EFFICIENT FORTRAN PROGRAMMING PRACTICE

by A. S. MORRIS and L. S. GRAY

RESEARCH REPORT NO. 182

This paper consists of printed notes given as the second lecture of a two-part special lecture course on efficient programming. Emphasis is placed on structured programming and the implementation of this disciplined approach to programming using Fortran 77.

Dept. of Control Engineering
University of Sheffield
Mappin Street
Sheffield S1 3JD

EFFICIENT FORTRAN PROGRAMMING PRACTICE

1. Introduction

The first part of this 'efficient programming' course covered the functioning of the operating system, and explained how such factors as program size and input/output timing affect overall system efficiency and speed. You should now be in a position to appreciate the benefits in program éxecution speed which can accrue from reducing program size and re-organising how program input/output is carried out.

Such improvements in execution speed of a program can be termed the direct way of improving computer system efficiency. However, there is also an indirect path to improved system efficiency through education of users to program in a structured manner, thereby minimising compilation errors and logic errors during program development. If the time spent in program development can be reduced, the benefit will be a reduction in overall computer loading and corresponding improvement in computer performance for everyone.

I shall therefore cover this part of the course in two parts. Firstly, I shall explain what structured programming is all about and why all modern computing languages are designed to encourage it. I shall explain how the features of Fortran 77 can be exploited to write programs in this disciplined manner. At this stage, I shall have covered what I term the indirect path to efficient computer usage, that of minimising program development time. The second part of this lecture will be an explanation of the direct means of improving system efficiency. The main attack will be on program size reduction with some secondary consideration of input-output planning. The language discussed will be Fortran 77 although similar principles can be applied to other languages.

2. Structured Programming

Structured programming is all about writing programs in a concise way which removes as many sources of error from the program as possible. Such a manner of programming minimises both compilation errors and logic errors in program execution. This results in reduced program development times which has a beneficial effect in decreasing the average loading level on the whole computer system, besides making the programmer more productive.

Structured programming is essentially a code of practice in program design. A useful analogy can be drawn between the design principles of a new car and the design of a new computer program. The same principles of good engineering design practice apply to each.

Let us consider this analogy, and compare the steps taken in car design with the equivalent steps in computer program design. The program is a simple one to input some data, manipulate it, and output it.

	CAR	PROGRAM
Step 1:	Split design into components	
	Engine	Input
	Transmission	Manipulation, stage 1
	Brakes	Manipulation, stage 2
	Suspension	Manipulation, stage 3
	Body	Output
Step 2:	Design each component	
	Make sure each component will fit together with other components without problems	Make sure all program modules will fit together without problems
Step 3:	Document	
	To facilitate future maintenance of car	To facilitate future maintenance and/or development of program

We are now at a stage where we can write down some general rules about structured programming.

Rule 1

The algorithm to be programmed should be split up into as many separately defined steps as possible. The execution of the program via these steps should be well-planned before any code is written. Flowcharts are a very good aid to this.

Rule 2

Each of these separately identified steps should be programmed as a self-contained module. Each module should communicate with a main (control) unit of the program by a number of well-defined parameters. By making each module independent except for well-defined communication parameters, any problems of fitting different modules together are avoided.

Rule 3

Each module should be written in such a way that its purpose and the manner in which it functions are clear. It should be well-documented within the program.

Rule 4

The flow of control in the program should be constrained to well-defined forms of looping and conditional branching such as

```
REPEAT          WHILE          IF       THEN
   .               .           ELSE IF
   .               .
   .               .           ELSE
UNTIL           DO             ENDIF
```

Note

ONLY THE LAST OF THESE STRUCTURES IS AVAILABLE IN FORTRAN 77

Rule 5

Any arbitrary change of control such as allowed by GOTO statements (and computed GOTO), logical IF statements, etc., should be avoided. (This is not always possible, especially using Fortran).

Rule 6

Everything should be rigidly defined rather than using default values. e.g., the TYPE of all variables and arrays should be defined.

2.1 Structured Programming Languages

A structured programming language such as Pascal is one which encourages programs to be written in a structured way by providing well-structured control architecture. As far as possible, no unstructured features are available and so writing in an unstructured way is made very difficult. However, the techniques of structured programming do not limit one to using a structured language. Structured programming is a code of practice which can be applied to produce reasonably well-structured programs even when using a programming language such as Fortran which is fundamentally unstructured. All that is necessary is more care in writing a program in Fortran as the language does not constrain you to write in a structured way. Whereas Pascal forces writing in a structured way, Fortran allows you to choose to write in a structured way but does not force you to do so. In this respect, Fortran 77 (Perkin-Elmer Fortran 7) is better than Fortran 4 in being somewhat more structured.

2.2 Structured features of Fortran 77

The major advance of Fortran 77 for programmers wishing to write in a structured way is the IF, ELSE IF, ELSE, ENDIF construct which are together described as a BLOCKIF block.

For example, consider character string variables COLOR and HUE which can be set equal to various string values, and an appropriate subroutine is to be called depending on the values (note - character string handling is a new feature of Fortran 77).

```
IF(COLOR.EQ.'RED') THEN
          CALL RED
ELSE IF(COLOR.EQ.'WHITE')THEN
          CALL WHITE
ELSE IF(COLOR.EQ.'BLUE')THEN
          IF(HUE.EQ.'LIGHT')THEN
               CALL GREEN
          ELSE
               CALL BLUE
          ENDIF
ELSE
          CALL ERROR
ENDIF
```

This example illustrates many points about the BLOCKIF construct. Each BLOCKIF construct consists of an IF statement, an ENDIF statement and optionally an ELSE statement plus any number of ELSEIF statements. Each IF statement must be matched with an ENDIF statement. A BLOCKIF statement can be nested. In this example, there is an inner BLOCKIF and an outer BLOCKIF.

Indentation and spaces can be freely used to aid program readability. All such spaces are ignored by the compiler. Thus, the above section could be written as follows and would mean exactly the same to the computer.

```
IF(COLOR.EQ.'RED')THEN
CALL RED
ELSEIF(COLOR.EQ.'WHITE')THEN
CALL WHITE
ELSEIF(COLOR.EQ.'BLUE')THEN
IF(HUE.EQ.'LIGHT')THEN
CALL GREEN
ELSE
CALL BLUE
ENDIF
ELSE
CALL ERROR
END IF
```

The other way by which **Fortran 77** has been made more of a structured language than Fortran 4 is through a tidying-up operation whereby certain 'loopholes' have been plugged. Many circumstances can arise in Fortran 4 where, because some statement is used 'outside the rules', the action at execution time is unpredictable. Fortran 77 has tightened up the rules so that such breaches are thrown out at compilation time and the resulting confusing logic errors at execution time are avoided. For example, whereas Fortran 4 allowed the transfer of control into the range of a DO loop, this is forbidden in Fortran 77. Appendix H of the Perkin-Elmer Fortran VIID Reference Manual gives a full coverage of the new features of Fortran 77 and the differences over Fortran 4.

Modularity is implemented in Fortran by programming each of the identified program steps as a function or subroutine. The flow of control through the modular subroutines is controlled by the main segment of the program. Programming each module of a program as a Fortran subroutine conforms precisely with the structured programming rule that each module should be independent. Interaction with other modules is limited to only those parameters defined in the subroutine CALL statement.

When a program consists of several such modules, it is good practice to test the functioning of each module separately. As the operation of each module is verified, it should be stored in semi-compiled form in a user library. This avoids the wasted time in repeatedly compiling validated subroutines along with those still under development. Such modular testing is an important principle of structured programming.

2.3 <u>Other new useful features of Fortran 77</u>

<u>DO loop</u>

One particularly useful feature of the standard language is that DO loop parameters can be both real and negative. For instance we can have a DO loop as follows:

```
      DO 1 A=2.3,-1.5,-0.1
      :
      :
      1  CONTINUE
```

This will execute the loop first with A set to 2.3. On the next loop execution A will be set to 2.2 and loop execution will continue with A decrementing in steps of 0.1 until it is equal to -1.5.

Note

The DO loop is an example of a situation where a GOTO statement is permissible within the rules of structured programming. It is a useful way of terminating a DO loop when a certain condition, e.g., an error criterion, is satisfied. However, the target of the GOTO statement should be the line immediately following the DO loop - it should not 'go' anywhere else.

```
e.g. C  EVALUATE EXP(X) BY POWER SERIES
          EXPX=1.0
          A=1.0
          DO 1 C=1.0,100.0
          A=A*X/C
          EXPX=EXPX+A
          IF(ABS(A).LT.1E-10)GOTO 2
      1   CONTINUE
      2   CONTINUE
      C  THIS IS THE END OF THE DO LOOP
```

Character Strings

Character strings can be handled and manipulated in the same way as numeric variables. A special operator for string concatenation is available.

Character string variables are declared in a CHARACTER type statement which defines the maximum length in characters of each character variable.

```
e.g.      CHARACTER*5 A,B
          CHARACTER*4 C
```

declares two string variables A and B each of length 5 characters (5 bytes) and a string variable C of length 4 characters.

Initial values can be given to character string variables in three ways; DATA statement, READ statement, assignment statement.

```
e.g.      DATA A/'TITLE'/
          READ(1,2)B
      2   FORMAT(A5)
          C='PQRS'
```

Character string variables can be handled like ordinary variables in many circumstances, e.g., logical expressions:

```
          IF(B.EQ.'TITLE')CALL OUTPUT
```

Arithmetic operators do not have any meaning with string variables, but there is a special concatentation operator //.

```
        CHARACTER*17 F
e.g.    CHARACTER*3 A,B
        CHARACTER*2 C
        CHARACTER*6 D
        CHARACTER*1 E
        DATA A,B,C,D,E/'THE','DOG','IS','ASLEEP',' '/
        F=A//E//B//E//C//E//D
        WRITE(2,1)F
        FORMAT(1X,A17)
```

Resultant output is THE DOG IS ASLEEP

Note if we had written F=D//E//C//E//A//E//B

the output would be ASLEEP IS THE DOG

Free format input/output

Whilst some Fortran 4 compilers have had free format facilities for a long time, these were extensions to the language, not standard Fortran 4.

Free-format is a standard feature of Fortran 77.

Thus, whereas previously a program using free-format statements only worked on some Fortran 4 compilers, it will work with all Fortran 77 compilers, and so programs using free-format are now fully portable between computers.

The standard Fortran 77 READ and WRITE statements are:

```
        INTEGER K,L,M
        REAL A,B
        READ (1,*)K,L,M
        READ (1,*)A,B
        WRITE(2,*)A,B,K,L,M   ++
```

This program will expect two lines of input data. The first should be three integer values separated by at least one space. The second should be two real values separated by at least one space. Output values are separated by commas.

Character strings can also be input and output in free-format.

```
e.g.    CHARACTER*6 F
        READ(1,*)F
```

++The command WRITE(2,*) is known as a list directed output statement. It is not in general suitable for writing data to a file for subsequent input to the same or another program. However, list directed output written to a file can be read back in by a free-format (list directed) READ statement provided the data only consists of numerical and logical values.

Note that when free-format input is used for character strings, the required characters should be contained within quotes when typed in at a terminal.

i.e., to enter the character string PQRSTU
when the executing program gives the prompt > requesting data input,
type 'PQRSTU'

In the above free-format READ and WRITE statements, the numbers 1 and 2 refer to the logical unit number (I/O channel).

There is an alternative form of READ and WRITE statement which uses default logical unit numbers.

These read from the user terminal but write to the lineprinter, using the default logical unit assignments set up on the Perkin-Elmer 3220.

These alternative forms are:

```
READ*,K,L,M             (reads from logical unit 1)
WRITE*,K,L,M            (writes to logical unit 3)
```

This form of free-format could be forced to read and write to the user terminal by including the following assignments in a user-written assignment file:

```
AS 1,CON:
AS 3,CON:
```

3. Program Size Reduction

Reducing program size is likely to have the greatest impact on improving execution speed out of the measures discussed. There are many areas in Fortran programming where careful thought can contribute to program size reduction and these will be considered in the following discussion.

3.1 Overlaying

One measure which can contribute greatly to increasing program execution speed is overlaying, the mechanics of which have been discussed in the last lecture. The benefits of overlaying are very difficult to predict quantitatively. Overlaying reduces the size of the program and so there is an inherent increase in execution speed. However, this is at the expense of the time taken to load each overlay from disc into memory as it is needed, and this detracts from the speed improvement resulting from the size decrease. However, if the subroutines in each overlay are carefully chosen to minimise the number of times each overlay has to be loaded during program execution, overlaying can have very significant benefits. For instance, a user recently reported that, having reduced his program size from 125Kb to 85Kb by overlaying, execution speed increased by a factor of 5. Such an improvement may not always be possible, but, providing some thought is given as to how subroutines are divided into overlays to avoid excessive swapping of overlays in and out of memory, overlaying can usually be guaranteed to give a substantial improvement in execution time.

3.2 Calculation of memory requirements of program variables and arrays

The memory requirements of each variable in the program differs according to the TYPE of the variable as given below.

Variable type		Memory requirement (bytes)
INTEGER*2	(range -32768 to +32767)	2
INTEGER*4 / INTEGER	(range -2147483648 to +2147483647)	4
REAL	(range 16^{-65} to $16^{63}-16^{57}$)(accuracy 7 digits)	4
DOUBLE PRECISION	(range 16^{-65} to $16^{63}-16^{57}$)(accuracy 16 digits)	8
COMPLEX	accuracy 7 digits for each part	8
DOUBLE COMPLEX	accuracy 16 digits for each part	16
CHARACTER*n		n

The usual lesson from this table is to use INTEGER*2 variables rather than INTEGER (INTEGER*4), providing the range is sufficient, as this saves 2 bytes per variable. This saving becomes significant in the case of integer arrays.

Careful thought should be given to the dimensions of arrays. For instance a (30,30,30) real array requires 108000 bytes of storage. Making a small concession on the accuracy of the program might mean that an array of (15,15,15) could be used instead, requiring only 13500 bytes of storage (a factor of eight reduction).

3.3 COMMON statement

COMMON statements allow the memory storage area used by variables and arrays in one program segment to be shared by other variables and arrays in other program segments. This is particularly useful for work arrays which are only used to hold intermediate values whilst a subroutine is being executed and are not required after subroutine execution. All such work arrays in different subroutines can be made to share the same storage area, which can have a significant effect in reducing the program size.

It should be noted, however, that as soon as we put COMMON statements in subroutines, we are digressing somewhat from the structured programming rule about independence of program modules. The COMMON statement allows the data values within a subroutine to be altered by other segments external to the subroutine, and so breaks the rule that a subroutine cannot be affected other than by the parameters defined in the CALL statement.

These common storage areas, of which there are two kinds, are called common blocks. The two types are blank common and labelled common, and the difference between them is that each labelled common block has a name associated with it whereas blank common blocks have no name. Thus, to enable all common blocks to be uniquely identifiable, a program can only have one blank common block but any number of labelled common blocks (as long as each has a different label).

The variables and arrays to be stored in each common block are defined in a COMMON statement, and the quantities are stored in the order in which they are declared.

e.g.	COMMON/P/A,B,C	Labelled common block C contains 3 variables A, B and C
	COMMON/Q/D,E,F(10)	Labelled common block Q contains 2 variables D and E plus a ten-element array F

These two common blocks could alternatively be declared in a single COMMON statement:

```
        COMMON/P/A,B,C/Q/D,E,F(10)
```

Elements to be stored in a blank COMMON block are declared by a statement such as

```
        COMMON B(3,4),C(3,4)
```

This puts 2 twelve-element arrays B and C in a blank common block.

The COMMON statement is sufficient to define arrays and no additional DIMENSION statement is necessary.

If a subroutine A uses a work array X(10,10) and subroutine B uses two work arrays P(5,5) and R(10,5), then the work arrays can be made to use the same labelled common block G as follows:

```
        SUBROUTINE A
        COMMON/G/X(10,10)
        .
        .
        .
        RETURN
        END
        SUBROUTINE B
        COMMON/G/P(5,5),R(10,5)
        .
        .
        .
        RETURN
        END
```

There is no rule which says that wherever a common block is declared in different segments the elements must add up to the same storage requirements in each segment. However, problems will arise if the elements declared to be in a COMMON block exceed the size that the COMMON block was first created as. It is good practice to declare all COMMON blocks used in the main segment, and make sure that the declared size there is as large as the largest size that the COMMON block is declared as in any subroutine.

Thus, in the above example, the maximum size of common block G is 100 elements in subroutine A (in subroutine B 75 elements are used).

Thus, the main segment should initialise COMMON block G as 100 elements

```
e.g.      COMMON/G/Z(100)              [in main segment]
```

3.4 EQUIVALENCE statements

An important use of EQUIVALENCE statements is to split up arrays into smaller sub-arrays.

```
e.g.      REAL A(4,4),B(4),C(4),D(4),E(4)
          EQUIVALENCE(A(1,1),B),(A(1,2),C),(A(1,3),D),(A(1,4),E)
```

This sets the first four elements of array A equivalent to B
and sets the four elements of array A starting at element (1,2) to C
and sets the four elements of array A starting at element (1,3) to D
and sets the four elements of array A starting at element (1,4) to E

As the elements of an array are stored internally column by column, this means that B,C,D and E are set respectively to columns 1,2,3 and 4 of array A. This can be a very useful technique of splitting arrays up into separate columns and avoids the alternative tedium and extra storage requirement of copying A to B,C,D and E via DO loops.

4. Input-Output

Program execution speed can be enhanced considerably if input and output to the program is organised efficiently. The time-sharing feature of the operating system operates to give each program running in the computer equal amounts of CPU time in turn. These time slices are typically 200 milli-seconds long and thus program A would get 200 ms followed by program B, followed by program C, etc. As soon as all other programs have had 200 ms, program A will get another 200 ms slice. In order not to waste CPU time, a program only gets its full 200 ms slice if it continues to have work for the CPU to do during all the 200 ms. If at any time it temporarily stops needing the CPU, its time slice is automatically ended. Such a cessation of CPU requirement occurs whenever a job needs to input from or output to a terminal. Thus, in order to make full use of its share of CPU time slices, the program must minimise the number of input and output operations, which in turn means that the number of items transferred during each input or output operation must be maximised.

What all this is leading to is to suggest that, as far as possible, all input and output operations to terminals should involve as many data values transferred per line as possible.

e.g. if three data values are to be read in or written out,

```
use            READ(1,3)A,B,C
               WRITE(1,3)D,E,F

rather than    READ(1,3)A
               READ(1,3)B
               READ(1,3)C
               WRITE(1,3)D
               WRITE(1,3)E
               WRITE(1,3)F
```

etc.

5. Coding Efficiency

A great amount can be gained in terms of improvements in program execution speed by coding a program in an efficient way. This generally means removing all redundant operations, where an arithmetic operation is repeated more than once when none of the elements being operated on have changed and so the result remains the same. Such redundancy of operations occurs frequently during looping.

Another area where useful computation time can be saved is when numbers are to be squared, because exponentiation takes considerably longer than a single multiplication operation.

Suppose the square of A is to be assigned to variable B.

```
Then      B = A*A          is much more efficient

than      B = A**2
```

A good example of redundancy in looping is afforded by the following part of a program to calculate the Fourier Series, where a_n has to be calculated according to the formula:

$$a_n = \frac{2}{m} \sum_{k=o}^{m-1} f(x_k) \cos\left[\frac{2nk\pi}{m}\right]$$

It is very tempting to translate this straight into Fortran, using the following DO loop:

```
          AN=0.0
          DO 1 K=0,M-1
          ANG=(4.0*ATAN(1.0)*N*K)*2.0/M
          FK=(2.0/M)*K-1.0
          AN=AN+FK*COS(ANG)*2.0/M
     1    CONTINUE
```

Unfortunately, this code is horribly inefficient. Several quantities within the DO loop never change value and yet are evaluated on every iteration of the loop. Thus, there are a large number of unnecessary multiplications and function evaluations carried out. The worst of these is the evaluation of ATAN(1.0).

Close inspection of the DO loop reveals that several elements inside it can be taken outside and evaluated just once, which is very much more efficient. The improved program looks like:

```
      X2M=2.0/M
      ANG=X2M*N*4.0*ATAN(1.0)
      AN=0.0
      DO 1 K=0,M-1
      FK=X2M*K-1.0
      AN=AN+FK*COS(K*ANG)*X2M
1     CONTINUE
```

6. <u>Program tracing</u>

When execution errors are occurring in a program, tracing allows the history of operations prior to an execution error to be examined. Either the whole or just one or more sections of a program can be traced. The change in value of some or all variables and the order of execution of statement labels is information which can be included within a trace report. Disclosure of array subscripts which go out of bounds is also optional trace information.

Tracing substantially increases the execution time of the program, and so it is prudent to limit tracing to the area of the program where an error is expected. The detail of trace information demanded should also be carefully chosen according to what sort of error is occurring and hence what information is required to detect the cause of error. For instance, the smaller the number of variables traced, the smaller the overhead on program execution time.

The basic commands associated with tracing are $TRACE and $NTRACE. Each section to be traced should start with $TRACE (in column 1) and end with $NTRACE. If the whole program is to be traced, the first line should be $TRACE and the last line $NTRACE.

Optional arguments can follow the $TRACE command, specifying variables and array names to be traced, and a line of trace output will be produced whenever one of the specified variables and arrays changes value. If no arguments are included with a ØTRACE command, then all variables and arrays are traced. Irrespective of ØTRACE arguments, all statement labels are traced as well as variables.

e.g.	ØTRACE A,B	initialises tracing of variables A and B only
	ØTRACE	initialises tracing of all variables
	ØNTRACE	terminates variable and statement label tracing

Array subscripts going out-of-bounds can be traced in a section of program by including the command ØTEST at the start of the required section. The command ØNTEST terminates subscript bound checking. Again, particular array names to be checked can follow the ØTEST command as arguments. If no arguments are specified, then all arrays are checked for subscripts going out of bounds.

e.g.	ØTEST C,D,E	initialises subscript bound checking for arrays C, D and E only
	ØTEST	initialises subscript bound checking for all arrays
	ØNTEST	terminates subscript bound checking

5.1 Trace output

All trace information goes to the peripheral assigned to logical unit 6. The only sensible medium for outputting this trace output is a disc-file from which the required information can be subsequently extracted. A VDU is unsuitable for trace output, as all but the last twenty or so lines of information are lost. Likewise, a line printer is unsuitable because the amount of trace information can be enormous should a program go into an unending loop, and a boxful of lineprinter paper can be quickly used up. In fact, trace output directly to a lineprinter is BANNED, in deference to the high cost of lineprinter paper.

A file for trace output is automatically allocated by the run-time default assignment command file (DEFAULT.ASN used by the RUN command) and assigned to logical unit 6 for any trace output which might be created.

A LISTTRAC command is provided (see below) to allow the extraction of the required information from the trace file.

If your program has a user-written assignment file rather than using the default assignment file, the file should include the following commands to allocate and assign the trace file.

```
XDE TEMP:PROG.TRC
AL TEMP:PROG.TRC,IN,80
AS 6,TEMP:PROG.TRC,SWO
```

5.2 LISTTRAC command

The LISTTRAC command allows the required information to be extracted from the trace file (TEMP:PROG.TRC) and displayed on either the user terminal or the lineprinter. Normally, the required information pointing to the execution error is contained within the last few lines of the trace output. However, to cover for all possibilities, LISTTRAC allows up to the last 250 lines of trace output to be displayed.

This facility is invoked by typing as follows, starting in column 1:

```
LISTTRAC
```

The computer responds by asking whether trace output is to be displayed on the user terminal or lineprinter (type 1 for user terminal, 2 for lineprinter). Finally, the computer asks you to specify how many lines of trace output are required (maximum 250).
e.g. entering 50 causes the last 50 lines of trace output to be displayed.
Following display of trace output, the computer asks if you want to re-display trace output. This facility is provided because you may wish to examine more trace records than at first, or perhaps you have initially looked at trace records on a VDU and now require hard copy on the lineprinter for more detailed examination.

At this stage, trace output can be repeated as many times as desired. When you finally reply to the computer that you don't require any more trace output, the final action of the LISTTRAC function is to delete the trace file (TEMP:PROG.TRC), which may be quite large and would seriously interfere with the amount of free disc space if left undeleted.