# Overview of
# The Methodology of Functional Blocks

## 1. Purpose

The methodology of Functional Blocks, as developed by the COMP-AID Company in New Braunfels, Texas, is applicable to

- the r*eclaiming* of old, difficult-to-understand FORTRAN modules:

  When used in conjunction with the **RENUMF©**[1] program, a minimum average reclaiming rate of 22.6 ISNs/hour[2] of completely structured and documented high quality code is achieved. This average restructuring rate includes the time:
  - to restructure the module into an effective set of Functional Blocks
  - to supply an accurate documentation header at the front of the module
  - to document all arguments and all local variables

  The structuring time in hours, $\boldsymbol{T_h}$**,** is related to the number of ISNs in the module at the *start* of the structuring, $\boldsymbol{N_{ISN}}$ , and the number of prior times a similar type module has been structured, $\boldsymbol{N_p}$ , as

  $$\boldsymbol{T_h} = \frac{0.044\boldsymbol{N_{ISN}}}{(1 + \boldsymbol{N_p})}$$

  This result is based on carefully timed studies conducted on 14 modules, as described in the COMP-AID technical report, *Publication of Rates at which Old, Difficult-to-Under-stand FORTRAN Modules Can be Reclaimed by Use of the Functional Block Outline Methodology*, January 22, 1990, 13 pages. By "minimum", we mean that rate at which $\boldsymbol{N_p} = 0$, since as $\boldsymbol{N_p}$ increases the structuring *rate* also increases.

- the *design/implementation/coding/checkout* of the algorithmic representation of a *new* module:

  When used in conjunction with the **RENUMF©** program, an average rate of 12.9 ISNs/hour of completely checked out and documented high quality code is achieved. This average design/implementation/coding/checkout rate includes the time:
  - to design/implement the module algorithm, by using the FBO[3] methodology
  - to code the module, by filling in the completed FBO
  - to complete internal documentation, including local variables
  - to checkout the module (includes time to write the test driver)

  The number of ISNs present in a completed module, $\boldsymbol{N_{ISN}}$ , is related to total time required, $\boldsymbol{T_h}$ , as

  $$\boldsymbol{N_{ISN}} = 12.9\boldsymbol{T_h}$$

---

[1]**RENUMF** is a for-sale program developed by the COMP-AID Company.

[2]"ISN" denotes **I**nternal **S**tatement **N**umber. The size of a module, in effective ISNs prior to the start of reclamation, *excludes* JCL, comment lines, and variable declarations. Moreover, each ISN (which represent a single FORTRAN statement) counts as *one* ISN, whether the statement occupies only a single line or 20 lines.

[3]COMP-AID's **F**unctional **B**lock **O**utline methodology.

This result is based on carefully timed studies conducted on twelve modules, ranging from 68 ISNs (requiring 3.22 hours) to 397 ISNs (requiring 25.15 hours), as described in the COMP-AID technical report, *Use of Functional Blocks in the Design, Coding, and Checkout of FORTRAN modules*, October 2, 1989, 28 pages. The productivity rates achieved are 2 to 5 times as great as those reported by Boehm,[4] based on a study of a large data base of software projects. The range of 2 to 5 is based on Brooks'[5] caution that

> "... a *software product* requires about three times as much effort to complete as an equivalent-sized *personal software program* ...."

Of the 12 modules, six are associated with a complex spooling program, and two are associated with a sophisticated parsing-type program — we should consider both of these programs to be of the "software product" type. We should then consider the remaining four modules to be of the "personal software program" type.

## 2. Schematic Representation of a Functional *Block* of Code

Just as an entire *program* can be decomposed into a set of independent functional modules, so also then can any individual *module* be decomposed into a set of independent functional blocks, where these blocks of code constitute the steps required to accomplish the module task. From a schematic viewpoint, we illustrate this in Figures la through ld on the next page. Figure la represents a single module — either a subroutine or a very short program — which performs a desired function. As an example, say that Figure la represents the very short Prime Number program, whose purpose is to print the list of prime numbers from 1 to 1000.

## 3. Decomposing a Module into Functional Blocks of Code

Next, in Figure lb, we decompose this module into its three parts, or functions:

1. Print header; define initial `I`.
2. Print values of `I` which are prime.
3. Print termination message.

We also enter these three function-descriptions into the functional outline shown in Figure 2 on the next page. Since each of these three blocks shown in Figure lb in turn perform desired functions, we refer to them as *Functional Blocks.* Since these three blocks make up the module, they are then the *major* Functional Blocks. We refer to them as *Level 1* Functional Blocks.

---

[4]Boehm, Barry W. *Software Engineering Economics*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981, pages 57–71.

[5]Brooks, Frederick P., Jr. *The Mythical Man-Month*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1975. Essays on Software Engineering. Pages 4–7.

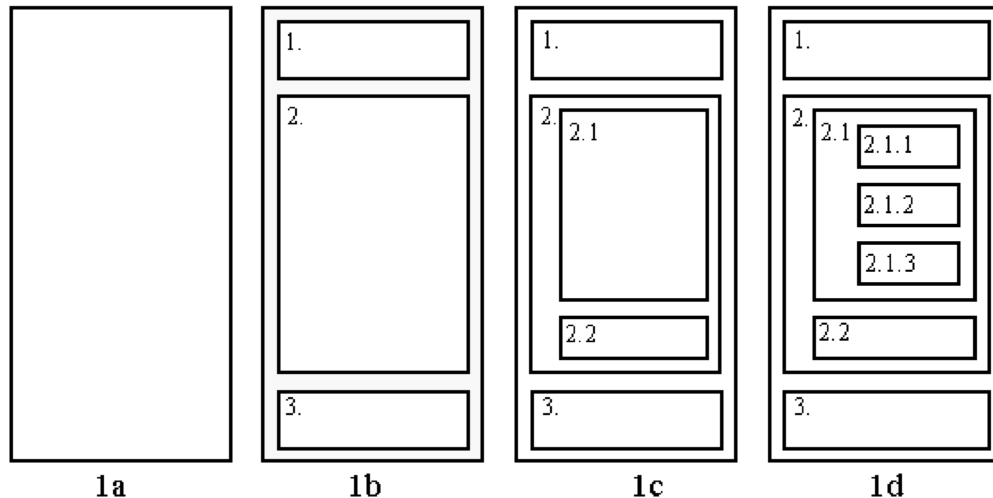


**Figure 1.** Schematic representation of the successive hierarchical decomposition of a module into Functional Blocks.

PRIME NUMBER PROGRAM: Print list of prime numbers from 1 to 1000.

1. <Entry>: Print header; define initial I.

2. <I defined>: Print values of I which are prime.

    2.1 <I within bounds>: If I is prime, print its value.

        2.1.1 <Entry>: Get upper limit MXDVSR for division test.
        2.1.2 <Upper limit MXDVSR obtained>: Perform division test.
        2.1.3 <I not divisible by any divisors>: Print as prime.

    2.2 <Processing for I complete.>: Increment I; re-loop if in bounds.

3. <All processing complete>: Print termination message.

**Figure 2.** Outline of single functions performed at each Level of Functional Block.

## 4. Decomposing a Functional Block into Sub-Functional Blocks

The first and last Functional Blocks in Figure 2 are no more than trivial initialization and termination steps. Obviously, since the non-trivial portion of the module is contained in the second block, we then in turn decompose it into two sub-blocks (cf. Figure lc):

1. <I within bounds>: If I is prime, print its value.
2. <Processing for I complete>: Increment I; re-loop if in bounds.

Since these two sub-blocks make up a Level 1 block (cf. Figure 2), we then refer to them as *Level 2* Functional Blocks.

The text within the angle brackets (e.g, "<Processing for I complete.>") refers to the logical-

state-of-affairs that must be true for control to reach that point. In other words, Step 2.1 tests the current candidate for a prime number (contained in `I`). If `I` is prime, Step 2.1 prints this value. If `I` is not prime, then Step 2.1 passes control directly to Step 2.2. In either case, we see that "Processing for the current `I` is complete" when control reaches that point.

Because the functional description in the Level 2 Block #2.1,

"If `I` is prime, print its value",

is vague, we must once again decompose this block into sub-blocks (cf. Figure ld), which will be the following Level 3 Functional Blocks

2.1.1 <Entry>: Get upper limit `MXDVSR` for division test.
2.1.2 <Upper limit `MXDVSR` obtained>: Perform division test.
2.1.3 <`I` not divisible by any divisors>: Print as prime.

## 5. Preceding each Functional Block by a Single Line of Comment

Entering these three Level 3 functional descriptions into our functional outline in Figure 2, we see that we have a concise *summary* of the module algorithm. If we precede each block of code by its associated functional description shown in Figure 2, then this description-type line of comment, *along with* the code following, constitutes a *complete* documentation of the module algorithm (see Figure 6a on page 16 for an example).

In our concept of Functional Blocks, each Functional Block — regardless of its level — is preceded by a single line of comment of the form,

c <*Event*>*: Process-description,*

where *event* refers to the logical-state-of-affairs that must be true upon entry, and *process-description* refers to the process to be performed. Since each block is preceded by only a *single* line of comment, then the programmer learns to update these single lines, as required, in the process of modifying the code in the respective block following. Moreover, such a visual segmentation of the module code permits the programmer (or user) to go directly to the code of interest.

At times, the single line of comment preceding a Functional Block must be severely abbreviated. In this case, a more complete commentary is placed at the beginning of the module in a *Notes* section, referencing (by outline number) the block in consideration (see Figure 5b on page 13 for an example).

## 6. Statement Nos. within Functional Block Start with Block No.

As you've noted from the functional outline in Figure 2 (or from the schematic representation in Figure 1), each block has an outline number associated with it. When we structure a FORTRAN

module, we start all statement numbers in the 1st block with a *1*, all those in the 2nd block with a *2*, all those in the 3rd block with a *3*, etc. An appropriate multiplier is of course used. For example, using a multiplier of 1000, the statement numbers in Block #1 might be *1000*, *1010, 1020*, etc.; those in Block *#2, 2110, 2112*, *2200*, etc.; and so on.

Starting each statement number with the number of the block containing it is a tremendous aid to a programmer (or user) studying the code. For example, if a programmer reviewing the code in Block #2 sees a `GO TO 5000`, then he or she immediately knows that reference is made to Block #5. Moreover, if a functional outline is available for the module, (like that shown in Figure 2), then the programmer can associate a *logical meaning* with the `GO TO 5000` statement, just as though it were an alpha label. Interestingly, our RENUMF programming aid performs this type of *block-renumbering*, once we have identified to it the beginning and the level of each Functional Block. Moreover, RENUMF also prints a functional outline of the module algorithm, such as that shown in Figure 7 on page 18.

## 7. Functional Blocks are *More* than Separated Blocks of Code

While the separation of a module into blocks of code, with each block containing statement numbers that are a selected multiple of its outline number, is a great asset to a programmer in understanding the various parts of the module algorithm, yet this separation alone is no guarantee of the *independence* of these blocks from each other. And without independence, a proof-of-correctness of the module algorithm is not possible.

To insure the independence of each of these blocks one from another, we require that branching from out of a given block to a separate target block be only to the *beginning* of the target block, with the additional understanding that the target block be at the same — or higher — level as the originating block. We term such branches "proper `GOTO` branches"; and we shall presently see that a proof-of-correctness is readily available for modules coded in this fashion. Illustrations of proper `GOTO` branches are shown in Figure 3 by the *solid* arrows.

On the other hand, a branch out of a given block into the "midst" of the target block (i.e., *not* to its beginning) represents an "improper `GOTO` branch". They are represented by the *dashed* arrows in Figure 3.

## 8. Importance of the Logical Entry Assertion upon Entry into a Functional Block

A Functional Block is very similar to a subroutine or function. For a programmer to use a single-entry subroutine such as `SORT (A,II,JJ)`, which sorts the `A` array into increasing order from `A(II)` to `A(JJ)`, he knows that he must first load the `A` array and define `II` and `JJ`, before calling `SORT`. Defining these three arguments `(A,II,JJ)` meets the entry requirements expected by `SORT`. In other words, it is the logical-state-of-affairs that must be true upon entry

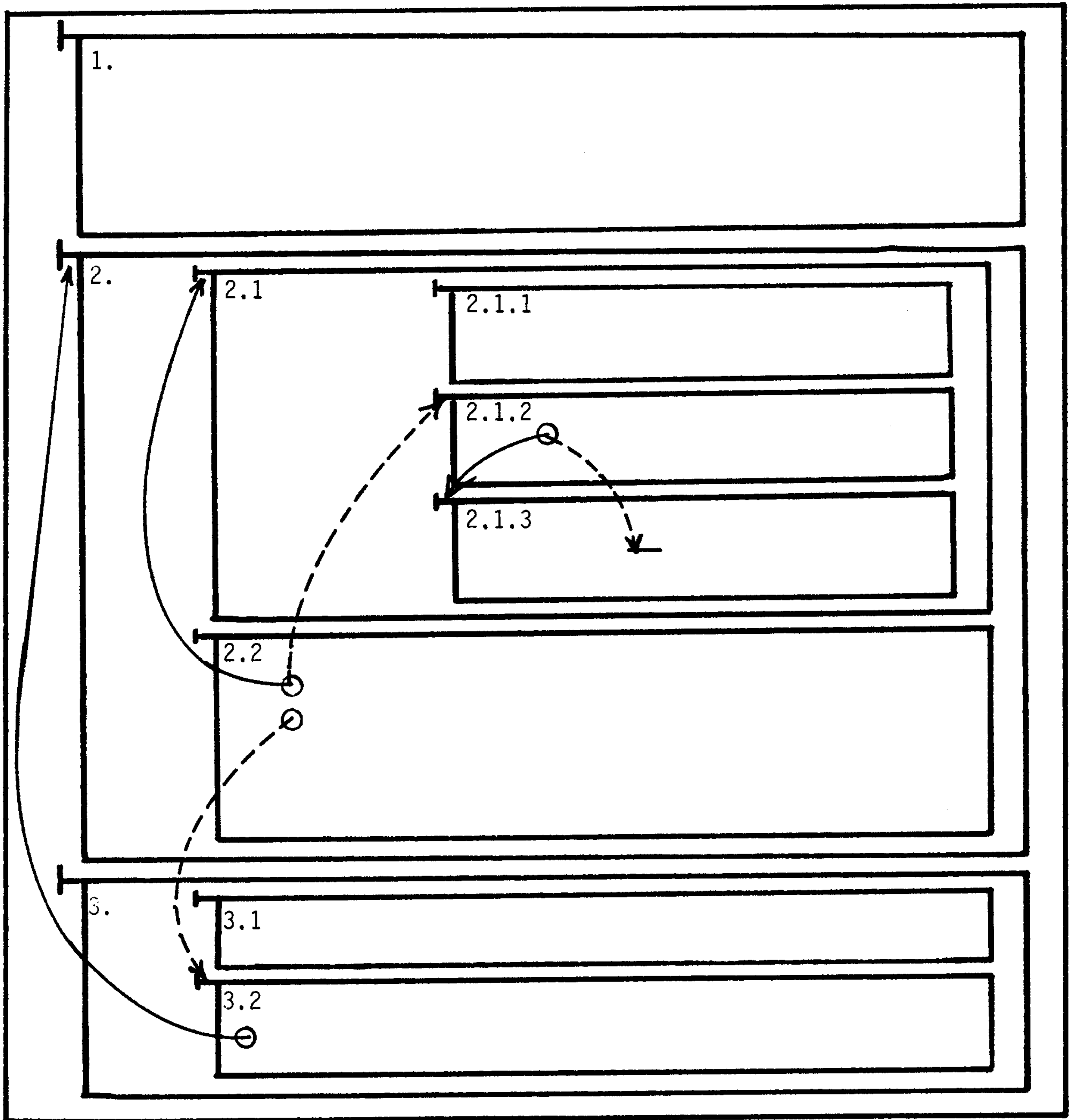


**Figure 3.** Schematic representation of the hierarchical decomposition of a module into Functional Blocks. The overhang at the top left of a given block represents the *valid* entry point (i.e., its beginning) for that block. "Proper `GOTO` branches" are denoted by *solid* arrows, while "improper `GOTO` branches" are denoted by *dashed* arrows. (This schematic representation is for illustrative purposes, and is not necessarily meant to be the same as that shown in Figure 1.)

into `SORT` for the `SORT` subroutine to work correctly. Thus, just as we precede each Functional Block by a line of comment of the form

`C` *<Event>: Process-description,*

we see that the event: process-description for this `SORT` module is

<`A` loaded, and `II` *&* `JJ` defined>: Sort `A` in increasing order from `II` *to* `JJ`.

A Functional Block of code, as we've seen, similarly has a logical-state-of affairs (or *logical assertion* or *predicate)* which should be true upon entry into the block in order for the block to correctly perform its processing. We refer to this logical assertion as the <*event*>. Now the logical assertion which must be true for entry into a certain Functional Block may require that only a *single* variable be defined; while that for entry into another Functional Block may require that an entire *set* of variables be defined. Also, the logical assertion required for entry into a Functional Block may not only assume that one or more variables are defined, but may also assume that one or more desired processes (events) have occurred. For example, the logical assertion associated with the Level 2 Functional Block #2.2 in Figure 2, "<Processing for `I` complete>", assumes

1. that `I` is defined, and
2. that processing for this current value of `I` is complete.

Because processing for the current `I` is complete, Block #2.2 therefore increments `I` to its next odd-numbered value.

## 9. Use of *only* Proper GOTO's Permits a Proof-of-Correctness

Because of the complex interactions in a non-trivial module, it is virtually impossible to understand it *if we view it as a single entity*. However, if we can hierarchically decompose the module into a set of Functional Blocks, in which each Functional Block is *small enough* that we can readily see that it correctly performs its required function if its logical entry-assertion requirements (<*event*> or interface conditions) are correctly met, then we can use these small understandable Functional Blocks as building-blocks to construct increasingly complex, higher level Functional Blocks. As an example, from Figure 2 (or Figure ld) we see that the three Level 3 Functional Blocks #2.1.1, #2.1.2, and #2.1.3 are used to construct the single Level 2 Functional Block #2.1.

We related these three Level 3 sub-blocks to each other during the construction process in such a way that their respective logical entry-assertion requirements were always met. Because of this, the composite Level 2 Functional Block #2.1 is also completely correct, readily understandable, and has its own logical entry-assertion requirement. Moreover, this Level 2 Functional Block #2.1 can now in turn be used as a sub-block, in combination with Functional Block #2.2, to construct the Level 1 Functional Block #2.

As we combined lower-level Functional Blocks to construct increasingly complex, higher-level blocks, we saw that the correctness of the composite block depended upon relating the sub-blocks to each other in such a way that their respective logical entry-assertion requirements were always met.[6] And because the logical entry-assertion for any Functional Block is valid only if entry is made to the *beginning* of the Functional Block, thus we see that one[7] necessary requirement for achieving a module which is amenable to a proof-of-correctness is that *only proper GOTO branches are allowed.*

Obviously, the foundation of Functional Blocks rests upon the assumption that block boundaries be selected such that no improper GOTO branches exist. The availability to the programmer of a programming aid that flags improper GOTO's, once he or she has indicated both the start and the hierarchical level of each of these blocks, is a great convenience in the use of Functional Blocks. And our RENUMF programming aid does this very thing.

## 10. Improper GOTO's Can Occur in *Two* Ways

One way, which we've not yet discussed, in which an improper GOTO can occur, is due to the so-called Spaghetti-Bowl syndrome, in which a programmer attempts to utilize a common chunk of code by branching out of the present block into the midst of the external block where this chunk of code is already located, rather than simply duplicating the chunk of code in the present block.

Because these two blocks are now coupled together, they are no longer independent. Thus a subsequent modification to one could affect the other. When the Spaghetti-Bowl syndrome is widely practiced, the code within a module becomes so tightly coupled that a “simple” modification anywhere in the code can result in the need for extensive re-coding of the remainder of the module — the so called “ripple effect”.

Cases may occur, of course, in which the chunk of code to be shared is large enough that we do not want to duplicate it in each block of use. We do point out, on page 4 of our companion brochure, “Suggested Eight-Step Iterative Process for Structuring FORTRAN Modules”, that there are three different ways in which identical sequences of code can be separated out in a *structured* (or *refactored*) fashion. By *structured* we mean:

1. that there are no improper GOTO's
2. that the “commonality” of the shared code is once again highly elucidated

Because sharing code in a structured sense avoids the Spaghetti-Bowl syndrome (and thus improper GOTO's), all Functional Blocks are once again independent. Hence the effect of modifications to any one block will be confined *to that block alone.*

---

[6]Tausworthe formalizes and exemplifies the proof-of-correctness of modules: Robert C. Tausworthe, *Standardized Development of Computer Software* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977), 287–306.

[7]A second requirement, as we point out in our COMP-AID brochure, “A Discussion of Non-Unique <*Event*>'s”, is that each of those entry logical-assertions (events) must be *unique*.

## 11. Improper GOTO's Can Occur when Block Boundaries Are Incorrectly Chosen

The second way, as we've previously discussed, in which an improper GOTO branch can occur is due to the improper selection of the boundaries of the Functional Blocks. For example, if a branch out of Block #5 goes into the "midst" of Block #2, and it's not for the purpose of simply sharing some common code, then two possibilities exist:

1. either Block #2 is not a single Functional Block, as we previously thought
2. the module algorithm, as presently implemented, is incorrect

The second possibility is allowable only if we are working with in-house code. We disallow it when working with a client's code, because we must *always* assume that the algorithm implemented in the client's code *is correct.* As a result, our structuring of a client's code results in the module algorithm, as it is *presently* implemented, becoming highly visible and clearly elucidated. Thus, because the complex inter-workings of the module algorithm are so clearly laid out before the client, he or she will now be able to:

1. verify whether the present module algorithm is correct
2. look for ways in which the present algorithm can be improved

(Of course, should we encounter what appears to be an obvious algorithm error, we shall discuss this with the client over the phone. If the client agrees the algorithm is in error, and directs us to correct it *as he specifies,* then we shall do so, carefully noting this modification, both in the “History” section of the structured module and in the report accompanying the structured module.)

We realize that an improper GOTO branch into the midst of a block of code is made under a logical assertion which is *different from* the logical entry-assertion for the block. This indicates two things:

1. that the inter-relation between the various blocks is still unclear
2. that the block in question performs more than a single function

Breaking this block into two parts, in which the new block now has its logical entry-assertion requirement clearly specified, thereby eliminates the improper GOTO branch.

## 12. Selecting the *Optimum* Set of Functional Blocks

As we point out in our brochure, “Suggested Eight-Step Iterative Process for Structuring FORTRAN Modules”, selecting a set of blocks of code, in an *existing* module, in which all improper GOTO branches are eliminated is an iterative process, usually requiring several cycles for successful completion. Of course, if we are working on our own in-house code, or you are working on your own in-house code using our RENUMF programming aid, then we or you have an excellent idea of the location and extent of each of the Functional Blocks. In this situation we could well achieve a properly structured module in a single pass.

The situation is vastly different, of course, when we are working on a completely unfamiliar out-of-house module. After a preliminary study of the module, we insert header comments[8] to indicate to RENUMF where we feel each Level 1 block starts. RENUMF will then print out all improper GOTO's which may exist. The elimination of those improper GOTO's which are due to the Spaghetti-Bowl syndrome are handled differently from those that result from the improper selection of the Functional Block boundaries.

Once we have selected a set of Level 1 (major) Functional Blocks in which no improper GOTO's exist, we can see if the process-clarity could be improved by in turn segmenting any of these into Level 2 sub-blocks. If so, then the process is repeated for these newly introduced Level 2 sub-blocks. (Of course, once all Level 2 blocks are free of improper GOTO'S, we may want to further decompose some of these into Level 3 sub-blocks, again necessitating that the process be repeated; etc.)

There is much more to structuring a module than just selecting a set of Functional Blocks in which all *improper* GOTO's are eliminated. Eliminating improper GOTO's does guarantee on the one hand two accomplishments:

1. that the set of Functional Blocks chosen are highly independent one from another
2. that the inter-relation of their associated <*event*>'s (entry logical assertions) are highly visible within the specified hierarchical structure.

Yet, on the other hand, this act of eliminating improper GOTO's cannot *in itself* resolve certain other design-related questions, e.g. —

- Are the one-line comments preceding each block correct?
- Are the associated <event>'s for each block unique?
- Could process-clarity be increased by in turn segmenting any given-level block into sub-blocks (hierarchical decomposition)?
- Could process-clarity be increased by chunking together two or more adjacent given-level blocks (hierarchical composition)?
- Do identical lines of code appear in two or more blocks, which could be separated out, using permissible structuring techniques?

When we structure a module for you, we of course also accomplish each of these five design-related steps. Having done this, we consider that this final set of Functional Blocks is optimum.

## 13. The Functional Block - An organizational Concept

Therefore we see that the Functional Block represents an *organizational* concept. It permits us to describe the complex inter-workings of any given module in terms of independent, single-function blocks of code, in which the inter-relation of these various blocks to each other becomes

[8] The insertion of the headers preceding the various levels of Functional Blocks is accomplished with the respective *single* keystrokes, using macros within a source program editor. In other words, the programmer is not required to type in *all* the characters making up any given header; only the appropriate single keystroke is required. Moreover, the programmer can modify the format of the headers which identifies them to to RENUMF.

highly visible. It is as Prof. Donald E. Knuth notes:[9]

> "We understand complex things by systematically breaking them into successively simpler parts and understanding how the parts fit together locally."

## 14. An Example

We shall now conclude with an example of structuring by employing it on a simple Prime Number Program, showing how it appeared *before* we structured it and *after* we structured it into Functional Blocks.

The prime number program, as it appears before structuring, is shown in Figure 4 below. Interestingly, this program module is shown *exactly*[10] as it appeared on pages 93–94 of the *IBM System/360 Operating System FORTRAN IV (G and H) Programmer's Guide*, Form C28-6817-1 (Second Edition, July 1969).

```
     Directory: C:\extract                                 Input file: pnbefore.for
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)   RENUMF I.6.2, Dec 09, 1997
     LISTING OF SOURCE DECK "Prime Number Program      "   01/28/00      11:01:29   PAGE   1

     C     NAME='Prime Number Program', NORENUM, NOI
     C     PRIME NUMBER PROBLEM
  1    100 WRITE (6,8)
  2      8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
          119X,1H1/19X,1H2/19X,1H3)
  3    101 I=5
  4      3 A=I
  5    102 A=SQRT(A)
  6    103 J=A
  7    104 DO 1 K=3,J,2
  8    105 L=I/K
  9    106 IF(L*K-I)1,2,4
 10      1 CONTINUE
 11    107 WRITE (6,5)I
 12      5 FORMAT (I20)
 13      2 I=I+2
 14    108 IF(1000-I)7,4,3
 15      4 WRITE (6,9)
 16      9 FORMAT (14H PROGRAM ERROR)
 17      7 WRITE (6,6)
 18      6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
 19    109 STOP
 20        END
```

**Figure 4.** Sample FORTRAN module used to illustrate the method of structuring into Functional Blocks, as it appears *before* structuring.

---

[9] Donald E. Knuth, "Structured Programming with go to Statements", *Computing Surveys* **6**, 4 (December 1974), 291.

[10] Well, *almost* exactly, since we inserted the first card, which is a "control card option list" card used to tell the RENUMF program to not renumber the statement numbers (NORENUM), and to not indent (NOI) any statements (e.g., the contents of a DO-loop). That way we used RENUMF to place the ISN value by each statement, but without otherwise modifying the original appearance of the module.

After structuring, it appears as shown in Figure 5a below.[11]

```
     Directory: G:\RENUMF\internet                          Input file: PRIME.for
           Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)     RENUMF 2.2.0, Apr 16, 2001

     LISTING OF SOURCE DECK "Prime Number Program        "   05/22/04   19:49:41   PAGE 1

     C     NAME='Prime Number Program', FORMAT=TRUNCATE
     C
  1        PROGRAM PRIME
     C
     C
     C*********************************************************************
     C(01) <Entry>: Print header; define initial I.                       *
     C*********************************************************************
  2        WRITE (6,1010)
  3   1010 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000 /
          1   19X,1H1 / 19X,1H2 / 19X,1H3)
  4        I = 5
     C
     C
     C*********************************************************************
     C(02) <I defined>: Print values of I which are prime.                *
     C*********************************************************************
  5        CONTINUE
     C
     C
     C     ----------------------------------------------------------------
     C     <I within bounds>: If I is prime, print its value.
     C     ----------------------------------------------------------------
  6   2100 CONTINUE
     C
     C       ===<Entry>: Get upper limit of J for division test.
  7           A = I
  8           A = SQRT(A)
  9           J = A
     C
     C       ===<Upper limit J obtained>: Perform division test.
 10           DO 2121 K = 3, J, 2
 11              L = I/K
 12              IF (L*K-I) 2121, 2200, 3000
 13   2121    CONTINUE
     C
     C       ===<I not divisible by any divisor>: Print as prime.
 14           WRITE (6,2131) I
 15   2131    FORMAT (I20)
     C
     C
     C     ----------------------------------------------------------------
     C     <Processing for I cmplt.>: Increment I; re-loop if in bounds.
     C     ----------------------------------------------------------------
 16   2200 I = I + 2
 17        IF (1000-I) 4000, 3000, 2100
     C
     C
     C*********************************************************************
     C(03) <Program error detected>: Print message indicating error; drop.  *
     C*********************************************************************
 18   3000 WRITE (6,3010)
 19   3010 FORMAT (14H PROGRAM ERROR)
     C
     C
     C*********************************************************************
     C(04) <All processing complete>: Print termination message.          *
     C*********************************************************************
 20   4000 WRITE (6,4010)
 21   4010 FORMAT (31H THIS IS THE END OF THE PROGRAM)
 22        STOP
 23        END
```

**Figure 5a.** The Prime Number module in Figure 4, as it appears *after* structuring.

[11] RENUMF outputs both a new *source* file of the processed code and a *print* file of the processed code. The file in Figure 5a is the *print* file version, in which the ISNs are shown to the very left of each FORTRAN statement.

But we are not through yet, as we now need to supply the "header" documentation. By this, we do not mean the text within the comment lines preceding each Functional Block. Rather, we mean a header section at the beginning of the module, such as shown in Figure 5b below.

```
* Abstract ***[01/29/1980]*******************************************
*   This program prints the list of prime numbers from 1 to 1000.
*
* Keywords
*   Prime, 1000
*
* Purpose
*   The PRIME NUMBER program prints the list of prime numbers from
*     1 to 1000.
*
* Arguments
*   None
       ●
       ●
       ●
*
* Modules called: SQRT
*
* Databases (associated data sets), or Files accessed:
*   Data base: None
*   Files:     None
*
* Errors
*     Block #  Type (W/F)  Description
*     -------  ----------  --------------------------------------------
*     2.1.2    Fatal       L*K > I, which is impossible due to L=I/K.
*     2.2      Fatal       I=1000, which is impossible, since I is odd.
*
* Notes
*
*   Block #2.1.2.  Clarification of Process Description.
*       Odd numbers are used as divisors (even though some are not
*       prime) to determine if the candidate number 'I' is not prime.
*       Only odd numbers K<=SQRT(I) need be used, as explained
*       below.
*
*   Block #2.1.2.  Clarification of Entry Assertion.
*       If 'I' is not prime, then there will always be an odd prime
*       number less than or equal to SQRT(I) by which 'I' is exactly
*       Divisible.  For example, when I=35, then it is divisible by
*       K=5.  Only when 'I' equals the next perfect square (49) is
*       the next value of K=7 required (since 49 = 7*7).
*
* History
*     [Key: IBM = Manual: "IBM System/360 Operating System FORTRAN IV
*                          (G and H) Programmer's Guide,
*                          Form C28-6817-1 (Second Edition, July 1969).
*           RCW = Ronald C. Wackwitz]
*   Original , 07/01/1969, #01, IBM, Original version.
*   ReFactor , 01/29/1980, #02, RCW, Structured original version into
*                                    Functional Blocks using RENUMF.
*
* End ***************************************************************
C
C
C     ------------------------------------------------------------------
C     Local variables.
C     ------------------------------------------------------------------
C      Name   Size  Type  Description
C     ------ ------ ----- ---------------------------------------------
C     A             R     Real value of current I required by SQRT of I
C     I             I     Value of current candidate No. being tested
C     J             I     Max DiViSoR (3,5,7,...,J), in div. test
C     K             I     Integer value of Current odd DiViSoR
C     L             I     Integer ratio of I/K+
```

**Figure 5.b** This header section follows the **PROGRAM PRIME** statement in Figure 5a above.

Four suggested points regarding this initial header documentation are:

1. The header documentation should summarize important information regarding the associated module. As shown above, it includes
   - a one-line abstract
   - keywords
   - a purpose
   - list and description of arguments (if a subroutine or function)

     •
     •
   - list of errors
   - notes regarding algorithms employed in various sections of the code
   - a history section
   - description of local variables
2. Obviously, the format of this header section is *arbitrary*, depending upon the requirements as set forth by each individual IT section. But once a format is defined, then all programmers within that section should abide by it, so that the initial portion of every header section is machine readable.
3. We would suggest that the following three items *always* be included in the header:
   • Errors
   • History
   • Description of local variables
   Moreover, Notes describing algorithms employed should also be included, if necessary.
4. The header documentation does not necessarily need to be within the FORTRAN module itself (within the `PRIME.FOR` module in this case). Rather, it can be within a separate file (e.g., within the `PRIME.HDR` file in this case). However, if the header section is not overly long, then we personally like to include it within the FORTRAN module itself, right after the `PROGRAM`, `SUBROUTINE`, or `FUNCTION` statement.

## 15. Can the Present Algorithm be Improved?

The term "refactoring" implies that *no change* in the module's functionality occurs in the process of clarifying the module code. We can certainly see that this is the case for code shown in Figure 5a, as compared with the initial code in Figure 4, for we have done nothing more than separate the code into its various independent set of Functional Blocks.

But our term "Functional Block" signifies much more than just an increase in clarity. It signifies that each block is functionally *isolated* from its neighbors, since

• only "proper `GOTO` branches" are used
• all logical entry assertions are *unique*

Now that we know what each block of code is supposed to do, we can look for ways to improve the implementation of the algorithm within that block, without worrying about affecting neighboring blocks in the process. We shall illustrate this on two segments of the code.

First, consider the code in Block #2.1.1 in Figure 5a, in which the upper index `J` used in the division tests is computed:

```
A = I
A = SQRT(A)
J = A
```

Notice how these can be combined into the single statement,

```
        J = INT(SQRT(FLOAT(I)) + 0.00001),
```
which not only is more efficient , but more importantly also improves upon the *visualization* of how J is actually computed.

We added the binary round-off factor of 0.00001 out of habit. Admittedly, some machines compute single precision values sufficiently accurately that they do not need the round-off added. But back in 1980 that was not true of all machines. For example, we ran IBM's program, exactly as shown in Figure 4, on a Data General Nova 840 in DG's FORTRAN 5. This code failed in three places in the computation of J, thereby printing 49, 289, and 961 *as prime*, when in fact they are all perfect squares.

Because the K index in the subsequent Block #2.1.2 is incremented by 2 every time, we can recast the above definition of J as
```
        J = NINT(SQRT(FLOAT(I)))
```
While this will result in J being calculated as an *even* number at times (e.g., when I=13, J=4), yet since the subsequent value of the K index of 5 will be greater than J=4, then the DO loop,
```
        DO 2121 K = 3, J, 2
```
will stop after using K=3. In other words, K will never go too high.

Secondly, considering the two statements in Block #s 2.1.2 and 2.2 respectively,
```
        IF (L*K-I) 2121, 2200, 3000
        IF (1000-I) 4000, 3000, 2100
```
we see in both cases that the branch to Block #3 can never occur, since — respectively —
```
        L*K ≤ I
```
and

        I is always an odd number.

We term this branch to Block #3 the "Arithmetic-IF syndrome, in which the programmer had to supply *some* third statement number even when only one was required. Thus, the fictitious Block #3 can be removed (with the old Block #4 becoming the new Block #3), and the two Arithmetic IF's are replaced by Logical IF's as respectively follows
```
        IF (L*K .EQ. I) GO TO 2200
```
[12]
```
        IF (I .LT. 1000) GO TO 2100
```

Proceeding in this fashion, we arrive in Figures 6 as the final version of the structured form of IBM's original program shown in Figure 4. Please note that this structured version shown in Figure 6a still employs the algorithm used by IBM in Figure 4; we've just clarified it and updated it. Now see how the importance of the History item within the initial header section looms important (see Figure 6b), as it documents each and every change made. Also, please note that we have changed the names of the variables employed to more meaningful mnemonics.

---

[12]Furthermore, the two statements in Block #2.1.2,
```
        L = I/K
        IF (L*K .EQ. I) GO TO 2200
```
can be replaced by the single (and more visually explicit) statement,
```
        IF (MOD(I,K) .EQ. 0) GO TO 2200
```

```
    Directory: G:\renumf\internet                         Input file: prime.for
    Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)    RENUMF 2.2.0, Apr 16, 2001
    LISTING OF SOURCE DECK "Prime Number Program        "  05/29/04   17:00:50   PAGE   1


    C      NAME='Prime Number Program', FORMAT=TRUNCATE
    C
 1         PROGRAM PRIME
    C~
    C
    C*****************************************************************
    C******Declarations.                                              *
    C*****************************************************************
 2         IMPLICIT NONE
 3         INTEGER I, MXDVSR, ICDVSR
    C
    C
    C*****************************************************************
    C(01) <Entry>: Print header & first three primes; initialize I to 5.   *
    C*****************************************************************
 4         WRITE (6,1010)
 5    1010 FORMAT (1H 'Following is a list of prime numbers from 1 to 1000:'/
          *        1H 18X '1' /
          *        1H 18X '2' /
          *        1H 18X '3' )
 6         I = 5
    C^
    C
    C*****************************************************************
    C(02) <I defined>: Print values of I which are prime.             *
    C*****************************************************************
 7         CONTINUE
    C
    C
    C      -----------------------------------------------------------
    C      <I within bounds>: If I is prime, print its value.
    C      -----------------------------------------------------------
 8    2100 CONTINUE
    C
    C        ===<Entry>: Get max no. for division tests, MXDVSR=SQRT(I).
 9              MXDVSR = NINT(SQRT(FLOAT(I)))
    C
    C        ===<MXDVSR defind>: Do div. tests; -> #2.2 if zero remainder.
10              DO 2121 ICDVSR = 3, MXDVSR, 2
11                 IF (MOD(I,ICDVSR) .EQ. 0) GO TO 2200
12    2121      CONTINUE
    C
    C        ===<I not divisible by any divisor>: Print as prime.
13              WRITE (6,2131) I
14    2131      FORMAT (1H  I19)
    C
    C
    C      -----------------------------------------------------------
    C      <Processing for I complete>: inc. I by 2; ->#2.1 if in bounds.
    C      -----------------------------------------------------------
15    2200 I = I + 2
16         IF (I .LT. 1000) GO TO 2100
     C
     C
     C*****************************************************************
     C(03) <All through>: Print termination message.                  *
     C*****************************************************************
17         WRITE (6,3010)
18    3010 FORMAT (1H0 'This is the end of the program.')
19         STOP ' '
20         END
```

**Figure 6a**. The *final* iteration of the Prime Number module, as it appears after structuring.

```
*
* Abstract ***[11/23/1991]*********************************************
*   This program prints the list of prime numbers from 1 to 1000.
*
* Keywords
*   Prime, 1000
                              ●
                              ●
                              ●
*
* Errors
*   None
                              ●
                              ●
                              ●
*
* History
*     [Key: IBM = Manual: "IBM System/360 Operating System FORTRAN IV
*                          (G and H) Programmer's Guide,
*                          Form C28-6817-1 (Second Edition, July 1969).
*           RCW = Ronald C. Wackwitz]
*   Original , 07/01/1969, #01, IBM, Original version.
*   ReFactor , 01/29/1980, #02, RCW, Structured original version into
*                                    Functional Blocks using RENUMF.
*   Clarify  , 01/29/1980, #03, RCW,
*              1. In Block #2.1.1, replace the three statements,
*                     A = I
*                     A = SQRT(A)
*                     J = A
*                 by the single statement,
*                     J = NINT(SQRT(FLOAT(I)))
*                 to improve both efficiency and visualization.
*              2. In Block #2.1.2, modify the statement,
*                     IF (L*K - I) 2121, 2200, 3000
*                 to
*                     IF ((L*K - I) .EQ. 0) GO TO 2200
*                 in which the Arithmetic IF is replaced by a logical
*                 one.  In the process, the impossible branch to the
*                 error flagging at 3000 can be eliminated, Block #3
*                 itself can therefore be eliminated, so that the cur-
*                 rent Block #4 now becomes the new Block #3.
*              3. In Block #2.1.2, replace the two statements,
*                     L = I/K
*                     IF ((L*K - I) .EQ. 0) GO TO 2200
*                 by
*                     IF (MOD(I,K) .EQ. 0) GO TO 2200
*                 again to improve both efficiency and visualization.
*   Clarify  , 11/23/1991, #04, RCW, Changed names of following vari-
*                                    ables,for the sake of clarity:
*                                          Old     New
*                                         ------  ------
*                                         J       MXDVSR
*                                         K       ICDVSR
*   Control  , 11/23/1991, #04, RCW, Added the IMPLICIT NONE statement
*                                    in order to enforce variable typing
*   Document , 11/23/1991, #04, RCW, Added this header section, and
*                                    documentation of local variables.
*
* End ******************************************************************
C
C
C     -----------------------------------------------------------------
C     Local variables.
C     -----------------------------------------------------------------
C      Name   Size  Type  Description
C     ------ ------ ----- --------------------------------------------
C     I             I     Value of current candidate No. being tested
C     ICDVSR        I     Integer value of Current odd DiViSoR
C     MXDVSR        I     MaX DiViSoR (3,5,7,...,MXDVSR) in div. test
```

**Figure 6b**. This header section follows the **`PROGRAM PRIME`** statement in Figure 6a above.

# 16. In Conclusion

The use of the Prime Number program as an example for structuring a module is taken from Chapter III of our 84 page COMP-AID technical report, *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*, February 1980, pages 11–48. Figure 6a of this overview occurs on page 42 of our report.

Many details are accordingly left out in this overview, describing the transition from Figure 4 (the *before*) to Figure 6a (the *after*). But our point was just to exemplify the concepts we have presented regarding Functional Blocks. Therefore, please note the following four points regarding the structured version in Figure 6a:

1. All statement numbers are now a multiple (using *1000* as the multiplier) of the Functional Block in which they occur. This was done *automatically* by RENUMF.
2. All branches are "proper GOTO branches", as *verified* by RENUMF.
3. All logical entry assertions which can be arrived at by *more* than one way are *unique*,[13] regardless of how they are arrived at. For example, with regard to the logical entry assertion in Block #2.1, "<I within bounds>", this assertion is true whether we fall down to it from Block #1 above, or branch back to it from Block #2.2 below. And with regard to the logical entry assertion in Block #2.2, "<Processing for I complete>", we have already discussed the uniqueness of this case at the top of page 4.
4. We can now see how our initial "Functional Block Outline" for the prime number program presented in Figure 2 on page 3 summarizes the algorithm employed. The programmer has only to place the appropriate code beneath each Functional Block to arrive at the completed implementation of the module, as now shown in Figure 6a.

RENUMF even gives back the present Functional Block Outline employed, as shown in Figure 7 below.[14]

```
-  OUTLINE OF "EVENT: PROCESS-DESCRIPTIONS"      05/29/04      17:00:50   PAGE   7
       IN SOURCE MODULE: Prime Number Program


   1. <Entry>: Print header & first three primes; initialize I to 5.   *


   2. <I defined>: Print values of I which are prime.                  *

         2.1 <I within bounds>: If I is prime, print its value.
                2.1.1 <Entry>: Get max No. for division tests, MXDVSR=SQRT(I).
                2.1.2 <MXDVSR defind>: Do div. tests; -> #2.2 if zero remainder.
                2.1.3 <I not divisible by any divisor>: Print as prime.

         2.2 <Processing for I complete>: Inc. I by 2; ->#2.1 if in bounds.


   3. <All through>: Print termination message.                        *
```

**Figure 7.** Functional Block Outline generated by RENUMF for module in Figure 6a.

[13]The verification of this presently has to be done manually by the analyst, since RENUMF in itself cannot currently verify the uniqueness of the various logical entry assertions.

[14]RENUMF also supplies a cross reference of the elements of the FORTRAN syntax, of the variables, and of the statement numbers, along with any errors detected.