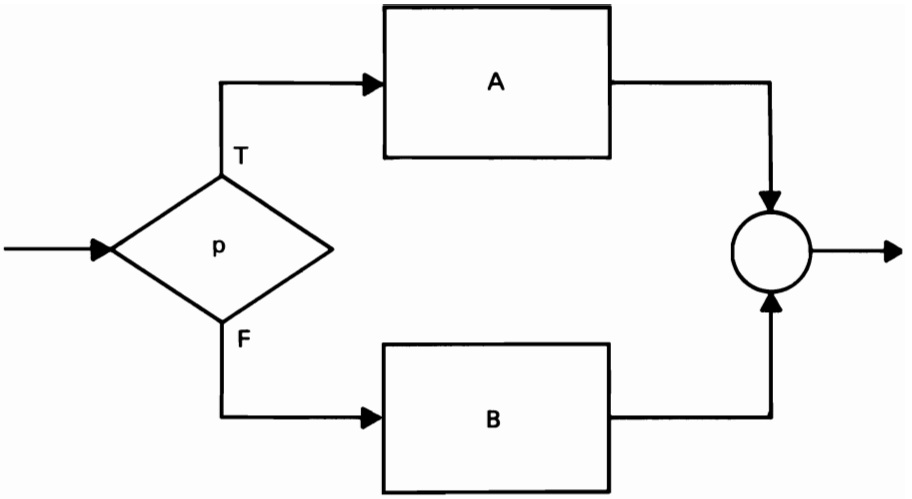


Figure 3. Flowchart for the control logic structure *selection*

The *iteration* structure, used for repeated execution of code while a condition is true (also called loop control), is the DOWHILE. In the flowchart in Figure 4, p is the predicate and A is the controlled code.

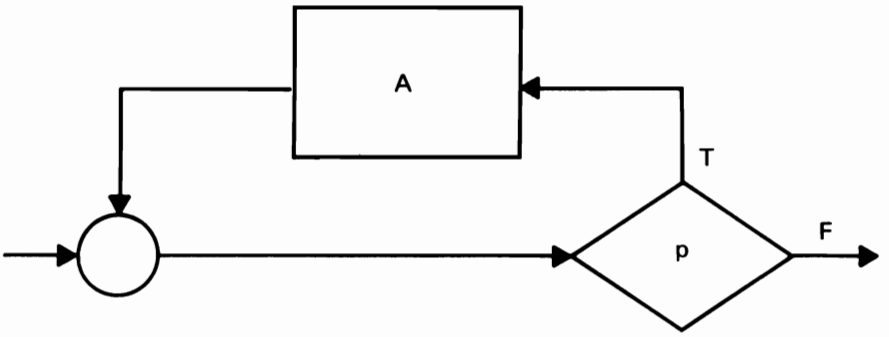


Figure 4. Flowchart for the control logic structure *iteration*, the DOWHILE

A fundamental idea is that wherever a function box appears, any of the three basic structures may be substituted, and the result is still a proper program. For example, the function box in Figure 4 could be replaced with *selection*, producing the flowchart of Figure 5. The dotted lines show where another structure has been substituted for a function. Or, one function in a *selection* might be replaced with three functions in sequence, and the other replaced with an *iteration*, producing the flowchart of Figure 6. Flowcharts of arbitrary complexity can be built up in this way. Figure 7 shows a flowchart with several control logic structures, drawn this time in top-to-bottom fashion. Other examples appear in Chapter 3.

The ability to substitute control logic structures for functions and still have a proper program is basic to structured programming. This process may also be called the *nesting* of structures.

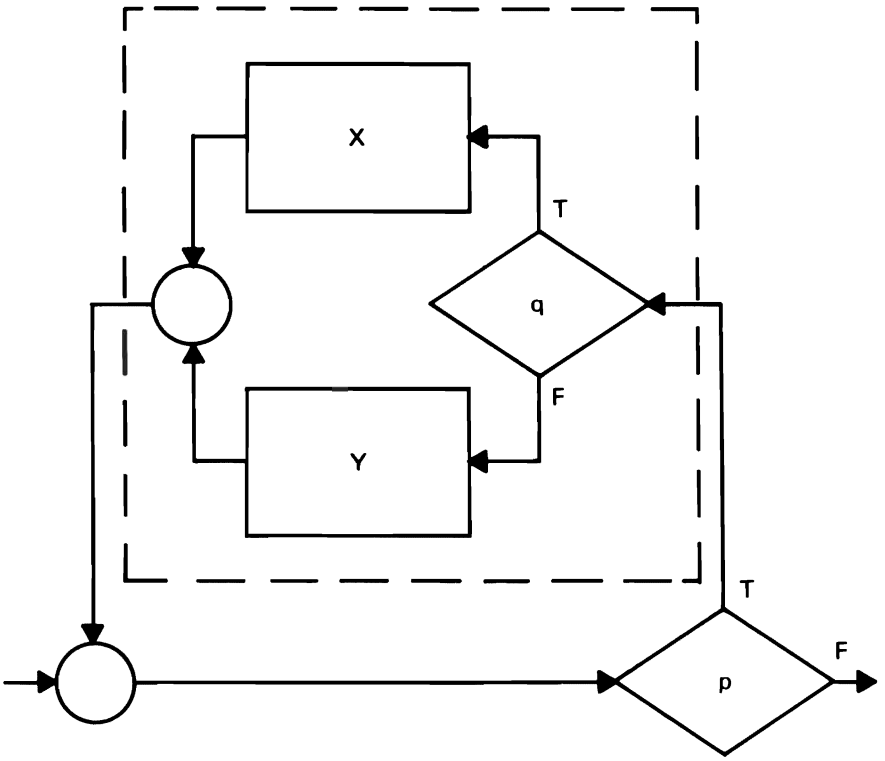


Figure 5. An example of the combination of two control logic structures, in which the function controlled by a DOWHILE is an IFTHENELSE

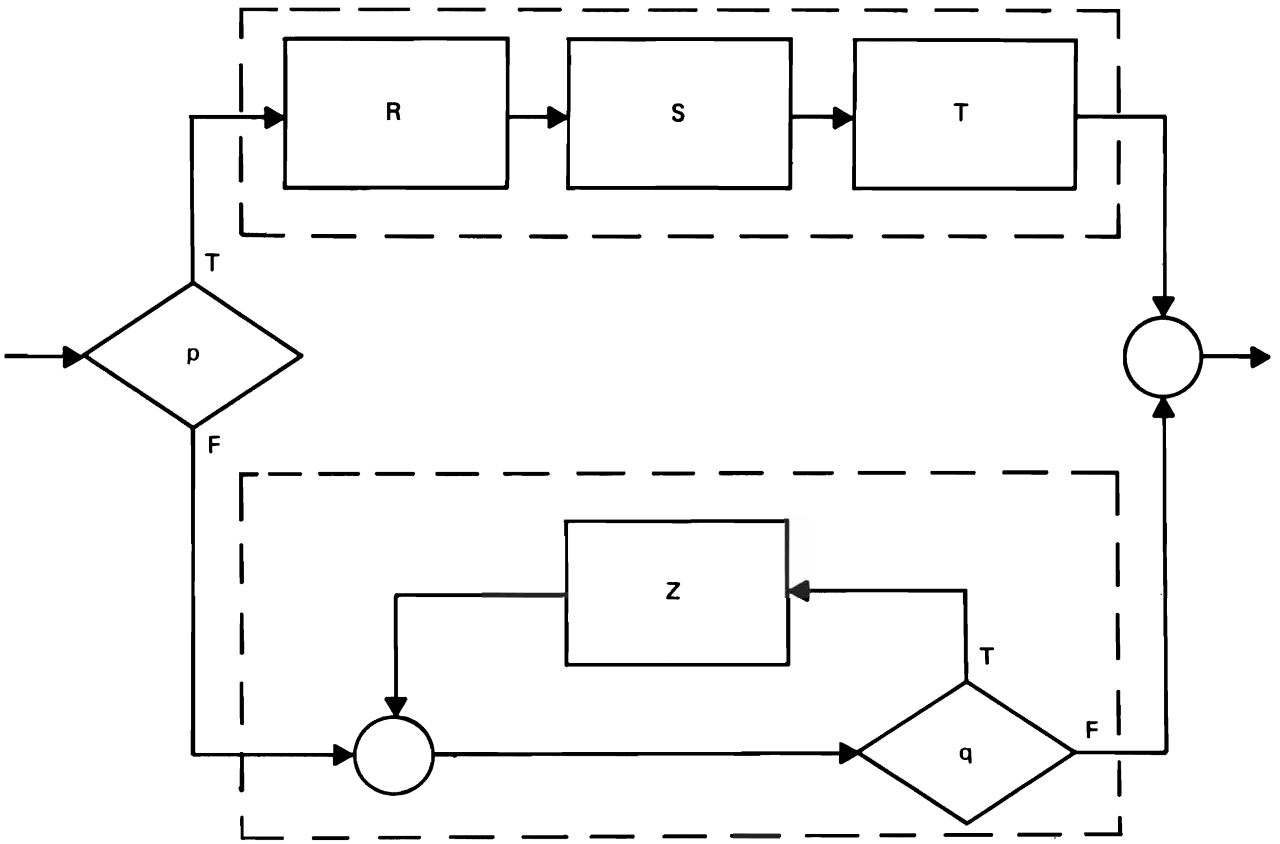


Figure 6. An example of the combination of control logic structures in which a *sequence* and an *iteration* are controlled by a selection

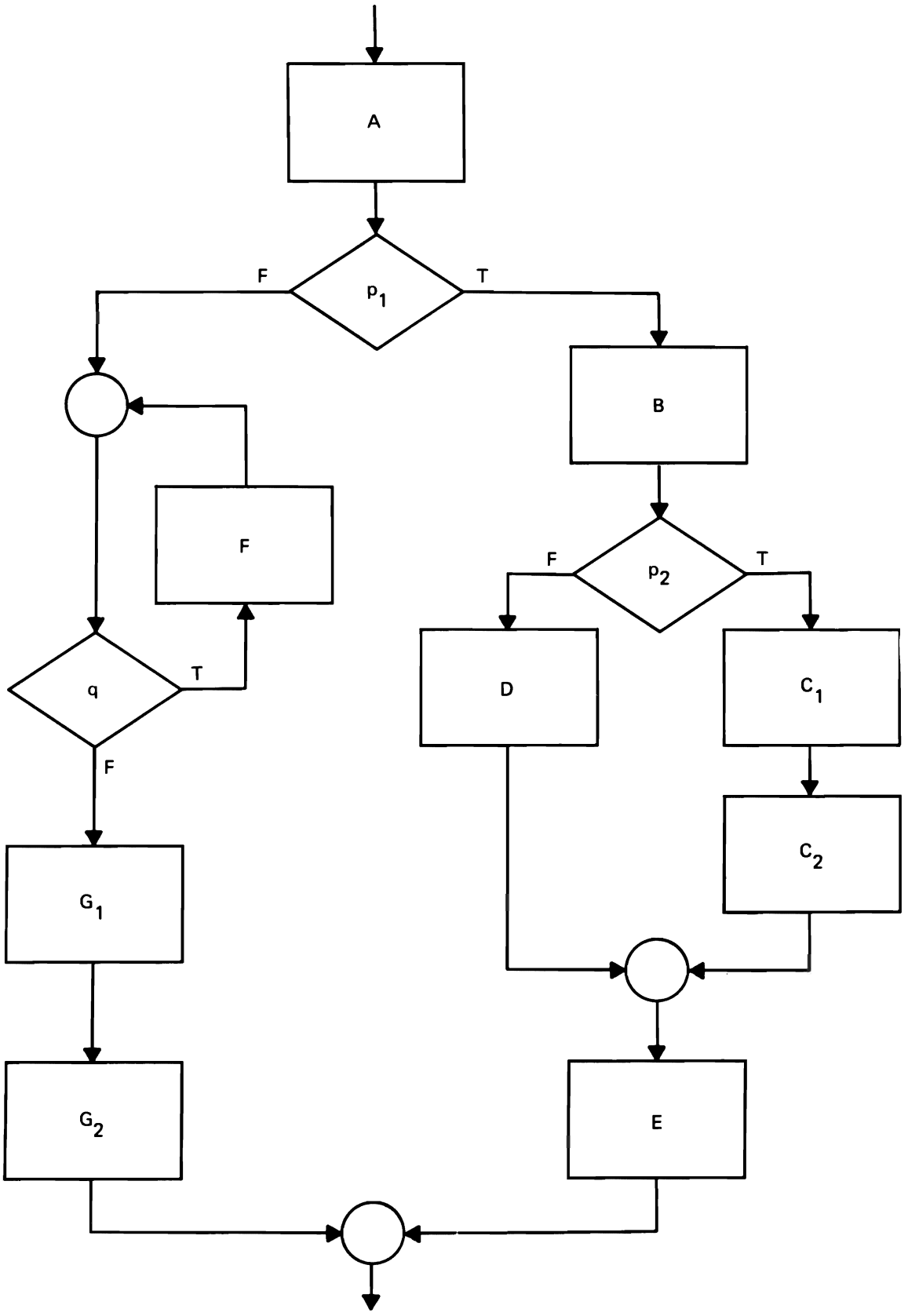


Figure 7. Another example of the combination of control logic structures

Additional Control Logic Structures

The DOUNTIL Structure

Although all programs can be written using only the three basic structures, the use of a few others is sometimes helpful.

The basic iteration structure is the DOWHILE, but a closely related structure, DOUNTIL, is sometimes used, depending on the procedure that is to be expressed and on the availability of appropriate language features. The flowchart is shown in Figure 8.

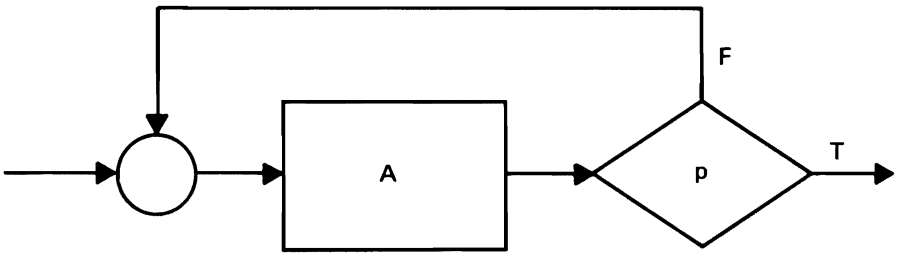


Figure 8. Flowchart for the control logic structure *iteration*, the DOUNTIL

The difference between the DOWHILE and DOUNTIL structures is that with the DOWHILE the predicate is tested *before* executing the function; if the predicate is false, the function is not executed. With the DOUNTIL, the predicate is tested *after* executing the function; thus, the function is always executed at least once, no matter whether the predicate is true or false.

The CASE Structure

It is sometimes helpful – from both readability and efficiency standpoints – to have some way to express a multiway branch, commonly referred to as the CASE structure. For example, if it is necessary to execute appropriate routines based on a two-digit decimal code, it certainly is possible to write 100 IF statements, or a compound statement including many IF's, but common sense suggests that there is no reason to adhere so rigidly to the three basic structures.

The CASE structure uses the value of a variable to determine which of several routines is to be executed. The flowchart is shown in Figure 9. Observe that DOUNTIL and CASE are both proper programs.

Efficiency and convenience dictate reasonable use of language elements that may carry out logic functions in ways slightly different from those of the three basic structures. FORTRAN examples include the use of the DO statement and the logical IF statement.

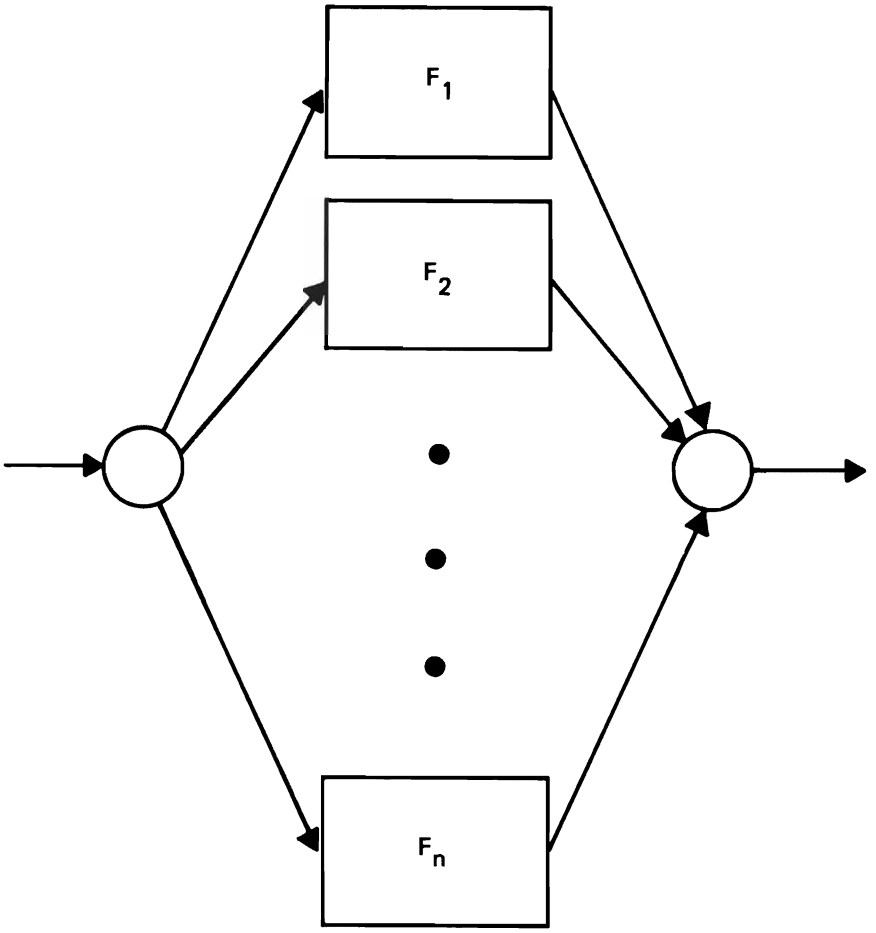


Figure 9. Flowchart for the CASE control logic structure

Labels and GO TO Statements

Structured programming has occasionally been referred to as "GO TO-less programming". Although well structured programs have few if any GO TO statements, assuming an appropriate programming language, the absence of GO TO's can be misinterpreted, and this issue should be put in context.

A well structured program gains an important part of its easy readability from the fact that it can be read in sequence, without skipping around from one part of the program to another. This characteristic is a consequence of the use of only the standard control logic structures (GO TO is not a standard control logic structure). This sequential readability, or "top-down" readability, is beneficial because the human mind is limited as to how much detail it can encompass at once. The function of a statement can be grasped far more easily if it can be understood in terms of just a few other statements, all of which are physically close by. GO TO statements generally defeat this purpose; in extreme cases they can make a program essentially incomprehensible.

The elimination of GO TO's has sometimes been misunderstood as the goal of structured programming. Although good reasons exist for not wanting to use them, no extra effort is required to avoid them; they just never occur when the standard control logic structures are used. Naturally, if the chosen programming language lacks essential control logic structures, they have to be simulated, and GO TO's are necessary; however, their use can be carefully controlled.

In certain exceptional, situations the use of GO TO's may improve readability as compared to other ways of expressing a procedure. Such examples, however, do not usually occur in everyday programming. The impact of deviations from installation guidelines, such as using GO TO's in other than prescribed ways, should be given careful consideration before such deviations are permitted.

Segmentation

Easy program readability requires that the programmer should not have to turn a lot of pages to understand how something works. A practical rule is that a segment should not exceed a page of code, about 50 lines. In FORTRAN terms a segment can be a main program, a FUNCTION, or a SUBROUTINE. (The term segment as used here has nothing to do with the different meanings of the term in connection with the functions of operating systems or data base management systems.)

Segmentation, however, is more than just breaking a program into page-size pieces. Three features that characterize good program segmentation can be identified:

1. The segmentation should reflect the division of the program into pieces that relate to each other in a hierarchy, that is, a tree structure. This organization, which may be displayed with a HIPO hierarchy chart, makes it simpler to understand how the segments relate to each other. Furthermore, the segments at the top of the hierarchy should contain high-level control functions, whereas the segments at the bottom should contain detailed functions.

2. A well-designed segment carries out functions that are closely related to each other. The programmer can more easily understand it and be sure that it does what it is meant to do. Also, when changes have to be made, either during original programming or in maintenance, there is less chance of disturbing portions of the program that do not change.

3. A well-designed segment communicates with other segments only in carefully controlled ways. Some proponents of structured programming urge that segments always consist of subroutines and that the only communication between them be through parameter lists, thus reducing the chance that segments will interact in unintended and undesirable ways.

Indentation

The use of indentation is important because consistent indentation enhances readability, so that the finished program exhibits in a pictorial way the relationships among statements. A central idea is that all the statements controlled by a control logic structure should be indented by a consistent amount, to show the scope of control of the structure. Indentation can be a major benefit, as shown in Figure 10 by the skeleton programs in pseudocode. (Pseudocode is an informal means of expressing logic.) Both programs do the same processing, but the second is far easier to understand and, therefore, to verify for correctness.

```
IF P                          IF P
THEN                          THEN
B = A + B                        B = A + B
IF Q                             IF Q
THEN                             THEN
C = 12                              C = 12
ELSE                             ELSE
C = 36                              C = 36
ENDIF                            ENDIF
IF R                                IF R
THEN                                THEN
Y = X + Y                              Y = X + Y
ELSE                                ELSE
Z = X + Z                              Z = X + Z
ENDIF                               ENDIF
ELSE                          ELSE
A = A + B                        A = A + B
ENDIF                         ENDIF
```

Figure 10. Nested IF pseudocode statement, with and without indentation

Establishing Indentation Guidelines

Guidelines for indentation in FORTRAN programs are suggested in Chapter 2. Note, however, that these are only guidelines; each installation will need to establish local conventions. Variation from the suggested guidelines is not important as long as the installation conventions are followed consistently. For example, it is not of fundamental importance whether the statements controlled by an IF are indented four spaces, or three, or two. Arguments can be made for each, but no way is absolutely right. Within any one installation, however, some set of rules should be followed or the value of indentation will be lost.

Creating a Structured Program

Structured programming, as the term is used in this publication, refers to the coding portion of the total program development cycle. It may help to sketch the cycle, indicating how structured programming relates to each phase.

The program development process can begin when, in response to a statement of requirements, a *specification* is developed that states the objectives of the application. The next step is initial design, during which each major function is identified and then subdivided into lower level functions. HIPO diagrams are a design aid and documentation tool at this stage. It is important in initial design not to become enmeshed in low-level details; the strategy is to manage complexity by attacking the problem one level of detail at a time.

Program design should not be expected to proceed in a straight-line fashion. The HIPO hierarchy chart may have to be drawn several times, as the expected segment size or the implications of logic flow become clearer. The basic idea is to begin with a top-level attack, with little detail, and then fill in the successive levels, refining original plans as necessary until the design is.

Once the initial design is complete, programmers refine the design to add the details necessary for the coding process. In *detailed program design*, additional HIPO diagrams are created to specify further detail about each process. If flowcharts are used to express logic flow, they should include only the basic structures. Another technique used in the detailed design phase is pseudocode, an informal means of expressing logic. Although HIPO diagrams can reduce the need for other documentation of logic flow, flowcharts and pseudocode can be used with HIPO diagrams.

In pseudocode the basic control logic structures and indentation are used in a carefully controlled way, but the programmer has discretion over everything else: elements of programming languages may be used, or mathematical notation if it is appropriate to the application, and so on. Pseudocode is similar to a programming language, but it is not compilable and is not bound by formal syntactical rules. Pseudocode is used to depict detailed logic while avoiding the distractions of the details of programming language requirements; it is easier to modify than programming language statements. When detailed program design is finished, the translation from pseudocode to the chosen programming language is straightforward, since the most difficult part (the logic) is finished. Examples of pseudocode appear in the illustrations in Chapter 3.

In the *coding* stage of program development, the techniques that have become identified with structured programming, as the term is used here, come into greatest prominence. Program statements implementing control logic structures are used, and they are indented to show the scope of influence of the structures; thus, the details of code are clearly related to the structure of the design. For ease of understanding, no structure is allowed to extend over a page boundary. The objective is to use meaningful variable and subprogram names, perhaps following conventions that suggest the functions of the data and procedures. Program segments are proper programs (one entry, one exit) and can be read in sequence from top to bottom.

It is becoming increasingly common for completed code to be checked by another programmer, either in a structured walk-through or in some other kind of code-reading process. During *test*, program errors are located, and a verification is made that the program performs according to specifications. With structured programs this stage may tend to take less time than before because errors can be located and corrected more rapidly in the more readable structured code.

Finally, the program has to be *maintained* over the period of its use. Specifications change, equipment configurations are modified, and coding errors are discovered; these may require program modifications. Over the life of a major program, maintenance often requires more effort than the original program development.

Structured programming facilitates program maintenance for much the same reason that it facilitates program testing: the program is easier to understand. Whether the original programmer or a different maintenance programmer is involved, changes are easier to make and are less likely to cause undesired effects elsewhere in the program.

In summary, program development consists of requirements specification, initial design, detailed design, coding, test, and maintenance. The most difficult task is design, which properly should receive the most attention and effort, since errors are least costly to correct at this stage.

Documentation

How much documentation of a program's logic is needed in addition to the program itself? In the past the argument has sometimes been made that the logic of a program should be documented with a complete set of flowcharts. This contention may need to be reevaluated for structured programs, which display their own logic better than conventional programs.

To reduce the need for documentation of logic, the code should follow certain guidelines of good programming practice that for many years have been characteristic of the best programmers. Data and subroutine names should be as indicative as possible of the functions of the data items and program elements. Tricky coding should always be avoided.

When these and other commonsense principles have been followed and the program has been written according to the principles of structured programming (only a few control logic structures, use of indentation, and top-down readability), there will be little need for documentation of the logic flowchart type. (The need for documentation of function provided by HIPO hierarchy charts and HIPO diagrams, and traditional documentation such as data layout charts, data preparation instructions, etc., is not affected by structured programming.)

Efficiency Considerations

Programmers are sometimes concerned that structured programming techniques may result in object programs that run slowly or that create problems in a virtual storage system. There is nothing inherent in the structured programming approach that leads to inefficiencies; the use of a restricted set of control logic structures and of segmentation does not automatically carry any time or space penalty.

Although no systematic study of many users has been made, some users have reported that usually no performance penalties result from structured programming techniques. If problems occur, they should be seen in the context of the full range of considerations that determine the effectiveness of a data processing operation. For instance, the ability to create programs on time may be much more important than a small object program speed penalty. Also, an apparently highly efficient program that is very difficult to maintain may not be really efficient in terms of total cost. Finally, efficiency always relates to a specific environment of compilers, hardware, and user code.

If object program speed does become a problem, however, the following approaches may be considered.

Identify those portions of the program that are most heavily used; various analysis programs may be helpful in doing so, for example, by providing counts of statement executions. A rather small part of the program will usually be found to have a large influence on speed. Concentrate on those few segments. It may be necessary to recode procedures to inline code, or to "unwind" short, heavily used loops. Consider the possibility of avoiding certain data conversions or language features that may adversely affect performance. Since usually only a small part of the program needs to be modified, this modification will ordinarily not take a great deal of effort.

If excessive paging in a virtual storage system is a problem, the basic solution is to place procedures that are used together in the same virtual storage page. Again, analysis programs can help. Structured programming can actually be a benefit in this kind of tuning, since heavy use of subroutines is encouraged. Of course the scope of the data references must be considered; extensive use of COMMON storage is generally considered inadvisable. Furthermore, performance problems can seldom be predicted in advance, no matter what coding techniques are used. Because of the ease of maintaining (changing) structured programs, the likelihood is that performance problems can be more easily corrected.

Getting Started in Structured Programming

One way to evaluate structured programming in an installation follows:

- Management authorizes the use of structured programming in a project. The first structured programming project should be neither trivial nor extremely difficult, but rather one of normal size and level of difficulty. At least two programmers should be assigned to the project so that they can check each other's code.

- Programmers assigned to the project familiarize themselves with the subject. Some installations have implemented structured programming on their own; others have found that attending a class was necessary.

- A set of guidelines for the initial effort is established. The guidelines in Chapter 2 on FORTRAN implementation can be used; many installations will prefer to establish their own. The guidelines for the first project should avoid extending the permissible control logic structures; uncontrolled extensions can easily destroy the value of structured programming. Some programmers find it helpful to summarize the guidelines in the form of a checklist or a simple illustrative program.

- After the HIPO diagrams and visual table of contents are created, pseudocode or flowcharts can be used for detailed logic, if appropriate. The code is then written and the program tested.

The evaluation process can be repeated and the guidelines modified until the programmers have sufficient experience with structured programming. At this time structured programming guidelines can be incorporated into the installation's standards.

Chapter 2: Implementing Structured Programming Using FORTRAN

Once the principles of structured programming are understood, writing structured programs in FORTRAN is a matter of habitually following a few simple rules. In FORTRAN the control logic structures other than *sequence* must be implemented using logical IF and GO TO statements; this can be done in a disciplined way, retaining most of the advantages of structured programming.

Basic Control Logic Structures

Sequence

Sequencing is implemented in FORTRAN simply by writing statements in succession.

IFTHENELSE

The IFTHENELSE structure tests a logical expression to determine which of two function blocks will be executed.

The flowchart of the IFTHENELSE structure is shown in Figure 11.

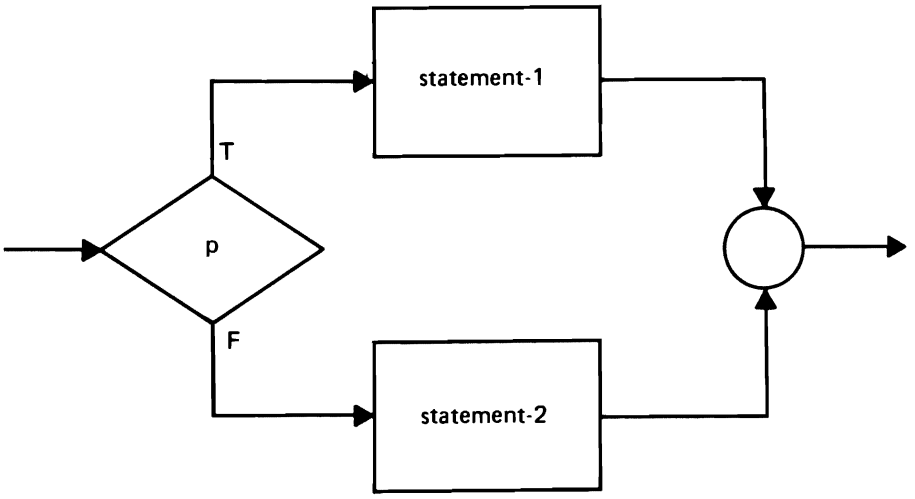


Figure 11. Flowchart for the IFTHENELSE

The pseudocode of the IFTHENELSE is:

```
IF condition-p
THEN
   statement-1
ELSE
   statement-2
ENDIF
```

The IFTHENELSE control logic structure can be implemented in FORTRAN in the following form:

```
     IF (.NOT.(p)) go to XXX
        statement-1
        .
        .
        .
        .
        statement-n
        GO TO YYY
XXX CONTINUE
        statement-2
        .
        .
        .
        .
        statement-q
YYY CONTINUE
```

XXX and YYY are statement labels assigned by the programmer.

It is permissible to reverse the logic of the condition and drop the .NOT. if doing so improves understandability, as it often will. The CONTINUE is aligned with the IF; in the case of a nested IF, the CONTINUE will not be column 7. The statements controlled by the structure are indented three spaces.

When no ELSE path is to be executed, this statement can be written in the form:

```
IF (.NOT.(p)) GO TO XXX
   statement-1
   .
   .
   .
   .
   statement-n
XXX CONTINUE
```

XXX is a statement label assigned by the programmer.

DOWHILE

The DOWHILE structure tests a predicate and executes a function as long as the predicate is true. The flowchart is shown in Figure 12.

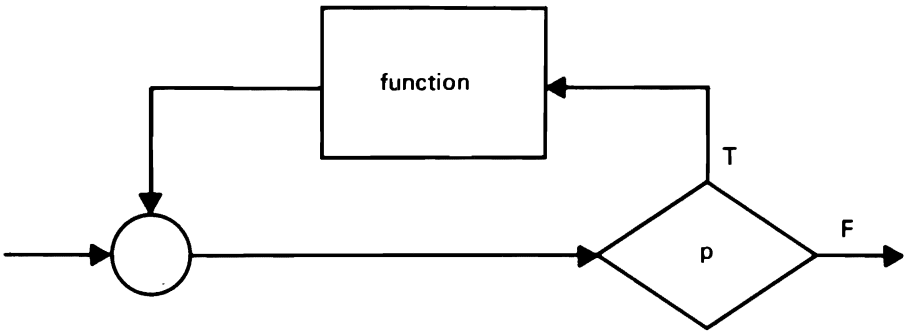


Figure 12. Flowchart for the DOWHILE

The pseudocode for the DOWHILE is:

```
DOWHILE p
   function
ENDDO
```

A DOWHILE can be implemented in FORTRAN in the following form:

```
XXX IF (.NOT.(p)) GO TO YYY
       statement-1
       statement-2
       .
       .
       .
       statement-n
       GO TO XXX
YYY CONTINUE
```

The XXX and YYY labels are assigned by the programmer.

Additional Control Logic Structures

DOUNTIL

The DOUNTIL structure executes a function and then tests a predicate to determine whether to repeat it again. The flowchart is shown in Figure 13.

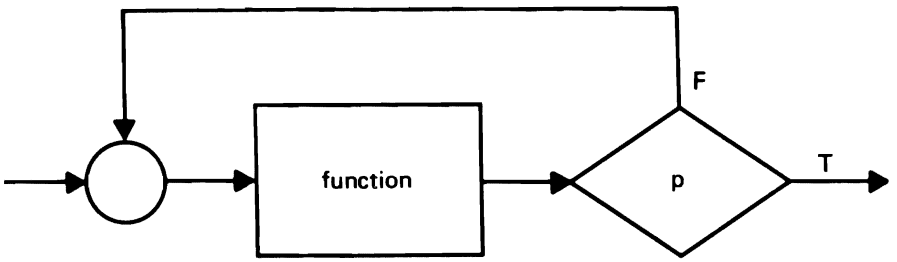


Figure 13. Flowchart for the DOUNTIL

The pseudocode for the DOUNTIL is:

```
DOUNTIL p
   function
ENDDO
```

A DOUNTIL can be written in FORTRAN in the following form:

```
XXX CONTINUE
        statement-1
        statement-2
        .
        .
        .
        statement-n
    IF (.NOT.(c)) GO TO XXX
```

XXX is a statement label assigned by the programmer.

CASE

The CASE structure selects one of a set of functions for execution, based on the basis of the value of a variable. The flowchart notation is shown in Figure 14. This construct can be coded in FORTRAN using the computed GO TO statement, as follows:

```
  GO TO (a, b, c, . . . n), kk
a CONTINUE
     statement-a1
     statement-a2
     .
     .
     .
     statement-an
     GO TO p
b CONTINUE
     statement-b1
     statement-b2
     .
     .
```

```
      .
      statement-bn
      GO TO p
      .
      .
      .
n CONTINUE
      statement-n1
      statement-n2
      .
      .
      .
      statement-nn
p CONTINUE
```

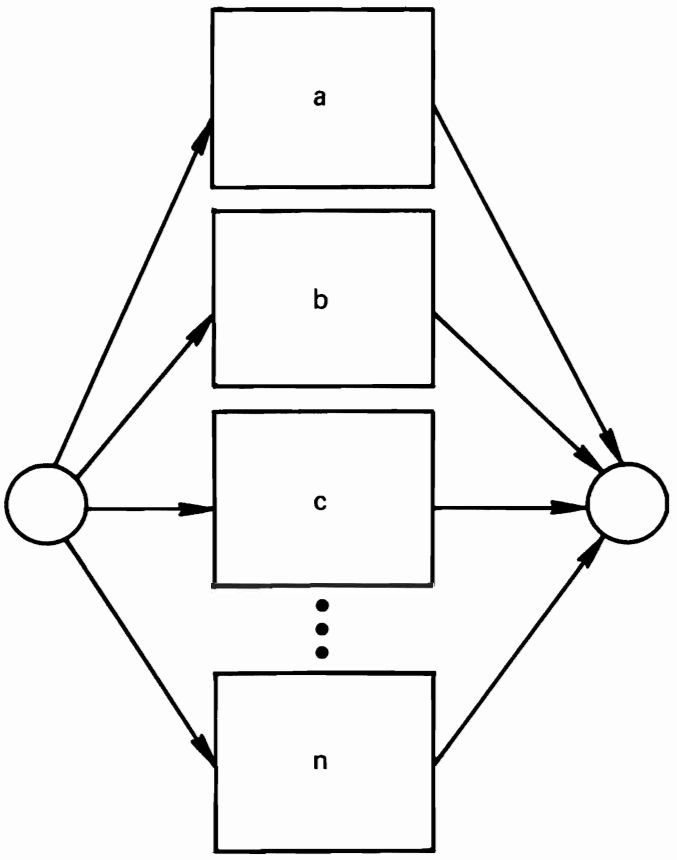


Figure 14. Flowchart for the CASE control logic structure

Program Organization

A structured FORTRAN program consists of a main program and usually a number of subprograms. The free use of subroutines to implement the division of a program into segments is encouraged. The linkage overhead is not burdensome in most modern compilers, and the advantages of good modularization are significant.

No segment – main program or subprogram – should exceed one page of code, since longer programs are more difficult to understand.

Communication between segments should be solely through SUBROUTINE or FUNCTION parameter lists, rather than through COMMON storage, since excessive use of COMMON creates interconnections between segments in a way that hampers easy program maintenance.

With minimum use of COMMON, it is less likely that a change in one segment will have an unexpected and harmful effect on some other segment.

Indentation and Readability Guidelines

No standardization of indentation and readability conventions has developed so far, and there seems to be little pressure for it as long as consistent standards are followed within any one organization. The key idea in devising these guidelines is the production of programs in which the visual layout of the program elements aids the reader in understanding program relationships and functioning. Some suggested ways of accomplishing this goal follow:

- Comment lines can be used freely to group statements having related functions.
- Statements are much easier to locate and to change if no more than one statement is written on a line.
- Placing all FORMAT statements together at the beginning of the program rather than interspersing them throughout the program aids the visual indication of program relationships.
- The scope of control of the simulated control logic structures is made clearer if the CONTINUE is aligned with the IF and the statements controlled by the structures are indented by some consistent amount. The suggested guideline is three columns, but the number is not critical as long as consistency is maintained within any one organization.
- Similarly FORTRAN IF and DO statements can be indented three spaces with the starting position of the beginning statement determined by the location of the previous statements.
- Permitting a CONTINUE only as the object of a GO TO.

The sample programs in Chapter 3 illustrate many of these guidelines.

Names

In devising names for data, considerable care should be exercised to make them as helpful as possible to the reader in understanding the function of the data elements. In spite of the FORTRAN limitation of six-character names, meaningful names can usually be chosen; for example, RESID is a better name for "residual" than R13XQ.

Comments

Experience has shown that well structured FORTRAN programs can be largely self-documenting, assuming the use of descriptive variable names. Since FORTRAN names are necessarily so short, however, it is often helpful to place a block of comments at the beginning of a segment to define the meanings of the variable names. Many programmers also recommend an initial section of comments briefly stating the function of the program.

Special Conditions

Most programming environments allow for specified unusual conditions to interrupt the normal flow of processing and activate exception-handling routines. Common examples are end-of-file conditions and arithmetic overflow. Whether the structure theorem applies to programs containing such elements depends on whether they violate the one-entry, one-exit principle and thus fail to be proper programs. Certain types of interrupts always break the normal flow; others may or may not, depending on how the program is written.

The only FORTRAN feature in this category is the END = option in the READ statement, for specifying the action to be taken when the input end of file is detected. If only a few simple operations or none remain to be carried out when the end of file is found, it may be acceptable to transfer to a closing section of the program with the END = . In other cases, however, it may be preferable to set a flag so that DOWHILE or DOUNTIL logic can be used to complete processing. Setting such a flag involves the use of GO TO statements, which is unavoidable given the FORTRAN syntax, since the END = is essentially a GO TO. Both methods are illustrated in the sample programs in Chapter 3.

Chapter 3: Two Illustrative Programs

The best way to get a quick idea of any programming technique is to see examples of programs that employ it. In that spirit, two illustrative programs are presented that have been written following the principles discussed earlier. The IBM FORTRAN IV (G1) Compiler (5734-F02) Release 2.0 was used to compile the examples in this chapter. The programs were executed under OS/VS2 Release 3.7 (MVS) and the TSO-3270 Structured Programming Facility (SPF) (5740-XT2). The programs were executed under VM/370 Version 2 Level 0.

The first program involves a very common and fairly simple operation in commercial data processing. Even if the reader's primary interest is in mathematical and engineering uses of computers, the program provides a good introduction to the use of structured programming in FORTRAN. The second program, for solving simultaneous equations, is more representative of the way FORTRAN is commonly used and is somewhat more complex in that it involves a main program and three subroutines.

A Two-Level Control Total Program

One of the most common data processing operations is the preparation of a summary report providing totals broken down by several levels of control, as well as a final total. A two-level control total report illustrates the basic ideas and can easily be extended to any number of levels. In this example it is assumed that the only report needed is the summary; extension of the program to include other processing and the printing of a detail line for each input record would involve no conceptual difficulties.

For concreteness, it is assumed that the major control is a sales district and that the minor control is a salesman number. Each record contains a district number, a salesman number, and a dollar amount. The transaction file has already been sorted into sequence on salesman within the district. To keep things simple, the printing of headings and the counting of lines on the pages are omitted.

Figure 15 is a HIPO diagram for this processing. A pseudocode representation is shown in Figure 16. Notice how the logic is clearly exhibited by the use of indentation with the basic control structures of sequence, selection, and loop control. The DOWHILE is used for the loop control with the controlled code shown inline. The same logic is shown in flowchart form in Figures 17a and 17b. Working either from the pseudocode or the flowchart, the FORTRAN program in Figure 18 is not difficult to prepare.

Note the use of blank comment lines, lines containing nothing but a C in column 1, to group statements having related purposes for easy readability. All FORMAT statements have been placed together at the beginning of the program. The alternative approach of placing them immediately after the statements that reference them is less preferable since it would diminish the value of indentation in providing a visual indication of program relationships.

The program fairly directly implements the pseudocode in setting a flag when the end of file is detected. Under FORTRAN syntax for the READ statement, this cannot be done without GO TO's since the END = feature is, itself,

essentially a GO TO. The CONTINUE statement is not actually required here, but the use of a CONTINUE as the only allowable object of a GO TO or DO statement is recommended by many programmers as enhancing program maintainability.

This program was run with a small sample of test data, producing the output shown in Figure 19. The program is quite rudimentary, since it does not include printing of headings, counting of lines, checking for sequence or other errors in the data, or any processing of the records other than the accumulation of totals. All of these operations can be included readily while still following structured programming concepts.

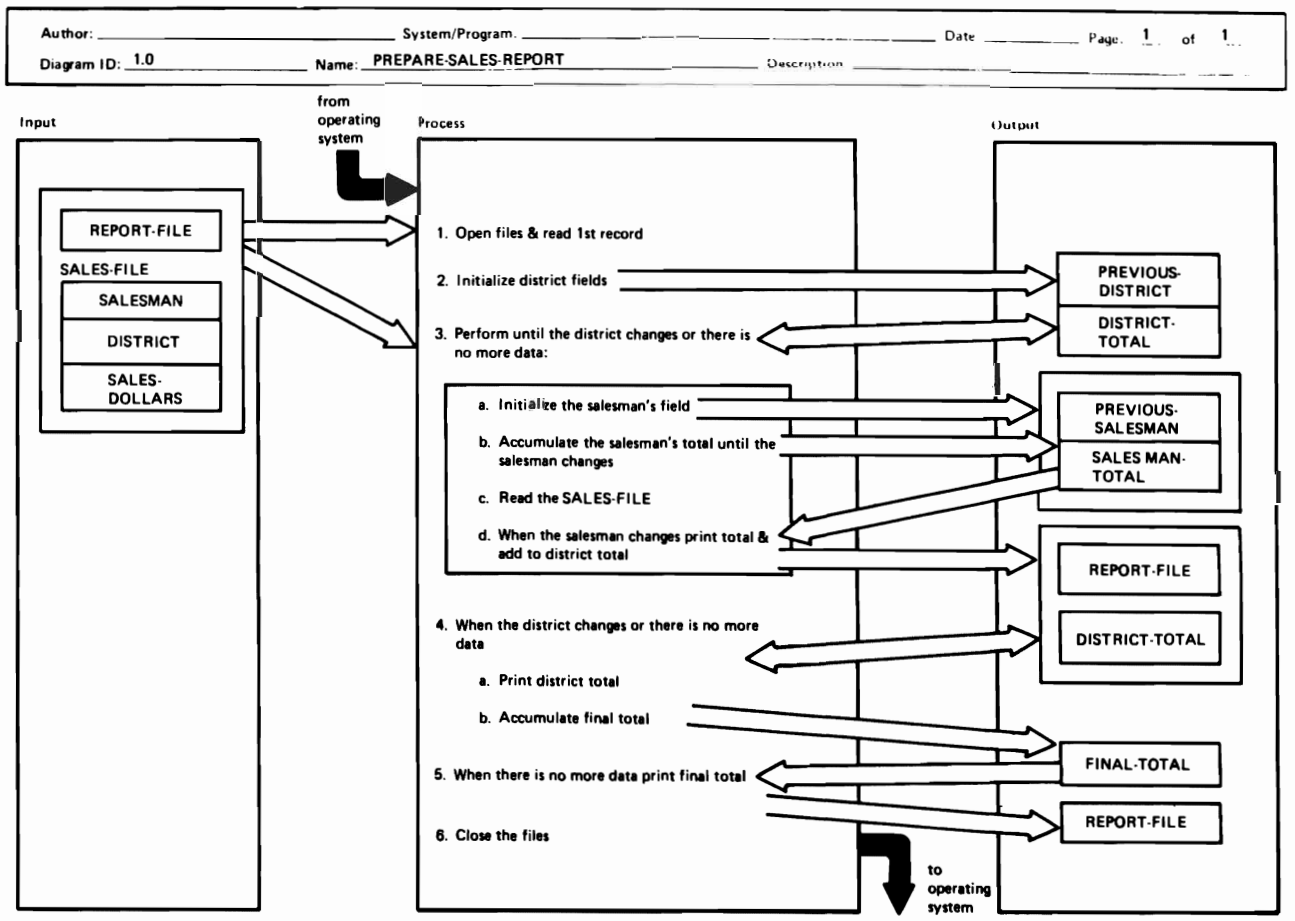


Figure 15. Detailed design level HIPO diagram for a two-level control total processing application

```
BEGIN SALES REPORT
   READ SALESMAN, DISTRICT NUMBER, DOLLARS
   ZERO FINAL TOTAL
   SET END OF DATA FLAG OFF
   DO WHILE NOT END OF DATA
      ZERO DISTRICT TOTAL
      MOVE DISTRICT NUMBER TO PREVIOUS DISTRICT
      DO WHILE DISTRICT = PREVIOUS DISTRICT
            AND NOT END OF DATA
         ZERO SALESMAN TOTAL
         MOVE SALESMAN TO PREVIOUS SALESMAN
         DO WHILE DISTRICT = PREVIOUS DISTRICT
               AND SALESMAN = PREVIOUS SALESMAN
               AND NOT END OF DATA
            ADD SALES DOLLARS TO SALESMAN TOTAL
            READ SALESMAN, DISTRICT NUMBER, DOLLARS
               IF END OF DATA
                  SET END OF DATA FLAG ON
               ENDIF
         ENDDO
         WRITE PREVIOUS SALESMAN, SALESMAN TOTAL
         ADD SALESMAN TOTAL TO DISTRICT TOTAL
      ENDDO
      WRITE PREVIOUS DISTRICT, DISTRICT TOTAL
      ADD DISTRICT TOTAL TO FINAL TOTAL
   ENDDO
   WRITE FINAL TOTAL
END SALES REPORT
```

Figure 16. Pseudocode for a two-level control total processing application

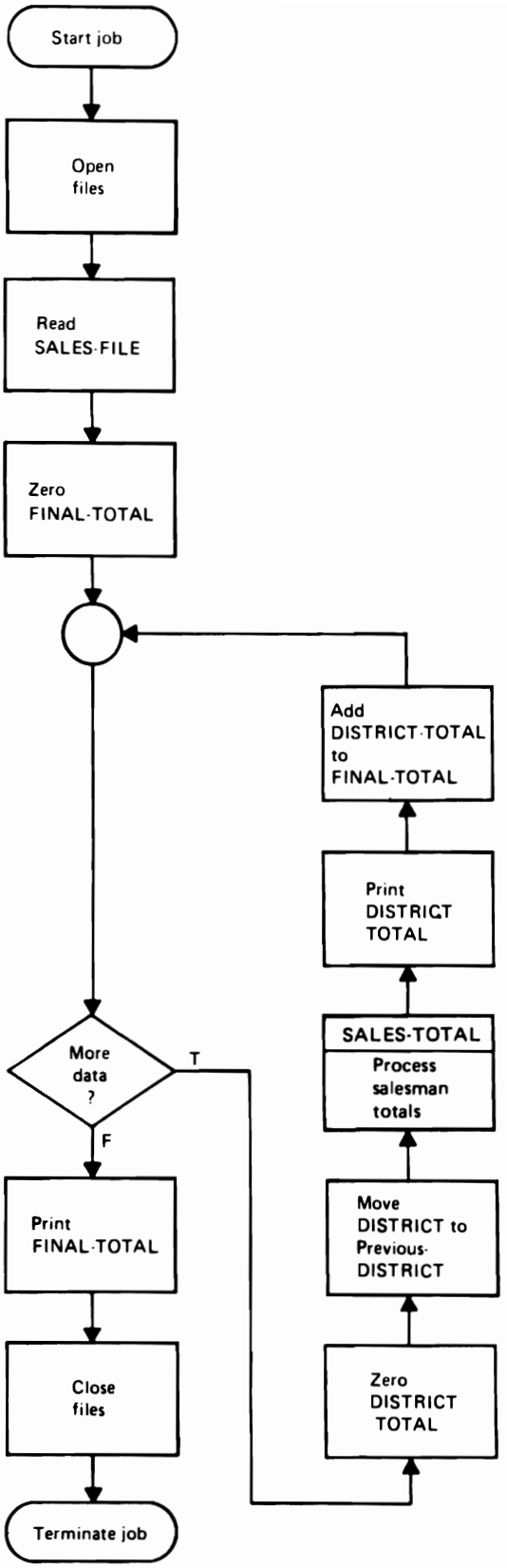


Figure 17a. Flowchart for the mainline processing portion of a two-level control total processing application

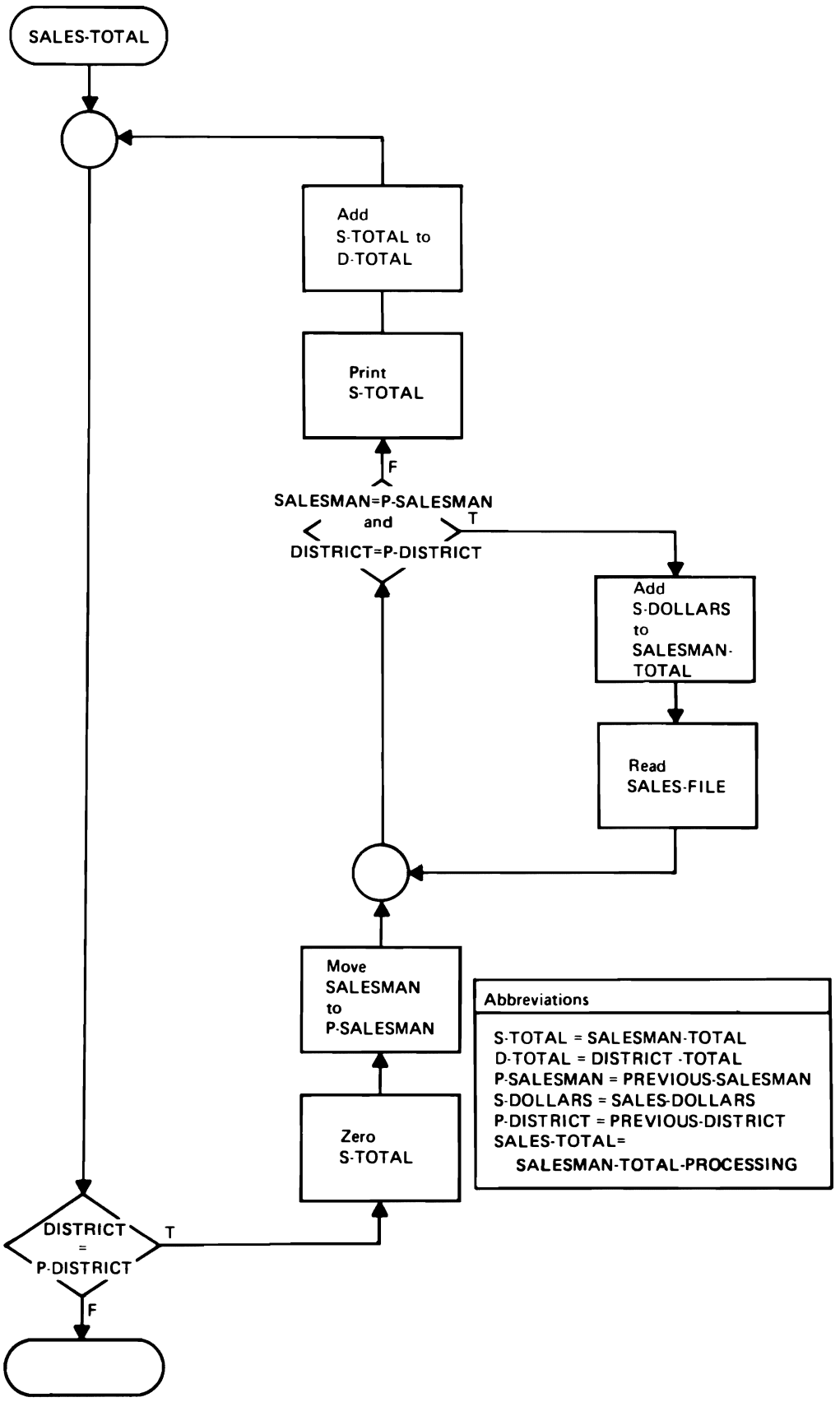


Figure 17b. Flowchart for the record processsing portion of a two-level control total processing application

```
C A PROGRAM TO PRODUCE A TWO-LEVEL SALES REPORT
C
C VARIABLE NAMES
C
C      DIST    DISTRICT NUMBER
C      PDIST   PREVIOUS DISTRICT
C      SLSMN   SALESMAN
C      PSLSMN  PREVIOUS SALESMAN
C      DATAFG  FLAG TO INDICATE THAT END OF DATA HAS BEEN REACHED
C      DISTOT  DISTRICT TOTAL
C      SLSTOT  SALESMAN TOTAL
C      FINTOT  FINAL TOTAL
C      DOLLRS  SALES DOLLARS
C
C      INPUT IS IN ASCENDING SEQUENCE ON SALESMAN WITHIN DISTRICT
C
       INTEGER DIST, PDIST, SLSMN, PSLSMN
       REAL DISTOT, SLSTOT, FINTOT, DOLLRS
C
    10 FORMAT (I5, I3, F7.2)
    20 FORMAT (1X, I5, F12.2)
    30 FORMAT (1X, 17X, I3, F12.2)
    40 FORMAT (1X, 32X, F12.2)
C
       READ (5, 10) SLSMN, DIST, DOLLRS
       FINTOT = 0.0
       DATAFG = 1
 1001 CONTINUE
       IF (DATAFG .NE. 1) GO TO 1007
          DISTOT = 0.0
          PDIST = DIST
 1002     CONTINUE
          IF (DIST .NE. PDIST .OR. DATAFG .NE. 1) GO TO 1006
             SLSTOT = 0.0
             PSLSMN = SLSMN
 1003        CONTINUE
                IF (      DIST .NE. PDIST
     1               .OR. SLSMN .NE. PSLSMN
     2               .OR. DATAFG .NE. 1) GO TO 1005
                SLSTOT = SLSTOT + DOLLRS
                READ (5, 10, END = 1004) SLSMN, DIST, DOLLRS
                GO TO 1003
 1004           CONTINUE
                DATAFG = 0
                GO TO 1003
 1005        CONTINUE
             WRITE (6, 20) PSLSMN, SLSTOT
             DISTOT = DISTOT + SLSTOT
             GO TO 1002
 1006     CONTINUE
          WRITE (6, 30) PDIST, DISTOT
          FINTOT = FINTOT + DISTOT
          GO TO 1001
 1007 CONTINUE
       WRITE (6, 40) FINTOT
       STOP
       END
```

Figure 18. Structured program for a two-level control total processing application

```
41              203.37
52              110.00
69              134.65
                                1               448.02
18              207.69
32              185.60
                                2               393.29
36              194.15
39              121.40
50               51.80
                                3               367.35
                                                                1208.66
```

Figure 19. Illustrative output from the two-level control total program of Figure 18

Solving a System of Simultaneous Equations by the Gauss-Seidel Method

This example is for the benefit of readers more concerned with technical applications. It assumes some familiarity with simultaneous linear algebraic equations and with their iterative solution by the Gauss-Seidel method.

As many as 80 equations in 80 unknowns are to be permitted; the actual size N, which may be smaller than 80, is read from the first data card. This card also specifies MAXIT, the maximum number of iterations to be permitted, the convergence criterion EPSLON, and the largest absolute value permitted of an element in the system array, BIGGST. The array is initialized to zero, so that only the nonzero elements need be read; row and column numbers are checked for validity as the data cards are read. All data values are checked and errors reported, but the solution is not attempted if any errors are found.

Not all systems of simultaneous equations can be solved by the Gauss-Seidel method. After the coefficients and constant terms have been read, a check is made to determine that the main diagonal element in each row is larger in absolute value than the sum of the absolute values of the other coefficients in the row. If not, the error is reported and the solution is not attempted.

The actual solution proceeds in a succession of sweeps. Starting with all zeros for the unknowns, new values for all unknowns are computed in one sweep. A variable named RESID holds the largest difference between the old and new values of unknowns. When this residual is found to be less than the convergence criterion, the system has been solved. If convergence cannot be achieved in the specified maximum number of iterations, the nonconvergence is reported.

If all data values are acceptable, if the system is suitable for solution by the Gauss-Seidel method, and if the solution converges, the values of the N unknowns are printed as the solution.

Figure 20 shows pseudocode for the method of solution that is to be used. Observe how the logic of the solution is displayed, without distracting details. For example, the precise form of switch-setting is left to be detailed in the program. Likewise, in the procedure for reading the data, the line "IF data card invalid" appears. Although this line conveys the meaning clearly, it does not specify exactly what tests are to be made; those details can be found in the program specifications and in the program. Note, too, that a summation sign denotes this commonly used mathematical function, which in the program becomes a simple DO loop. If it were necessary to keep the pseudocode in machine-readable form, as is sometimes the practice, the Greek symbols would naturally have been represented in some transliteration, or the loop could be shown in detail.

This program is shown in Figure 21. The mainline logic, according to which the various tests are made to determine at each stage what further actions are possible, is made clear by the use of meaningful data names, simple IFTHENELSE logic, and consistent indentation. Observe the use of ordinary FORTRAN DO statements in a situation where their use is natural and easy to understand, and where some compilers may produce more efficient object code than if a DOWHILE or DOUNTIL were used.

The subroutine READAT (for "read data") obtains the data and tests the validity of each element separately. The choice of how much testing to do is a design decision that is taken for granted here; if further tests, such as the reasonableness of the value N, were desired, they could be incorporated easily.

The subroutine VALDAT (for "validate system") is called into play if it is determined that the individual elements are acceptable. This function could, of course, have been made part of READAT, which might then have been named something like REDVAL, or READAT could have called this subroutine. The form chosen was picked because it gives the clearest picture of the logic at the top level.

The actual solution of the system, if it is found to be potentially solvable, is done with the subroutine named SOLVER. Observe the use of the DOUNTIL control logic structure to get one unconditional execution of the controlled code before testing the condition. Notice the use of two FORTRAN DO statements for heavily used loops in familiar matrix operations for which the DO statement is well suited. Notice the use of a FORTRAN logical IF statement where only one statement is to be controlled and the operation is in a heavily used inner loop. Finally, note the use of the built-in function AMAX1 to establish whether the newly computed difference between the old and new values of an unknown is greater than the previous value of RESID; this could also have been done with an IF statement.

After the program is tried with various erroneous data to check the error-detection handling, it is tested with the system shown in Figure 22. Using a convergence criterion (IPSLON) of 0.01, the method finds the solution shown in Figure 23.

```
Open files
Initialize bad data switch off
Clear arrays
Read data
IF no errors in data
THEN
  Validate system
  IF system is valid
  THEN
    Attempt to solve system
    IF solution converges
    THEN
       Print results
    ELSE
       Print 'did not converge'
    ENDIF
  ELSE
    Print 'cannot solve this sytem by Gauss-Seidel'
  ENDIF
ELSE
  Print 'bad data'
ENDIF
Close files
```

Figure 20. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (1 of 4)

```
Read data:

Get N, maximum iterations, epsilon, biggest
More data switch = yes
DOWHILE more data remains
   Get a card
   IF more data remains
   THEN
      IF data card invalid
      THEN
         Print data values and error message
         Set bad data switch on
      ELSE
         Store element in array
      ENDIF
  ENDIF
ENDDO
```

Figure 20. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (2 of 4)

```
Validate system:


DO I = 1 to N
    WHILE no bad rows have been found
  SUM = Σ (i≠j) |a_ij|
  IF |a_ij| ≤ SUM
  THEN
     Set bad row switch on
  ENDIF
ENDDO
```

Figure 20. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (3 of 4)

```
Solve system:


Iterations = 1

DOUNTIL iterations > max iterations or residual ≤ epsilon
    Residual = 0
    DO I = 1 to N
       Sum = Σ (i≠j) a_ij x_j
       Temporary = (a_i,n+1 - Sum) /a_ii
       Residual = max(residual, abs(temporary - x_i))
       x_i = temporary
    ENDDO
    Add 1 to iterations
    If iterations > maximum permitted
    THEN
       Set no-converge switch on
    ENDIF
ENDDO
```

Figure 20. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (4 of 4)

```
C A PROGRAM TO SOLVE UP TO 80 SIMULTANEOUS EQUATIONS
C BY THE GAUSS-SEIDEL METHOD
C
C          FORTRAN VERSION
C
      INTEGER I, J, N, MAXIT, BADDAT, VALID, CNVRGE
      REAL A(80, 81), X(80), EPSLON, BIGGST
C
   10 FORMAT (1X, I2, 1PE14.6)
   20 FORMAT ('0', 'DID NCT CONVERGE IN ', I4, 'ITERATIONS')
   30 FORMAT ('0', 'CANNOT SOLVE THIS SYSTEM BY GAUSS-SFIDEL')
   40 FORMAT ('0', 'BAD DATA -- JOB ABORTED')
C
      DO 1002 I = 1, 80
         X(I) = 0.0
         DO 1001 J = 1, 81
            A(I, J) = 0.0
 1001    CONTINUE
 1002 CONTINUE
C
      BADDAT = 0
      CALL READAT (A, N, MAXIT, EPSLON, BIGGST, BADDAT)
      IF (BADDAT .NE. 0) GO TO 1007
         VALID = 1
         CALL VALDAT (A, N, VALID)
         IF (VALID .NE. 1) GO TO 1005
            CNVRGE = 1
            CALL SOLVER (A, X, N, EPSLON, MAXIT, CNVRGE)
            IF (CNVRGE .NE. 1) GO TO 1003
               WRITE (6, 10) (I, X(I), I = 1, N)
               GO TO 1004
 1003       CONTINUE
               WRITE (6, 20) MAXIT
 1004       CONTINUE
            GO TO 1006
 1005    CONTINUE
            WRITE (6, 30)
 1006    CONTINUE
         GO TO 1008
 1007 CONTINUE
         WRITE (6, 40)
 1008 CONTINUE
      STOP
      END
```

Figure 21. Structured program to solve simultaneous equations by the Gauss-Seidel method (1 of 3)

```
      SUBROUTINE READAT (A, N, MAXIT, EPSLON, BIGGST, BADDAT)
      INTEGER I, J, N, NPLUS1, MAXIT, BADDAT
      REAL A(80, 81), EPSLON, BIGGST, TEMP
C
   10 FORMAT (2I2, 2F10.0)
   20 FORMAT (2I2, F10.0)
   30 FORMAT (1X, 'ERROR IN CARD WITH I = ', I2, ', J = ',
     1        I2, ', VALUE = ', 1PE14.6)
C
      READ (5, 10) N, MAXIT, EPSLON, BIGGST
      NPLUS1 = N + 1
 1001 CONTINUE
      READ (5, 20, END = 1004) I, J, TEMP
      IF (       (I .GE. 1)
     1     .AND. (I .LE. N)
     2     .AND. (J .GE. 1)
     3     .AND. (J .LE. NPLUS1)
     4     .AND. (ABS(TEMP) .LT. BIGGST) ) GO TO 1002
          WRITE (6, 30) I, J, TEMP
          BADDAT = 1
          GO TO 1003
 1002 CONTINUE
          A(I, J) = TEMP
 1003 CONTINUE
          GO TO 1001
C
 1004 CONTINUE
      RETURN
      END
```

Figure 21. Structured program to solve simultaneous equations by the Gauss-Seidel method (2 of 3)

```
      SUBROUTINE VALDAT (A, N, VALID)
      INTEGER I, J, N, VALID
      REAL A(80, 81), SUM
C
      I = 1
 1001 CONTINUE
      IF (VALID .NE. 1 .OR. I .GT. N) GO TO 1003
         SUM = 0.0
         DO 1002 J = 1, N
            IF (I .NE. J) SUM = SUM + ABS(A(I, J))
 1002    CONTINUE
         IF (ABS(A(I, I)) .LT. SUM) VALID = 0
         I = I + 1
         GO TO 1001
 1003 CONTINUE
      RETURN
      END
      SUBROUTINE SOLVER (A, X, N, EPSLON, MAXIT, CNVRGE)
      INTEGER I, J, N, ITERS, MAXIT, CNVRGE, NPLUS1
      REAL A(80, 81), X(80), SUM, TEMP, RESID
C
      NPLUS1 = N + 1
      ITERS = 1
 1001 CONTINUE
         RESID = 0.0
         DO 1003 I = 1, N
            SUM = 0.0
            DO 1002 J = 1, N
               IF (I .NE. J) SUM = SUM + A(I, J) * X(J)
 1002       CONTINUE
            TEMP = (A(I, NPLUS1) - SUM) / A(I, I)
            RESID = AMAX1 (RESID, ABS(X(I) - TEMP))
            X(I) = TEMP
 1003    CONTINUE
         ITERS = ITERS + 1
      IF (RESID .GE. EPSLON .AND. ITERS .LE. MAXIT) GO TO 1001
      IF (ITERS .GT. MAXIT) CNVRGE = 0
      RETURN
      END
```

Figure 21. Structured program to solve simultaneous equations by the Gauss-Seidel method (3 of 3)

$$
\begin{aligned}
12.063\,x_1 + 1.018\,x_2 - 4.200\,x_3 + 0.110\,x_4 &= 3.013 \\
1.934\,x_2 + 1.011\,x_3 - 0.500\,x_4 &= 1.165 \\
-0.110\,x_1 + 0.901\,x_2 + 6.914\,x_3 + 0.100\,x_4 &= 18.429 \\
-1.952\,x_1 + 2.139\,x_3 + 5.000\,x_4 &= -15.500
\end{aligned}
$$

Figure 22. System of simultaneous equations used to test the program of Figure 21

```
 1   1.493188E+00
 2  -1.947272E+00
 3   2.997915E+00
 4  -3.799566E+00
```

Figure 23. Output of the program of Figure 21 when run with sample data corresponding to the system of simultaneous equations shown in Figure 22

An Introduction to Structured Programming in FORTRAN

Installation Management Manual

GC20-1790-0

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

_____

_____

_____

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 825
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | Please Do Not Staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

An Introduction to Structured Programming in FORTRAN

Installation Management Manual

GC20-1790-0

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape | **Please Do Not Staple** | Fold and tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 825
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | **Please Do Not Staple** | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

Cut or Fold Along Line

Installation Management Manual An Introduction to Structured Programming in FORTRAN Printed in U.S.A. GC20-1790-0

GC20-1790-0

Installation Management Manual An Introduction to Structured Programming in FORTRAN Printed in U.S.A. GC20-1790-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601