X-530-77-72

TMX 71307

OPTIMIZATION GUIDE FOR PROGRAMS COMPILED UNDER IBM FORTRAN H (OPT = 2)

(NASA-TM-X-71307) OPTIMIZATION GUIDE FOR PROGRAMS COMPILED UNDER IBM FORTRAN H (OPT=2) (NASA) 174 p HC A08/MF A01 CSCL 09B

N77-26827

Unclas
G3/61 28946

D. M. SMITH
A. H. DOBYNS
H. M. MARSH

APRIL 1977

GSFC
GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

X-530-77-72

OPTIMIZATION GUIDE FOR PROGRAMS COMPILED UNDER IBM FORTRAN H (OPT=2)

D. M. Smith
A. H. Dobyns
H. M. Marsh

Prepared Under Contract NAS5-20747 by
Boole & Babbage, Inc.
Sunnyvale, California

for Computer Management Branch
Mission Operations Computing Division
Mission and Data Operations Directorate
Goddard Space Flight Center
Greenbelt, Maryland
National Aeronautics and Space Administration
Technical Monitor: Evmenios P. Damon

April 1977

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

CONTENTS

CONTENTS (Cont'd)

CONTENTS (Cont'd)

CONTENTS (Cont'd)

SUMMARY OF FIGURES

APPENDICES

ACKNOWLEDGEMENTS

This guide is the accumulation of knowledge of many people and experiences from their time in the electronic data processing field. Contributors of various sections are: Mr. Al Dobyns for Good Programming Practices; Mr. Howard Marsh for Computer Architecture; Mr. Mike Kascic for Floating Point Arithmetic, Functions and Approximations, and Polynomial Evaluation; and Mr. Stu Bell for Overlay. Some of the Code Comparison examples were run by Mr. Jim Fisher. Most of the double checking and assurance that the generated code illustrated the proper intended examples were done by Mr. Walter Martin. The endless effort of proofreading was performed by Messrs. Jack Balakirsky, Al Dobyns, Howard Marsh, and Tom Parker. The bulk of the typing and correction work was done by Mrs. Pat Apel.

1.0 INTRODUCTION

1.1 OPTIMIZATION

This document is designed to provide the programmer with various techniques for optimizing programs when the FORTRAN IV H compiler is used with OPT=2. For optimization tips for programs compiled with FORTRAN IV G, FORTRAN IV G1, and FORTRAN IV H (OPT=0 or OPT=1), see Goddard document X-543-71-99.

The programmer has a number of considerations to make as a program is developed. Is it better to write obvious code or more efficient and less obvious code? When is the deadline? How often will a section of code be modified, or who else must work with the finished program?

Many features of FORTRAN allow the programmer relative ease in writing the programs, but at times, this ease may be costly in execution time.

In general the high usage areas should have the most time spent on them to allow the best execution and coding possible. The programmer may know many of these high usage areas of code as they are being written. When this is not the case, Boole and Babbage's Problem Program Evaluator (PPE) may be used to locate the high usage areas. In general it is a good idea to double check the programmer's guesses with PPE on production programs and long running jobs where any small change will net a considerable savings. The Boole and Babbage representatives may be contacted at 982-2863 or through the Programmer Assistance Center (PAC), 982-6768.

1.2 DOCUMENT USE

This document is divided into a number of sections. All the information may be required to design and write an optimal program. This is a near impossibility and would require an inordinate amount of time. With the exception of the Optimizing Suggestions sections (4 - 13), the information presented here is background and assumes familiarity with the internal operation of IBM large scale scientific computers.

Each of the Optimizing Suggestions sections is presented in three parts. The first, Summary (.1), is a brief synopsis of the results of the test programs and a quick summary of which techniques are, most generally, the best to use. The second part, Code Comparisons (.2), is a description of the programming techniques used and the results from the test programs. Examples are presented to demonstrate the specific techniques used. The last part, How the Statements are Compiled (.3), is a discussion of the results, and briefly what is occurring to make the results as they are. This should give the programmer a feel for applying the demonstrated techniques to his own programs.

This paper does not pretend to be complete but should present most of the commonly seen programming practices. Comments and suggestions concerning this document are welcome and should be directed to the Boole and Babbage staff.

1.3 TEST PROGRAM DESIGN

Whenever possible all of the various techniques were compared for execution speed. Since the optimizer is included in these tests, it was necessary, at times, to defeat the code movement optimization to avoid conflicts between sections of various tests using the same, precalculated results. The different FORTRAN's section presents a discussion of the various techniques the compiler uses to improve the internal machine language code generated.

The coding techniques shown here are pieces of programs and meant for substitution to individual program specifications and designs. The timings presented throughout the document are for the tests as run on the 360/65 with the code as noted in each example. The Computer Architecture section (3) should also be read to understand the variability in timings obtained in the resultant statistics and the effect of moving a program from one CPU to another.

The first test program used for the earlier document was originally designed as a single program. A great amount of difficulty was encountered in the main loop to avoid the compiler's optimizer recognizing and moving or removing similar code from within the large loop to outside the loop. The reliability of the measurements was in doubt, with a few sections of code taking most of the execution time. This left other sections with less execution time than the accuracy of either the internal CPU timer or the confidence levels for the PPE.

Each group of tests was placed in a separate program and run to obtain enough samples to insure statistical accuracy, as given in the Boole and Babbage PPE Guide. The run times varied from about 4.5 minutes to over 20 minutes CPU time or between 14,000 and 40 million executions of different test programs. All timings, the percent, total time, and the number of passes are given in Appendix H. The best code of each set of examples is marked with an ! after the example number.

The MOCF 360/65 was selected for timing tests as the architectural features are the simplest and would cause the least variability in measurement and code interdependencies (see the Computer Architecture section (3) and the description of the optimizer features in section 18). The machine architecture plays an important role in how particular jobs perform on a specific machine, but the interest here

is in describing the best general programming techniques. In "real" programs the effect of data structures and code location in memory may make a technique execute better than one which has been shown here to be more effective.

1.4 STATISTICS USED

All timings in minutes or seconds are the result of taking the percent of run time, as indicated on the PPE Specific Intervals Report, and the total step measurement time, as reported on the step end statistics, to arrive at the figure reported for the particular sections of code. No comparisons were made between different jobs. The accuracy of these timings is not exactly known but should be accurate to the internal CPU timer and may vary by about 5 to 8 percent. Any comparisons closer than that may essentially be considered the same, except that the relative timings were the same as have been noted. The suggestions of which techniques are best will still hold true as these were also examined for best internal language code generated by the compiler which would execute the fastest independent of machine architecture.

The statistics collected are the result of interrupting the program to be measured every 16 milliseconds, as measured by the system clock, and recording the Program Status Word (PSW). The PSW contains the address of the next instruction to be executed. Some interrupts were ignored as a higher priority program may have interrupted the PPE extractor or the problem being measured. This sampling error is taken into consideration in the accuracy discussion in the PPE User's Guide. The extractor for PPE was set to a priority of 195 and the problem program to 160 (out of a maximum priority of 255) to be placed above most of the general work in the system and hopefully increase the reliability of the measurements. Since the PSW is pointing to the next instruction to be executed, all intervals reported need to be backed up by one instruction (two, four, or six bytes) to reflect the time spent executing by the proper instruction. On the 360/65 (with IBM OS MVT Release 21.8), it is usual that several instructions are fetched from memory (double-word fetch). The fetch time is included in the measurements and amounts to between 0.05 percent and 0.09 percent of the run time for each 16 bytes of code. All tests are run long enough for this variability to be removed. The result of boundary alignment of instructions on double-word boundaries as opposed to instructions off double-word boundaries was checked independently. The resulting increase in fetch times was considered small enough to be discounted for the difference between different sections of code where there might be one more fetch than in another section of code. This time is meant to be measured when one technique is longer and is part of the overhead involved with longer code.

2.0 GOOD PROGRAMMING PRACTICES

The use of good programming practices is essential in achieving the goal of a well written, easy to use program within a reasonable schedule. To rush through the first important steps of problem definition, selection of algorithms, data structures, and languages in order to "get to work" on program logic and coding will most likely be heavily paid for in the debugging phase! Experienced programmers are already aware of this; however, being human, not all have disciplined themselves.

The following list is a reasonably acceptable breakdown of the process of writing a program:

Problem Definition
Selection of Algorithms and Data Structures
Selection of Programming Languages
Specification of Program Logic and Structure
Program Coding
Testing and Debugging
Documentation
Maintenance

The content of the following sections is presented in a general sense. Specific references to the facilities available at GSFC will be mentioned where applicable. It should be assumed that there will be some parallel effort on some of the steps. In particular, documentation should be a part of every step (some projects require documentation on the progress of the documentation itself). Effort should be spent documenting on a continuing basis to provide a more accurate picture of the work being done and to avoid a last minute rush to meet a deadline or wasted effort by several people.

2.1 PROBLEM DEFINITION

Problem definition may sound too trivial to mention, but it is essential that the customer understands what it is he wants the program to do (and not do) and that he imparts this knowledge to the programmers assigned to the task. Program needs do change and it is necessary for both sides to check on a regular basis with one another. It would be rather embarrassing should the customer forget to let the programmers know of a new development and then several meetings later discover that an important specification was omitted. If the customer is someone with little or no programming background, extra effort needs to be made on the part of the programmers in the problem definition phase. Also, if a schedule is formulated, care must be taken to avoid overly optimistic target dates. Some customers are probably not aware of computer requirements or the time required to formulate, check out, and debug a computer program. Allowances should be made, if possible, when it is known in advance that special circumstances will occur within the program's development schedule. A change of computing hardware or a switch to a different operating system can cause delays of weeks or possibly months.

2.2 SELECTION OF ALGORITHMS

Selection of algorithms and data structures is the next step after problem definition. A data structure may be defined to be the relationship between data elements, characteristics of the elements, and the order in which the elements are arranged within the records. Records within data sets may be ordered if required by the program's specifications. In some cases the methods to be used will be stated in the specifications. There may be some latitude in that the mathematical formulae may be given, but not the techniques to be used. Several sources of literature on algorithms exist including the leading computer journals (Collected Algorithms from the ACM), textbooks (the Art of Computer Programming series by D. E. Knuth), and indexes to program libraries (GSFC Computer Program Library Catalog, IMSL, etc.). Much effort can be saved if the program already exists, even if it needs modification to satisfy the customer's specifications. Data structures should fit the program's algorithms and should be designed to reduce the complexity of the program. For efficiency's sake, unformatted data records are best for handling quantities of data between programs or between executions of the same program. The use of formatted data should be restricted mainly for use in generated reports. Input data formats should be easy to read and use. The use of NAMELIST in FORTRAN programs allows the user to input data by variable name while not being overly concerned about column usage. However, NAMELIST requires more processing than formatted reads.

2.3 SELECTION OF PROGRAMMING LANGUAGE

The four predominate choices of programming languages available on the M&DO System/360 computers are, in order of use, FORTRAN IV, OS/360 Assembler language, PL/I, and COBOL. The choice of language should depend on the needs of the customer but may be fixed by such factors as the knowledge of the programmers, the need for portability of the programs, and the ease in maintaining the program. FORTRAN is a well known and stable language suitable for the predominantly scientific programs needed at GSFC. A variety of FORTRAN compilers are available to the GSFC computer users. The FORTRAN IV H compiler is available on all of the larger GSFC 360 computers and is commonly used due to its optimization features. The FORTRAN IV G compiler is available on all M&DO 360s as well. The IBM FORTRAN IV H Extended Plus compiler is available on the SACC 360/91. Libraries available include the regular IBM mathematical functions (SIN, ATAN, etc.), the International Mathematical and Statistical Library (IMSL), and a GSFC FORTRAN library containing commonly used subroutines not found in the others. The IBM 360 Assembly language contains all the power needed to handle any situation which FORTRAN cannot. There are definitely areas where either language can be used, such as bit/byte manipulation. The FORTRAN IV H compiler contains several useful bit and byte manipulation statements (or functions) which are described later in this document. The choice of language may depend

on compatibility problems. PL/I is a powerful language in the sense that a wide range of data types, I/O methods, and statement type are available. If PL/I is chosen, it may be difficult to maintain the program as there are very few good PL/I programmers available for assistance should you need any. Also, other installations may not support the language (no PL/I compiler or libraries available). Its use on the 360/91 or /95 will result in severely degraded system performance if a program relies heavily on the use of decimal instructions or uncontrolled (automatic) storage. The simulation software for decimal instructions must run in a special system state during which no other processing can take place. For this reason, care must be taken to avoid declaring and using constants or variables with FIXED DECIMAL attributes. The use of uncontrolled storage results in extra overhead in the use of the GETMAIN/FREEMAIN Supervisor Calls (SVCs). A mix of FORTRAN, COBOL, and ALC subroutines presents little, if any, difficulty. Interfacing FORTRAN or COBOL with PL/I can be done, but it usually requires some form of interface subroutines (as PL/I data structures are formulated quite differently).

2.4 SPECIFICATION OF PROGRAM STRUCTURE AND LOGIC

Program logic and structure determine to a large extent the ease of coding and debugging the final product. Ill-defined logic will leave loopholes which will plague the programmer long after the program is in use. Patches applied to the program will plug some but, most likely, not all of these loopholes. Even if all of the loopholes were found and fixes applied, the patched program will not be as efficient as one based on complete and well-defined logic. Additional time will have to be spent on reorganizing the source so that a more efficient and easier program to read and maintain is produced.

Modularization is a common method of designing a program. The general goals of the program are broken down into a series of major tasks. These tasks are subdivided until a unit level is reached. A unit level can be considered as the smallest reasonable amount of logic to be coded and, quite often, can be readily retained in the programmer's mind. Modularization is usually accompanied by program structuring or flowcharting. Flowcharts are a visual description of a program module's logic. The major stumbling block in the writing of flowcharts is in their oversimplification or in the inclusion of too much detail. For large programs, it may be advisable to have two levels of flowcharts. One level is to give an overview of the major parts of the whole program. The second level is more detailed and may result in separate charts for the more complicated modules. These flowcharts are intended for use by programmers new to the system and those responsible for maintaining the program.

Program logic should also include debugging aids. At the module level, debugging output should reflect the correctness of the input, the computations and/or data manipulation, and the output. The modules should not depend on the presence, absence, or execution of the debugging aids. In addition, consideration should be given to having the debugging output controlled by the main program or in the case of interactive programs through requests made by the operator at a display terminal. A successful merger of debugging output and normal output was achieved in one program by dividing the output into as many as six levels. Each level gave more detail on the computations involved. The deeper levels were used only if "troublesome" data was received which made it difficult for the program to arrive at reasonable solutions.

2.5 PROGRAM CODING

Coding is the processing of the previous steps into a form suitable for input to a computer. Since the translation of a coded program into machine executable form is done by the computer through compilers or assemblers, the symbols chosen need not appear meaningful. But programs are written and read by people and, therefore, must be coded to convey as much meaningful information as possible. The use of variable names such as HOUR, MIN, and SEC are very obvious in there use whereas RH, RM, and RS are not. The fact that MIN will be treated by the FORTRAN compilers as an integer variable should not discourage the programmer from explicitly typing MIN as a real variable. Statement labels in FORTRAN must be numeric; therefore, a "meaningful" label is less obvious and may be chosen based on the programmer's personal preference. Using an ascending sequence of statement labels does have the advantage of making it easier to read a module's logic. It is strongly recommended that a description of the routine's input, output, COMMON area usage, and other useful information be coded as comments at the beginning of each routine. Variable names should be chosen to avoid confusion such as having similar spelling. It is easy to mistake the letter O and the numeral 0 (also the letter I and the numeral 1). Language processors will recognize the difference and use the different storage areas assigned to each one. The varying results from one run to the next can be due to misspelled and, therefore, uninitialized variables. A similar and perhaps more difficult problem to diagnose is the use of arrays as arguments in successive calls to subroutines. A DIMENSION statement is required in each subroutine to pass the correct address of an array. This statement is also required if the only reference to an argument-received array is in a CALL statement and no reference to the array by subscript exists. Also important is the fact that argument types must agree between calling and called subroutines. Quite different results can occur when the same source is compiled with the FORTRAN G and H compilers. FORTRAN G generates code to move the contents of arguments via the MVC (move character) instruction. The FORTRAN H compiler generates load and store instructions based on the type of each argument. The G-compiled code will not abend during argument processing, but unwanted bytes may be moved which could easily cause incorrect values to be generated. The H-compiled code will abend if the address boundary

of the calling arguments does not agree with those of the called routine's arguments. If a specification (0C6) error occurs in an H-compiled subroutine at a relative address past its last executable FORTRAN statement, then a conflict in argument type is the most likely cause.

Flexibility should be coded in the program where anticipated changes can occur so that changes can be made easily. Specification of I/O units can be done using variables which are set by the main program either through default or through input values. This allows an easily made change from FT06F001 to any other unit, for example. As previously stated, input formats should be designed for maximum user convenience. Other aspects of good coding practices are discussed at length elsewhere in this document. The last item of concern here is the use of "clever" coding. Clever coding tends to be very obscure and requires more than an average amount of time to debug. A common example of "clever" coding is the following:

```
      DO  10  I = 1,N
      DO  10  J = 1,N
10    X(I,J) = (I/J) * (J/I)
```

After some investigation, it should be apparent that the truncation which occurs during integer division is the key. I/J = 0 when I is less than J. Also, J/I = 0 when J is less than I. Only when I equals J is the product non-zero; in fact, the product is 1. All this code accomplishes is to initialize a matrix, X, to the Identity matrix (all diagonal elements equal to 1 and all off-diagonal elements equal to zero). Not only is this example "clever", but it is expensive to execute on the 360/91 and /95 since an integer multiply requires 9 machine cycles and a divide 35 cycles. This is quite a contrast to the 2 to 3 cycles required for Load and Store instructions. Two much more understandable forms of the code are shown below:

```
      DO  10  I = 1,N                      DO  10  I = 1,N
      DO  20  J = 1,N                      DO  20  J = I,N
20    X(I,J) = 0.0          -or-           X(I,J) = 0.0
10    X(I,I) = 1.0                   20    X(J,I) = 0.0
                                     10    X(I,I) = 1.0
```

The next example performs a common function in a "clever" manner.

```
A = A + B
B = A - B
A = A - B
```

Suppose A = 5 and B = 3. After the first line, we have A = 8 and B unchanged. The second line gives us B = 5 and A still equal to 8. Finally we have A = 3 and B = 5. All that was accomplished was a swap of the contents of A and B. The only benefit that can be found is that no additional storage area is needed! That may not offset the lack of readability as compared to the more straightforward logic shown below:

```
TEMP = A
A = B
B = TEMP
```

2.6 TESTING AND DEBUGGING

After all of the coding has been written, the next phase entered is program testing and debugging. Testing can be considered to begin with the verification of the flowcharts or logic diagrams with the logic coded in the program's source. Included in this task is proof-reading the entire program. Quite often it is helpful to have others double check your work. It is very common to miss the same error again and again due to the closeness of the programmer to his work. Others can spot this type of error quickly, thereby reducing the time spent in debugging. After the above is completed satisfactorily, a few selected cases should be tested by following the program source. This method is more practical now with the availability of low-cost pocket or desk calculators. It is likely to yield results, either positive or negative, in less time than to submit the job and then wait for the output. Usually the fastest machine-time turnaround is through the use of a remote terminal system such as TSO. After the source is entered and saved, the programmer can then request a compilation. Once desk checking is completed, actual runs should be made using data for which the true answers are known. The most likely error conditions should be checked; and if sufficient time remains, all other paths should be tested.

At this stage program failures that have begun to occur can be attributed to faults in logic present in the original design or in the coding. Errors in design should be few if a diligent effort was spent in the first phase of programming process. Errors in coding can easily exist without the compiler recognizing them as such. The previously mentioned example of misspelled variables is a common problem. To minimize the possibility of their existence, the programmer can use the FORTRAN H compiler's cross-reference and the map to locate variables (or labels) which have no references or those which are being used without having been initialized. Coding errors which occur without being detected by the current compilers are: a different number of arguments being passed than is expected, arguments in incorrect order, and the use of a constant as an argument which is being changed by the called routine. If the last situation occurs, the constant is "updated" with the new value and all statements referring to that constant will be using the new value. Disasterous results are likely to occur and the programmer may be mislead as to the cause of the program failure. FORTRAN DO loops are executed at least once; therefore, if the upper limit of a loop is less than the initial value (a reasonable case in several programs), a test must be made so that the DO loop can be skipped completely. The ability to read a storage dump is a valuable asset worth the time it takes to learn how to read them. Information of interpreting dumps can be found in the IBM Programmer's Guide to Debugging and in a video-taped series in the GSFC Video-Tape Library.

Documentation should be kept of each error and program change. Users usually discover sooner or later that the adage: "If you can't re-create it--you didn't need it," should not be treated lightly! Tape backups should be kept for the executable load modules as well as the source used in creating them. Utility programs such as IEHMOVE or VSCOPY may be used to unload a module or source library from disk to tape (VSCOPY is recommended because of its ease of use, more efficient I/O, and the capability to select members both to or from the data sets). Source statements (also object, data, JCL) may be retained in a PANVALET library. PANVALET provides both compression (of blank fields) and protection features. Information concerning the use of PANVALET can be obtained from the Programmer Assistance Center (PAC). Other locally written source compression packages are available and are described in the GSFC Computer Program Library Catalog. User disks (permanently mounted) are dumped to tape twice weekly on the M&DO 360s. Mountable user disk packs must be maintained by the user.

2.7 DOCUMENTATION

Upon completion of testing and debugging, all of the current documentation, flowcharts, etc., should be brought up to date. Comments in the source should be reviewed and corrected. If the program is written in FORTRAN, the programmer may wish to use the TIDY program which is documented in the M&DO IBM 360 User's Guide, to clean up the source. By this time there should be a large percentage of material available for proper program documentation. All of the potential material should be gathered and edited into one complete manual.

The second part of documentation is at least equal in importance. This is the writing of an operator's manual. The operator's manual should contain a section describing the purpose of the program, its JCL requirements, input data formats, output formats, and error messages. The programmer should remain available to assist in training the users, and when necessary to make minor changes. The operator's manual guide needs to be carefully proofread as the users will tend to rely on it in a most literal sense. Any errors such as missing commas, too many blanks, etc., will not be automatically weeded out as the original programmer is likely to do. A good test is to give a copy of the operator's manual to someone not familiar with the program and ask him to run a few sample problems. The results could be very enlightening and can contribute significantly to the success of the final program and document.

2.8 MAINTENANCE

Most operational (production) programs are not completely bug free. A few bugs may be made apparent in the first few months of use, and some may remain undiscovered for years. If a group of programmers is assigned the task of maintaining programs, it is essential that they be provided correct and complete documentation in addition to the source. This will give them the best possibility

of correcting a problem with a minimum of delay. The first step in maintaining a program is to be sure that a problem does exist and that it is not due to user error. If additional diagnostic output can be obtained, it should be provided. Care should be taken to logically tie the program error to a specific cause in the program source. After this has been done, tests need to be run using a separate copy of the program to avoid conflicts while the production version is being used. Upon implementation of the changes, all documentation should also be changed. These changes should include both "before" and "after" coding or logic diagrams. The source should retain the original error in comment form as well. It may be advisable to retain the changes in a form suitable for use by a source updating program.

3.0 COMPUTER ARCHITECTURE

This section explains the differences between the 95, the 75 C1 and C2, and the 65 computers. The architectural and hardware differences between these machines determine the relative speed. This section explains how to effectively use the machine hardware to increase the speed of a program.

Since the 95 is unique in design and the most generally utilized computer of the three, the major portion of this section is devoted to it. The 95 is an IBM System/360 Model 91 computer with 1024 thousand bytes of thin film, high speed memory in addition to 4096 thousand bytes of CPU storage. The 91 has hardware for each instruction rather than microprogrammed software, as do all other 360s Model 75 and below.

Six of the nonscientific instructions are not included in the hardware. They are the decimal instructions, AP, CP, DP, MP, SP, and ZAP, which are simulated on the 95 and used in some ALC, PL/1 and COBOL programs. Whenever possible programs using decimal instructions should be run on the 75 or on the 65.

For more information on the computers discussed in this section see the IBM System/360 Model (91, 75, 65) Functional Characteristics manuals GA22-6907, GA22-6889, GA22-6884 respectively.

Why should the application programmer be concerned with the architecture of the computer for which a program is being written? There exist many hardware features specific to a given computer that govern how much time a specific program will require for execution. Once known, many of these factors can be used to the advantage of the program.

Although this section is aimed at the FORTRAN programmer, all programmers can benefit from its reading, for the ideas presented are universal.

All CPU times given are approximate and for comparison purposes only.

3.1.0 EFFECT ON MACHINE SPEED

The amount of memory that a machine has does not affect the amount of execution time that a program requires. For the machines that are discussed, the memories have different speeds, ranging from 0.12 microseconds to 8.0 microseconds. The speed of a memory is determined by the way it is designed.

The 95 has two types of memory, M120J thin film and 2395-2 core. The M120J memory is 1024 thousand bytes long and has an access time of 0.12 microseconds. The 2395-2 memory is 4096 thousand bytes long with a cycle time of 0.78 microseconds. The 95 has a total of 5120 thousand bytes of memory.

The 75 C1 and C2 have less memory than the 95 and part of it is slower. The 75's memory consists of 2365-3 and 2361 storage units. The 2365-3 storage unit contains 1024 thousand bytes and has an access time of 0.75 microseconds. The 2361 storage unit has 1024 thousand bytes and an access time of 8.0 microseconds. Each 75 has a total of 2048 thousand bytes. The difference in memory speeds is one of the reasons why programs executed on the 75 require more run time than on the 95.

The 65 has three memory units, a 2365-2, an ARM 2365, and a 2361-1. The 2365-2 has 512 thousand bytes and an access time of 0.75 microseconds. The ARM 2365 is Ampex memory compatible with IBM and is like the 2365-2. The 2361-1 memory unit is 1024 bytes long and has an access time of 8.0 microseconds. The total amount of memory available on the 65 is 2048 thousand bytes.

One other design item that affects the speed of programs in a computer is the interleaving of memory. The principal of interleaving on the 95 is the same as on the other computers. Only the number of leaves is different--16 on the 95, 4 on the 75 and 2 on the 65.

3.1.1 Interleaving on the 95

The 95 has 16 functionally separate memory units, each capable of operating independently. Each unit is called a memory leaf. The beginning address of each leaf is eight bytes greater than the beginning address of the leaf preceding it. The first byte of the first leaf has an address of zero. The storage on the 95 does double-word store and fetch. This means that each time a request for a store or fetch is executed, eight bytes are transferred. Fetch and store operations are done from double-word boundaries only, those addresses divisible by eight (addresses ending in a zero or eight).

Thus the 95 can fetch or store 16 sequential eight byte double-words simultaneously. Figure 1 should assist the reader in understanding the structure of memory on the 95.

Double-Word Number

	0	1	2	N*	Last Address		Byte
0	8 bytes	8 bytes	8 bytes	8 bytes	X	000	0 XXX
1	8 bytes	8 bytes	8 bytes	8 bytes	X	000	1 XXX
2	8 bytes	8 bytes	8 bytes	8 bytes	X	001	0 XXX
3	8 bytes	8 bytes	8 bytes	8 bytes	X	001	1 XXX
4	8 bytes	8 bytes	8 bytes	8 bytes	X	010	0 XXX
5	8 bytes	8 bytes	8 bytes	8 bytes	X	010	1 XXX
6	8 bytes	8 bytes	8 bytes	8 bytes	X	011	0 XXX

Double-Word Number (Cont'd)

	0	1	2	N*	Last Address		Byte
7	8 bytes	8 bytes	8 bytes	8 bytes	X	011	1	XXX
8	8 bytes	8 bytes	8 bytes	8 bytes	X	100	0	XXX
9	8 bytes	8 bytes	8 bytes	8 bytes	X	100	1	XXX
10	8 bytes	8 bytes	8 bytes	8 bytes	X	101	0	XXX
11	8 bytes	8 bytes	8 bytes	8 bytes	X	101	1	XXX
12	8 bytes	8 bytes	8 bytes	8 bytes	X	110	0	XXX
13	8 bytes	8 bytes	8 bytes	8 bytes	X	110	1	XXX
14	8 bytes	8 bytes	8 bytes	8 bytes	X	111	0	XXX
15	8 bytes	8 bytes	8 bytes	8 bytes	X	111	1	XXX

Storage Leaf Number

Byte Desired Within Eight Byte Double Word

* N = 32,000-1 for the 2395-2 memory on the 95
N = 8,000-1 for the M120J memory on the 95

Figure 1 - Memory Organization on the 95

The FORTRAN programmer should take the effort to align all arrays on double-word boundaries and lay out the storage area with care. The benefit from this effort will be fewer fetches from and the stores to memory. This will reduce the amount of execution time required by the program.

Two cases were designed to test the effects of interleaving on execution speed. They showed the difference in speed between the program that fully utilized interleaving and one which did not. Both programs were run on the 95. The programs were written in assembler to insure all execution factors were equal, i.e., boundary alignment of loops and alignment of variables. The FORTRAN equivalents of the programs are given below.

Case 1

```
      REAL*8 ARRAY(16,1600)
      DO 100 K=1,10000
      DO  90 I=1,16
      DO  80 J=1,1600
 80   ARRAY(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

Case 2

```
      REAL*8 ARRAY(17,1600)
      DO 100 K=1,10000
      DO  90 I=1,16
      DO  80 J=1,1600
 80   ARRAY(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

The layout of the arrays for the cases is given below.

Case 1

(1,1)	(1,2)	(1,1600)
(2,1)	(2,2)	(2,1600)
(16,1)	(16,2)	(16,1600)

Figure 2 - Interleaving Improperly Used for Array Storage

Case 2

(1,1)	(17,1)		(2,1600)
(2,1)	(1,2)		(3,1600)
(16,1)	(15,2)	(1,1600)	(17,1600)

Figure 3 - Interleaving Properly Used for Array Storage

From studying the code and Figures 2 and 3, for Cases 1 and 2, it is clear that Case 1 accesses the same memory leaf 1600 times in succession. In Case 2, no sequential accesses to memory use the same core leaf. For these reasons, Case 2 used 0.118 minutes to execute, while Case 1 used 0.366 minutes. A savings of 68% CPU time.

3.1.2 Interleaving on the 75s and 65

Timing studies were made on the 75 C2 and on the 65 to determine the effects of interleaving on the speed of a program. The 75 high speed memory has four leaves. The 65 high speed memory has two leaves. The programs were similar to those run on the 95. The FORTRAN equivalents of the programs are given below. A different program was required for each computer because each has a unique interleaving factor.

Case 3

```
      REAL*8 TEST(4,1600)
      DO 100 K=1,10000
      DO  90 I=1,4
      DO  80 J=1,1600
 80   TEST(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

75 CT

Case 4

```
      REAL*8 TEST(5,1600)
      DO 100 K=1,10000
      DO  90 I=1,4
      DO  80 J=1,1600
 80   TEST(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

Case 5

```
      REAL*8 TEST(2,1600)
      DO 100 K=1,10000
      DO  90 I=1,2
      DO  80 J=1,1600
 80   TEST(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

65

Case 6

```
      REAL*8 TEST(3,1600)
      DO 100 K=1,10000
      DO  90 I=1,2
      DO  80 J=1,1600
 80   TEST(I,J)=1.1D0
 90   CONTINUE
100   CONTINUE
      STOP
      END
```

Cases 3 and 5 access the same leaf 1600 times before using the next leaf. Cases 4 and 6 access a different leaf each time. On the 75 and 65 both cases took approximately the same amount of time to run. The reason for this occurrence is that neither the 75 nor the 65 has the CPU waiting to access the memory. The overhead on the 75 and 65 is large enough that differences in access time are not a measurable factor of the execution time. The overhead time is large because the CPU must calculate each address at the time that each address is used.

The FORTRAN programmer should set up arrays as outlined above. This will enable the program to make better use of the computer's hardware facilities.

Other runs were made to determine if floating point arithmetic hardware is faster than fixed point arithmetic hardware on the 95. Below are given the two examples and the results.

Case 7

```
      REAL*4 A,B,C,D
      A=1.D0
      B=1.D0
      C=1.D0
```

Case 8

```
      IA=1
      IB=1
      IC=1
      ID=1
```

Case 7 (Cont'd)

```
      D=1.DO
      DO 100 I=1,25000000
      A=A+1.DO
      B=B+1.DO
      C=C*D
100   C=C/D
      STOP
      END
```

Case 8 (Cont'd)

```
      DO 100 I=1,25000000
      IA=IA+1
      IB=IB+1
      IC=IC*ID
100   IC=IC/ID
      STOP
      END
```

The programs were written in assembler to insure that both programs would be similar except for the instructions used. Both programs utilized loop mode. Case 7 used 0.709 minutes to run; Case 8 used 1.672 minutes to run. The results show that the floating point hardware is twice as fast as the fixed point hardware. The reason for this difference is that the fixed point element has one execution unit while the floating point element has two execution units.

The floating point element consists of one add unit and one multiply/divide unit. The add unit is capable of performing two add operations concurrently while the multiple/divide unit does one operation. Thus the floating point execution element can handle three operations at one time provided that they are logically independent. Another reason for the floating point arithmetic hardware being faster is that the fixed point arithmetic processor also handles requests for direct store into one of the general registers by the instruction processor. This will delay arithmetic instructions. The FORTRAN programmer should be aware of the factors so they may be controlled; the result being a faster program. Use floating point arithmetic whenever feasible for programs that are to be run on the 95, thus the program will better utilize the machine and its capabilities.

Of the three IBM SYSTEM/360 machines, (the 95, the 75, the 65), discussed in this document, the 95 is the fastest while the 65 is the slowest. Figure 4 gives the CPU times for the execution on all three machines.

	Floating Point Times	Relative Ratios
65	7.962	11:1
75	4.005	5.6:1
95	0.709	1:1

Figure 4 - Comparison of Floating Point Execution Speeds

The 65 has an arithmetic-logic unit which does the following: addressing, instruction fetching, and actual operation. None of these functions can be done concurrently, thus the time to run a program is long.

The 75 is faster than the 65 because it has an instruction unit and an execution unit, which are able to operate independently. The instruction unit does instruction sequencing and address preparation. The execution unit performs the arithmetic functions. This separation of functions into two independent units accounts for some of the 75's increased speed over the 65. The 75 is a scientific computer, and for that reason it has faster hardware using more efficient algorithms than the 65.

The 95 is a scientific computer designed to be a number-cruncher. Rather than using microprogramming to do operations, like the 65 and the 75, the 95 has hardware to do the operation. The hardware is much faster than microprogramming. The 95's speed is also increased because its processing unit is composed of independent components, an instruction processor, a floating point execution unit, and a fixed point and variable-field-length execution unit.

The instruction processor does the fetching and buffering of instructions and fetching of required operands. It also issues instructions to the proper execution units, handles interrupts, does I/O, and controls status switching. The instruction processor sets up and executes branches and loop mode. The floating point execution unit performs all floating point arithmetic functions. The fixed point and variable-field-length execution unit executes all fixed point arithmetic, logical, and variable-field-length arithmetic operations.

Since all these units can operate independent of each other, and the 95 has hardware instead of microprogramming for all instructions, except for decimal instructions, it is the fastest computer of the three.

3.2 BRANCHING 360/95

The 95 is designed to handle two types of conditional branching. The first type branches forward beyond prefetched instructions, or branches backwards where the branch address is greater than eight double-words from the branch. The second type of branch, a short loop, is a branch whose target address is within eight double-words previous, that is within the range of addresses from present address to present address minus 64. The first type of branch is associated with the GO TO FORTRAN statement and a DO FORTRAN statement where the end of the loop is far from the DO statement. The only way to be certain, with a DO loop, that a program is not in a short loop is to look at a listing of a program which has the LIST option specified.

Since the instruction processor does not know in advance if the branch will be taken, the processor attempts to be ready for both cases but assumes that the branch will not be taken. In order to be prepared, should the branch be taken, the instruction processor fetches the branch target double-word and the double-word which follows it. It is able to do this because it has available two

alternate instruction registers. Thus the instruction processor is prepared to go in either direction on the branch. To FORTRAN users this means that the program should generate code such that the most frequent case will fall through a logical IF statement which has a GO TO as the appended statement.

The second type of conditional branch causes a short loop to be executed. The short loop, or loop mode, is issued when the branch target is before the branch instruction and within 64 bytes, eight double-words. When this occurs the complete loop is fetched into the instruction stack, after which the fetching of instructions ceases. Since all addresses are calculated and all instructions decoded, an effective one instruction per machine cycle is achieved when no data fetches or stores are required. Otherwise instruction double-word fetches are made on alternate cycles. Should the instruction processor find that it cannot process the next instruction it will search the instruction stack, "pipe line", for an instruction that it can process. Thus instructions may be executed out of sequence. During loop mode it is assumed that conditional branches will be taken. Special registers hold the branch target address so when the branch occurs the address does not have to be recalculated, thus saving one machine cycle. Loop mode terminates when any of the following occurs:

1) A branch out of the instruction stack is taken.
2) The branch, rather than occurring, fall through such that the loop is ended.

To the FORTRAN user it is nearly impossible to determine from the FORTRAN code whether or not loop mode will be used. It is best to get an object listing of the program and check to see if small loops will be utilizing loop mode. An example of a short loop is given below, both the FORTRAN code and the compiler generated code.

FORTRAN		Compiler Generated Pseudo-Code			
		Addresses	Code		Comments
DO 1 I=1,6		EAF4	LA	6,4(0,0)	
1 JREV(I)=0		EAF8	LA	5,0(0,0)	
		EAFC	LA	3,24(0,0)	
		EB00	LA	2,4(0,0)	
		EB04	L	8,8(0,12)	
	Actual	EB08	ST	5,3656(2,12)	Store 0 in JREV(I)
	Loop	EB0C	BXLE	6,592(2,12)	
		Register 12 has as contents E8B8.			

Figure 5 - DO Loop Generated Code

In this example the BXLE (Branch on indeX Low or Equal) instruction does the branching. It branches to address EB08, (adds the contents of register 12, E888 hexadecimal, and 592 (250 hexadecimal).

The BXLE initiates loop mode; when the contents of register six gets larger than 24, the BXLE will not branch and loop mode will stop. The FORTRAN compiler also generates BC (Branch on Condition) instructions for DO statements, but not every BC is a loop.

Often the FORTRAN code can be moved about so that the compiler will generate code that uses loop mode. While in loop mode, no conflicts arise between instruction fetching and data fetching.

3.3 REGISTER USAGE 360/95
"Wise Use of Statement Numbers"

Registers are much quicker to access than memory. The placement of statement numbers in a FORTRAN program affects the compiler's ability to optimize register usage.

The usage of registers can affect a program in two ways--size and speed. Given two similar assembler instructions, for example:

1) L R1,DATA and
2) LR R1,R9

where both R9 and DATA contain the same thing, the LR instruction uses half the amount of core as the L instruction. While both instructions, on the 95, require one machine cycle to complete, the LR instruction will often complete before the L instruction because the L instruction requires the use of the addressing hardware, whereas the LR instruction does not.

For comparison's sake, since exact timings are not available on a 95, the following is given:

On a Model 65 a LR instruction takes 0.65 microseconds while a L instruction takes 1.20 microseconds. On a Model 75 a LR instruction takes 0.40 microseconds while a L instruction takes 0.70 microseconds. Thus it is advantageous to make as much use of registers as possible.

In the FORTRAN compiler, when one specifies OPT=2, the compiler scans the code searching for statement numbers. It uses statement numbers to delimit blocks of code. Within a block of code, the compiler attempts to make maximum use of registers, i.e., it attempts to keep variables in registers rather than continually loading and storing frequently used variables and intermediate values. By the end of a block, the compiler must store variables that have been used in registers. Consider the following:

```
                .
                .
                .
a)        Q = R+3
b)    10  S = T+9/5
c)        T = Q+S
                .
                .
                .
```

Figure 6 - Effect of Statement Numbers

Q is used shortly after its assignment and could be kept in a register. It is also apparent that S, in b, could be kept in a register, providing that enough registers are available, until c. However, FORTRAN will stop its scanning at b, statement number 10, because it will be unable to save both Q and S in registers, since entry to the block of code is not necessarily from a. By statement b the program will have stored Q and S, and at statement c it will load Q and S.

Thus statement numbers can be costly and should only be used when required.

3.4 EXECUTION OF INSTRUCTIONS

Figure 7 below shows what the computer does to execute instructions.

```
(addresses)  EXAMPLE  CSECT                    (machine instruction code)
                        .
                        .
                        .
                      USING  EXAMPLE,12
      0               L      5, CON12          5850C 128
      4               A      4, CON13          5A40C 12C
                        .
                        .
                        .
    128       CON12   DC     F'12'
    12C       CON13   DC     F'13'
```

Figure 7 - Example Instruction Execution

As the program executes, the instruction processor fetches instructions from storage, two double-words at a time, and places them in the instruction stack. The instruction processor normally has in the instruction stack the current instruction double-word and the next three double-words. When fetched from storage, the two instructions above will be stored in a single double-word in the instruction stack as follows:

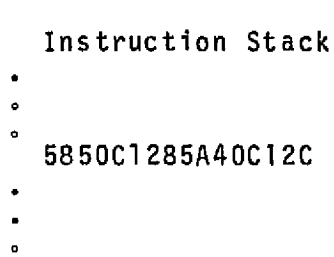


Figure 8 - Double Word Fetch Instruction Stack

The instruction processor will then begin to decode the load instruction. As part of the decoding process, the absolute address of the data in core (CON12) is calculated, and a request for a fetch sent to the Main Storage Control Element (MSCE) to obtain the data. While waiting for the data from MSCE and until the Fixed-Point Execution Element is free, the instruction processor will begin to decode the add instruction. The instruction will send a request for a fetch from core for CON13 to MSCE. When the MSCE receives a request for a fetch (CON12), it searches queue lists, which contain addresses of requested fetches and stores and requests just processed. If a match is not found, the MSCE will add the request to the queue of fetch request addresses. The MSCE processes the queued requests sequentially. For each request a double-word is fetched. When the MSCE receives the request for CON13, it finds that either the request for CON12 is queued or has just been processed, and the data from the fetch is in a buffer. Since both CON12 and CON13 are contained in the same double-word, the MSCE will not do another fetch for CON13. When the data has been fetched and the Fixed-Point Execution Element (FPEE) is free, the instruction processor will send a load instruction to the FPEE. The FPEE will transfer the data from a buffer in the MSCE to register 5. Upon completion the instruction processor will send an add instruction to the FPEE. The FPEE will get the data (CON13) from a buffer in the MSCE, fetch the contents of register 4, add the two together, and transfer the result from the FPEE to register 4. Upon completion of the add instruction, the instruction processor will fetch another double-word and continue decoding the instruction stack.

4.0 SUBSCRIPTING

4.1 SUMMARY

Subscripts for variables should be kept as simple as possible. Involved expressions cannot be incremented by a given amount which the compiler can ascertain. The optimizer is able to recognize variables and expressions in subscripts and calculate them separately from the same variables or expressions not used in subscripts. Expressions should be fully written out as outlined in the Common Expression Elimination section (8). Subscripts should contain no subtraction as this does not compile easily to machine language code.

4.2 ARRAY STORAGE

Arrays are useful data structures and necessary mathematical entities for solving problems on computers. Subscripts are used to refer to the individual elements of the array. To locate the element in the array, its exact location in memory must be calculated.

For example: DIMENSION V(100,50)

where V is a four byte floating point array of variables. The array is stored in memory with the first index varying most rapidly and the last most slowly as shown below, assuming that the first element is located at location 1000:

V(1,1)	V(2,1)	V(3,1)	...	V(98,1)	V(99,1)	V(100,1)
1000	1004	1008		1388	1392	1396
V(1,2)	V(2,2)	V(3,2)	...	V(98,2)	V(99,2)	V(100,2)
1400	1404	1408		1788	1792	1796
.	.	.	...	.	.	.
.	.	.	...	.	.	.
.	.	.	...	.	.	.
V(1,49)	V(2,49)	V(3,49)	...	V(98,49)	V(99,49)	V(100,49)
20200	20204	20208		20588	20592	20596
V(1,50)	V(2,50)	V(3,50)	...	V(98,50)	V(99,50)	V(100,50)
20600	20604	20608		20988	20992	20996

Figure 9 - Two Dimensional Array Storage

An array with three indices as: DIMENSION X(3,4,2)
would be stored with the subscripts as:

1,1,1	2,1,1	3,1,1	1,2,1	2,2,1	3,2,1	1,3,1	2,3,1	3,3,1
1,4,1	2,4,1	3,4,1	1,1,2	2,1,2	3,1,2	1,2,2	2,2,2	3,2,2
1,3,2	2,3,2	3,3,2	1,4,2	2,4,2	3,4,2			

Figure 10 - Three Dimensional Array Storage

4.3.0 CODE COMPARISONS

4.3.1 Arithmetic in Subscripts

```
1)!  V(2*(I+1),2*(I+1))

2)   V(2*I+2,2*I+2)
```

Example 1 requires approximately three times as long to execute as Example 2. Ninety-six seconds as opposed to 27.6 seconds of the 70,000 passes through the loop.

4.3.2 Temporary Variables

The use of temporary variables to hold subscript expressions requires more execution time as shown in the following examples.

```
3)   J= I+2
     U=V(J,J)

4)!  U=V(I+2,I+2)

5)   J1=(I*I)/I
     J2=(I*(I+1))/(I+1)
     J3=J1+J2
     U=V(J3,J2)

6)   U=V((((I*I)/I)+(I*(I+1))/(I+1),(I*(I+1))/(I+1))
```

Example 3 takes longer than Example 4, 89.4 seconds against 28.8 seconds. Example 5 took 367.2 seconds execution time whereas Example 6 took 328.2 seconds for 70,000 executions.

4.3.3 Multi-Subscript Arrays and Vectors

The effect of trying to avoid some subscripting by calculating the expression in single subscript form will consume more time. As the expression becomes more complex and the optimizer can no longer 'see' the simple relationship, the time may even be doubled. However, if the expression is already complex, it may be advantageous to rewrite the subscript with one index and equivalence the single and double subscript arrays together as in the following examples:

```
7)   VV(2*I+2+(2*I+1)*100)

8)!  V(2*I+2,2*I+2)
```

Where Example 7 has VV EQUIVALENCED to V and the single dimension is the product of the doubly dimensioned array limits. Example 7 required 5.75 percent, or 79.2 seconds, of the run time whereas Example 8 took only 2.00 percent, or 27.6 seconds of the 70,000 passes.

4.3.4 Addition and Subtraction in Subscripts

The machine language instructions are organized to allow a forward displacement from a given address very easily, but not a displacement backwards from an address. Addition in subscript expressions may be done quite well whereas subtraction is slow.

```
9)       DO 9 I=7,100
       9 U=V(I-3,I-4)

10)!     DO 10 I=1,94
      10 U=V(I+3,I+2)
```

Example 9 took 1.18 CPU minutes to execute, but Example 10 used only 0.93 minutes for 70,000 executions.

4.4.0 HOW THE STATEMENTS ARE COMPILED

4.4.1 Element Location

To obtain the location of an element in the two index case, the dimension of the first index is multiplied by the second subscript minus one, then add the first subscript value. This quantity is then multiplied by the byte length of the data and added to the start of the array, minus the byte length of a data element. The start of the array must be backed up by the byte length of a data element which allows the first element to be added to the stored address and have the resulting address computation indicate the proper location.

All other subscript values must have one subtracted from the value to obtain the correct location. For example: V(1,1) as outlined earlier would be located as follows:

```
((2nd subscript-1)*dimension 1st index+1st subscript)*length+origin-length
((       1      -1)*          100      +       1      )*   4   + 1000  -4=
                  (0+1)*4+1000-4 = 1000
```

or V(3,49)

```
((      49      -1)*          100      +       3      )*   4   + 1000   -4=
                  (4800+3)*4+1000-4 = 20208
```

The general location may be stated as follows:

$$L=O-1+1(s_1+s_2-1*D_1+s_3-1*D_2*D_1+s_4-1*D_3*D_2*D_1+\ldots+s_{n-1}*D_{n-1}*D_{n-2}*\ldots*D_1)$$

where L = memory location
O = origin of array
1 = length of data element
s = subscript value
D = dimension of the index

For three dimensional arrays the location is obtained as follows:

[(3rd subscript - 1)* product of the dimensions of the 1st and 2nd indices + (the 2nd subscript - 1)* dimension 1st index + 1st subscript]* length of an element + the origin of the array - an element length.

The four index case adds to the previous statement:

length of data element *(4th subscript * product of 1st 3 indices dimensions).

It is obvious, therefore, that the more subscripts used in an array the longer it will take to locate the particular element required.

4.4.2 DIMENSION Statement

The dimension of the last index is not used in calculating the location in the array but is necessary in reserving the proper amount of memory for the array. With IBM FORTRAN, vectors (arrays of one dimension) and arrays used in subroutines use the space allocated in the highest level program unit which defines the array (not true of simple variables). For this reason vectors used in called subroutines need only have their dimension set to one to make the variable an array. The last DIMENSION of a multidimensional array only need be one. The general equation, for a vector, reduces to the origin, minus byte length of an element, plus the subscript, times the byte length. Good programming practice is to document the size of the vector or array in the DIMENSION. When debugging remember that the size dimensioned does not necessarily define the real limit.

4.4.3 Compiler Created Indices

The compiler recognizes the origin of the array and subtracts the length of a data element and stores that constant for reference to the array element. If only a part of an array is referenced as in A(I,50) where only the most rapidly varying subscript changes, the constant stored will be for the beginning of the referenced section. If the subscript expression has constants added to it (less than 4096), the constants are translated as part of a single machine language instruction and only the variable is incremented. The value of that increment is known in the loop. The increment is simply added from its location to obtain the address of the next element referenced. When the expression is not so simple, the increment not known, or the entire loop structure involved, the expression is recalculated each time the subscript is needed (unless the common expression eliminator has found a sub-expression). This is the difference between Examples 1 and 2. In loops such as:

```
      DIMENSION V(100,100)
          .
          .
          .
      DO 1 I=1,4
      L=L+I
    1 U=V(I,I)
```

Figure 11 - Compiler Generated Indices

The variable I is used in three separate forms: 1) the simple variable I starting at one and incrementing by one (1, 2, 3, 4) (added to L), 2) the first index starting at four and incrementing by four (4, 8, 12, and 16), and 3) the second index starting at 400 (400, 800, 1200, and 1600).* When possible the compiler will hold separate forms for all uses of the loop index and increments. When this is not possible, the index is stored in the form as coded (starting and incremented exactly as coded) and the location formula is applied to obtain the element location.

*See section 4.4.1 for explanation of subscript values.

5.0 EXPONENTIATION

5.1 SUMMARY

In writing variables with exponents, it is best to use an integer constant or an integer variable and worst to use a real constant or variable. Exponents of integer constants require only the standard instructions to be generated in order to multiply the base to calculate the result. Any other exponent requires a function to be called involving extra memory and time to pass arguments to that function.

5.2.0 CODE COMPARISONS

5.2.1 Integer Constants and Variables as Exponents

```
11)   X*K**J

12)!  X*K**2
```

Example 11 calls a library function to raise an integer base to an integer exponent (IHCFIXPI) and costs 40.67 CPU seconds for 300,000 executions. Example 12 will simply multiply K by K and uses 7.53 CPU seconds for the 300,000 executions. Any integer constant will cause repeated multiplication. A power of 1000 used 14 multiplies.

5.2.2 Higher Power Exponentiation

```
13)   X**5
```

Example 13 will generate 3 consecutive multiplies, one of them multiplying the previous result.

5.2.3 Single Exponent Versus Repeated Multiplication

```
14)  X**2 * X**3

15)  X * X * X * X * X
```

Example 14 causes separate calculations of squaring and cubing, using one more multiply than Example 13. Example 15 doesn't recognize that the previous products may be multiplied to obtain the final result. Equivalent results were obtained by multiplying X any number of times.

5.2.4 Integer Versus Real Exponents

Using integer variables is a better procedure than using real constants or variables. The function used to calculate the results (IHCFIXPI or IHCFRXPI, depending on the base, integer, or real) is better than the one used for real exponents (IHCFRXPR). Real values for exponents also require the ALOG and EXP library functions. Constants written as real numbers will be treated as real exponents even if their value is integral.

```
16)    X * I**J

17)!   X * X**I

18)    X * I**X

19)    X * X**X

20)    X * X**2.0

21)    X * I**2.0
```

In the following cases each example was executed 300,000 times. Example 16 will call the integer to integer power subroutine and uses 40.67 seconds. Example 17 will need the real to integer exponent subroutine and uses 10.07 seconds (floating point hardware being faster). Example 18 will convert I to a real number and use the same real base to real exponent which Examples 19 and 20 will also use. The timings were 80.74, 76.17, and 76.78 seconds. Case 21 is treated similarly to Example 18 and took 80.19 seconds.

5.2.5 Square Root Function Versus Exponentiation

```
22)!   SQRT(X)

23)    X**0.5
```

To find the square root of a number the base may either be raised to the one half power (specified as a floating point number, not 1/2) or by calling the square root library function. In the 300,000 executions of each of the examples, Example 22 took 25.06 seconds, and Example 23 required 3 times as much as the specific library function, or 73.70 seconds.

5.3 HOW THE STATEMENTS ARE COMPILED

The compiler will try to use the shortest code possible. Multiplying the number by itself or a previous product is possible only for integer constant exponents. Any other cases are handled by the library functions. The requirements for each call are an initialization of a location which points to an argument list (the base and exponent addresses), loading the function address and passing control to that function. Upon return the result of the function is always stored, even if it is to be used immediately. Extra memory is used for the calling instructions, parameter list, and the flag for the ISN (if the compiler ID option is specified, default is on). The relative speeds of the functions for 300,000 executions in seconds are:

```
24)    IHCFIXPI  (I**I)  1.67
25)!   IHCFRXPI  (X**I)  1.43  (floating point hardware faster)
26)    IHCFRXPR  (X**X)  6.92
```

Figure 12 - Speed of Exponentiation Library Subprograms

The integer power functions loop on the power multiplying the base, or a previous product, by itself or the other results the proper number of times. To raise real numbers to a fractional power requires the logarithmic function which subsequently calls the ALOG and EXP functions, whose timings have been included. Real constants are not inspected to verify if the value is integral or not. Since the EXP and ALOG functions are not exact and the total number of instructions executed is larger, the results will not be as precise.

6.0 MIXED MODE ARITHMETIC

6.1 SUMMARY

Numbers are stored and used in two forms within the computer. Whole numbers are called integers or fixed point variables (starting with I - N, explicitly declared, and constants without a decimal point). Numbers with fractions or exponents are called real or floating point (starting with A - H or O - Z, explicitly declared, or constants with exponents and/or decimal points). The representation of the two types is different, and code is automatically generated to convert the values from one form to another when the types are mixed in expressions. If done to excess, or in a loop, the conversion may be very expensive.

Within a DO loop, it may be advantageous to increment a separate counter to use in real expressions rather than convert the index on each pass through the loop. The optimizer will, in most cases, hold the converted index in a temporary variable and convert it only once.

Expressions involving constants of different mode than the variables associated with the operator are often generated as the proper type by the compiler; the major exception being exponents.

On the larger scientific machines, the floating point hardware is significantly faster for multiplication and division than is the fixed point hardware.

Conversion from real single precision (four bytes) to double precision (eight bytes) and expressions involving both have only one added machine language instruction. Conversion from double to single merely uses different instructions and ignores the lower half of the double precision variable (i.e., no rounding is performed).

Complex arithmetic uses two real variables or constants and calls library functions to do multiplication and division operations. Conversion from either real precision to complex of length eight or 16 bytes uses zero for the imaginary part and treats the rest of the conversion the same as it does for single to double precision if required. This adds only four instructions. The complex to real conversion drops the imaginary part of the complex number.

6.2.0 CODE COMPARISONS

6.2.1 Fixed Point to Floating Point Conversion

The conversion of integer to real, single, or double is a lengthy process for which a conversion constant (one per program unit) and 50 bytes of instructions are required for each conversion. Real to integer conversion takes 44 bytes.

```
27)   A = I

28)   J = X
```

29) A = V

30)! J = I

In the test program each of the above statements was executed 1.44 million times. Example 27 took 19.2 seconds, and Example 28 took 16.8 seconds. The two non-converts were very short, each eight bytes, with 4.2 seconds for Example 29 and 3.6 seconds for Example 30.

6.2.2 Mixed Mode Expressions

31) A=J+AJ+K+AK+L+AL+J*K+AJ*AK+J*L+AJ*AL+J*AL*K

32)! A=(J+K+L+J*K+J*L)+(AJ+AK+AL+AJ*AK+AJ*AL+J*K*AL)

In the same loop Examples 31 and 32 took 15.46 percent and 9.20 percent, respectively, to compute the two expressions for the 1.44 million executions. Example 31 required five conversions whereas Example 32 took only two.

Wherever possible group like mode terms together. The result of the expression determines the kind of conversions necessary as each operation from left to right is evaluated according to the FORTRAN language rules.

6.3 HOW THE STATEMENTS ARE COMPILED

Expressions are evaluated by doing the higher order operations first. If any conversion is necessary to complete the evaluation, it is done immediately. As each pair of operands is evaluated, the conversion is in favor of the longer and more complex form until the last level of operations which take place, and the final conversion, if necessary, is to the form of the result to be stored.

The optimizer, when possible, will recognize that a variable or expression is needed elsewhere in the evaluation of a larger expression and will try to eliminate excess conversions. (For further explanation, see Common Expression Elimination, section 8.)

Example 31 is treated as:

a) convert J and save

b) add AJ to FLOAT(J) to form start of running sum

c) convert K and save

d) add AK to sum

e) add FLOAT(K) to sum

f) convert L

g) add FLOAT(L) to sum

h) add AL to sum

i) multipy J and K

j) convert product from i

k) add FLOAT(J*K) to sum

l) multiply AJ and AK

m) add l to sum

n) multiply J and L

o) convert product from n

p) add FLOAT(J*L) to sum

q) multiply AJ and AL

r) add result from q to sum

s) multiply AL and FLOAT(J) (from a)

t) multiply s by FLOAT(K) (from c)

u) add t to sum

v) store sum

Example 32 is evaluated as:

a) add K to J, start sum'

b) add L to sum'

c) multiply J and K, save

d) add c to sum'

e) multiply J and L

f) add e to sum' and save

g) add AK to AJ, start sum

h) add AL to sum

MIXED MODE ARITHMETIC

i) multiply AJ ard AK

j) add i to sum

k) multiply AJ and AL

l) add k to sum

m) convert J*K from c

n) multiply FLOAT(J*K) by AL

o) add to sum

p) convert sum' (from f)

q) add FLOAT(sum') to sum

r) store sum

7.0 DO LOOPS

7.1 SUMMARY

FORTRAN programs may alter their sequential flow in a number of ways. One is by repeating a section of code a given number of times. The DO statement provides this capability. It also provides the compiler with a great deal of information. The information used includes the starting value, the ending value, and the increment added to the index each time through the loop. Sometimes, as with subscripts, more than one index may be created (see Subscripting, section 4), and each index has its own increment. This avoids repeated operations and simplifies the use of the index in the machine language instructions. All of these values must be initialized and incremented before and during the execution of the loop. For that reason short loops, ones which include little code or ones which run over a short range, are to be avoided. The setting up of the loop can cost more execution time than simply writing out the code that is contained in the loop. Additionally, if the loop is kept simple (relatively few variables), it is possible to use faster and shorter machine language instructions than more complex loops.

7.2.0 CODE COMPARISONS

7.2.1 Loop Elimination

Loops used for initialization of variables with less than 32 elements will execute faster if written out, at the expense of eight bytes per variable set.

```
33)     DO  1  J=1,6
      1 KK(3*J)=J+3
```

```
34)!    KK(3)=4
        KK(6)=5
        KK(9)=6
        KK(12)=7
        KK(15)=8
        KK(18)=9
```

Example 33 took 75.0 seconds and 30 bytes whereas Example 34 took 25.8 seconds to execute and 48 bytes of memory for the two million executions. Short running loops are best used when the code within the loop is complex.

7.2.2 Initialization

The best way to initialize variables in an array is to use a DATA statement. While this requires more compile time and increases the size of the object and load modules (by the number and data length

of the array), it requires no execution time and no more memory than not initializing the array or using any other of the methods. (Caution: If the LIST option of the compiler is turned on, the listing of the generated machine instructions will print a line for every element initialized.) COMMON areas initialized with DATA statements and their declaration statements are in a separate section of code called BLOCK DATA. This creates a separate object module for each COMMON area initialized.

The following examples show several ways of re-initializing an array or, excepting Example 35, to transfer the contents from one array to another. These re-initialization techniques in Examples 36 through 39 are faster when the array has not been set to a constant.

```
35)     DIMENSION A(1000)
            .
            .
            .
            .
        DO 3 I=1,1000
      3 A(I)=-1.0

36)     DIMENSION A(1000),WORK(1000)
            .
            .
            .
            .
        DO 4 I=1,1000
      4 WORK(I)=A(I)

37)     DIMENSION A4(1000),WORK4(1000)
        REAL*8 A8(500),WORK8(500)
        DATA A4/1000*-1.0/
        EQUIVALENCE (WORK4(1),WORK8(1)),(A4(1),A8(1))
            .
            .
            .
            .
        DO 5 I=1,500
      5 WORK8(I)=A8(I)

38)     INTEGER*2 L(50),J(50)
        REAL*8 XL(12),XJ(12)
        DATA J/50*392/
        EQUIVALENCE (XJ(1),J(1)),(XL(1),L(1))
            .
            .
            .
            .
        DO 6 I=1,12
      6 XL(I)=XJ(I)
        L(49)=XJ(49)
        L(50)=J(50)
```

```
39)!  DIMENSION A(1000),WORK(1000)
      DATA A/1000*-1.0/
           .
           .
           .
      CALL FMOVE(A,4000,WORK)
```

Example 35 which is straightforward, uses 138.33 seconds, or 19.90 percent of the execution time for 50,000 passes through the 11.586 minutes test step. Example 36 is essentially equivalent to Example 35, but an extra load from memory is required to move the data from one location to the other. This used another 10 percent for 29.98 percent, or 208.40 seconds of the same test programs time. Example 37 sets up a more complicated data structure but only requires half the passes through the loop that Examples 35 and 36 used taking about half the time, 103.23 seconds (14.85 percent). The DO loop moves eight bytes at a time rather than the four in the other two examples. No conversions are done since the type of variables on both sides of the equal sign are the same. The data in memory could be illustrated as shown:

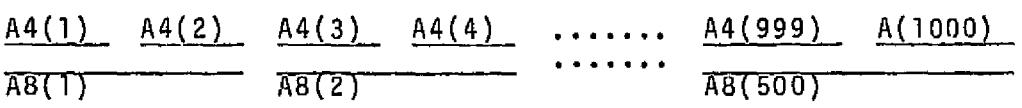


Figure 13 - Four Byte Data Equivalenced With Eight Byte Data

This technique works as well with logical and integer values as it does for these real variables. The savings are more pronounced since the floating point hardware is faster than the fixed point. The amount of data moved must be a multiple of eight. The excess above an even multiple may be transferred by specific assignments as illustrated below and in Example 38:

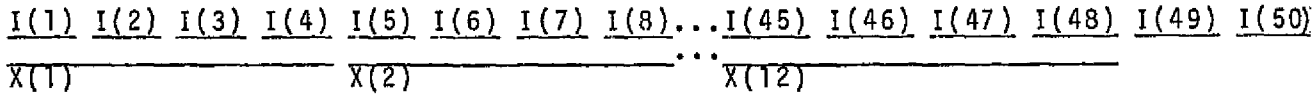


Figure 14 - One Byte Data Equivalenced with Eight Byte Data

Example 39 shows a call to the subroutine FMOVE, which is an assembly module utilizing a machine language data moving instruction. The documentation is in Appendix A. This is the fastest and most obvious move and takes only 51.65 seconds, including the program calling sequence and the time spent in the subroutine (7.43 percent).

7.3 HOW THE STATEMENTS ARE COMPILED

The DO loop testing is performed after the last statement in the loop. The index is incremented and then compared with the final value. If the index is smaller than or equal to the final value, the loop is re-executed with the updated index. When the incremented

index is larger than the final value, processing proceeds with the next sequential instruction. For this reason loops whose loop end value is zero or negative when a variable is used will execute once. The value of the index is left at the loop end value plus the increment. This is important if the index is to be used after the execution of the loop. If the loop is very simple in structure, it is possible that the index will never be stored in memory. The final index will then be the initial value. For this same reason it is possible that the index will not be known in code which does not fall in the logical limits of the loop (between the DO statement and the statement containing the statement number named on the DO statement). An easy solution to the problem is to set another variable equal to the index at the beginning of the loop and use this variable for code outside the loop or after the loop's completion.

8.0 COMMON EXPRESSION ELIMINATION

8.1 SUMMARY

The optimizer, as part of its operation, tries to avoid as much calculation as it can foresee. The programmer can do some things to help the compiler recognize expressions that only need be calculated once for a group of statements. Each expression should be written exactly the same each time excepting spacing, blanks, and breaks for continuation cards. If the expression does not immediately follow the equal sign, it should be placed in parentheses. In general, the use of temporary variables to hold sub-expressions should be avoided as the compiler does a better job of maintaining values internally. Temporary variables should be used when the compiler is unable to pick up common expressions; when the limits of the optimizer are exceeded.

8.2 CODE COMPARISONS

```
40)   H=H+A+B
      G=G+A+B+C
      F=F+A+B+C+D

41)   H=A+B+H
      G=A+B+C+G
      F=A+B+C+D+F

42)!  H=H+(A+B)
      G=G+((A+B)+C)
      F=F+((A+B+C)+D)
```

Example 40 will not recognize any common expressions in any of the three statements. FORTRAN interprets the statements left to right and cannot 'see' that A+B is common in the first two statements or A+B+C is common to the second and third statements. Example 41 will recognize A+B in the first pair of statements and A+B+C in the last pair. Example 42 shows the use of parenthesis to explicitly state common expressions and is interpreted in the same way as Example 41.

8.3 HOW THE STATEMENTS ARE COMPILED

```
40)  a)  H+A          41)  a)  A+B          42)  a)  A+B
     b)  a+B               b)  save a            b)  save a
     c)  store H           c)  a+H               c)  a+H
     d)  G+A               d)  store H           d)  store H
     e)  d+B               e)  a+C               e)  a+C
     f)  e+C               f)  save e            f)  save e
     g)  store G           g)  e+G               g)  e+G
     h)  F+A               h)  store G           h)  store G
     i)  h+B               i)  e+D               i)  e+D
     j)  i+C               j)  i+F               j)  i+F
     k)  j+D               k)  store F           k)  store F
     l)  store F
```

Figure 15 - Expression Translation

COMMON EXPRESSION ELIMINATION

Expressions are remembered ir high speed storage (registers) for up to about two or three statements and for as many as one real or about three integer unique expressions.

In DO loops, often used subscripted variables are moved to internally generated temporary variables. This may help to avoid calculating subscripts many times. If the subscripted variable is re-used many times, the variable may access the temporary variable. The last time the variable is referenced in a loop will always cause it to be saved in storage.

Unsubscripted variables are always used from their real locations unless they are held in high speed storage (registers). Then they are stored at the end of the loop; or when the variable is next used, it is located in the register until it is stored.

The optimizer can only recognize expressions which have the symbols and operators in exactly the same order each time. The spacing and syntax are not important as the symbol names are reduced to unique internal symbols, which are not dependent on the programmer designated names. A+B is not B+A, however, -(A+B) would be recognized as the complement of the common expression A+B. The order of expressions decoding is as given in the FORTRAN language references manual (GC28-6515) and as summarized below:

1. expressions in parentheses
2. functions
3. exponentiation
4. multiplication and division
5. addition and subtraction
6. relational operators (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)
7. .NOT.
8. .AND.
9. .OR.

Figure 16 - Operation Evaluation Order

Expressions written in parentheses are, therefore, recognized first; and when the terms are written in a consistent order in each occurrence, the expression will be saved by the compiler from the first use. Common expressions may be built up, as Example 42 shows. When parentheses are not used, expressions are evaluated left to right in order by the type of operator. Example 41 uses this to build its common expressions and is why Example 40 has no common expressions. Internal limitations set the limit at about 300 expressions that will be recognized, and some are seen for about 39 statements, 672 bytes, and others are not seen in the following statement. Common expressions should be placed early in the statement. If the expressions occur closely enough together or if the instructions generated are simple enough, the common expressions (up to two or three) will be held in registers, allowing for the fastest recall. Single variables will also be saved in registers when used frequently enough

in a small section of code (up to three values each used six times in 12 statements). Integers seem to be saved longer than real variables, perhaps because there are more fixed point registers. When a subscripted variable or expression is not kept in a register, it will be placed in a compiler generated temporary. For subscripted variables this is done if the use is frequent or if the variable is set often in a loop.

9.0 STATEMENT FUNCTIONS

9.1 SUMMARY

Statement functions can make the job of the programmer easier and eliminate some possibilities of coding errors on expressions which are frequently used. The increased cost in run time and the questionable amount at compile time are more than offset by the ease of use. (See also the section Common Expression Elimination, section 8, for additional thoughts.)

9.2 CODE COMPARISONS

```
43)   IFUN(J,K,L,M,N)=J*K + J*L + M*N + K*K
        .
        .
        .
      II1=IFUN(IJ,IK,IL,IM,IN)
      II2=IFUN(IL,IM,IN,IJ,IK)

44)!  II3=IJ*IK + IJ*IL + IM*IN + IK*IK
      II4=IL*IM + IL*IN + IJ*IK + IM*IM
```

Example 43 took 38.00 percent of the 5.637 CPU minute run time, or 128.52 seconds, of the 2.5 million executions. Example 44 required eight bytes less memory, due to the optimizer recognition of terms used later in the expression, and 36.74 percent or 124.26 seconds of execution time for the same 2.5 million passes. Some saving is seen but only 4.26 seconds.

To test the effect on compile time, 510 statement function references were compiled as were the equivalent 510 statements. For ease in creating the programs, the same five statements, or references, were repeated 102 times. The compile time for the functions was .552 minutes on the 360/95 and .548 minutes CPU time for the equivalent statements, also on the 360/95. The difference is within the accuracy of the timer and therefore considered the same.

9.3 HOW THE STATEMENTS ARE COMPILED

Statement functions are defined as a name on the left side of an equal sign with a list of variables in parentheses. The name is not dimensioned, and the definition occurs before any executable statements. This acts as a pattern to generate the real statements in the references. The variables used in the definition are dummy and used only to connect the position in the variable list of the definition with the position in the expression. The correct variables are generated when the reference is used in the program according to the pattern. The dummy variables will not be used or even generated. When a reference is found to a statement function name, the first variable in the list is substituted in the function expression wherever the first dummy variable is used. The two examples illustrate the process. It is possible to get better optimization without the statement function, but the bulk of the optimization is done after the expression is expanded. This accounts for the rather slight difference during execution.

10.0 ARITHMETIC OPERATORS

10.1 SUMMARY

Some internal machine instructions are faster to execute than others. This is particularly true of fixed point operations. Multiplication and division are very slow, whereas addition and subtraction are quick. The difference in the floating point operations is most notable with the mutliply and divide, which are slow relative to addition and subtraction, but better than their fixed point equivalents.

Floating point multiplication is ordinarily better than repeated addition, but not when the quantity is to be doubled, then addition should be used. Fixed point addition should be used until four additions, rather than multiplying by the constant. Multiplication by constants is faster from five and up.

10.2.0 CODE COMPARISONS

10.2.1 Floating Point Addition Versus Multiplication

		Percent of Run	Seconds
46)!	X+X	1.72	8.55
47)	2.0*X	2.18	10.84
48)	X+X+X	4.29	21.32
49)!	3.0*X	2.74	13.61
50)	X+X+X+X	5.92	29.41
51)!	4.0*X	2.82	14.01
52)	X+X+X+X+X	7.39	36.72
53)!	5.0*X	3.18	15.80
54)	X+X+X+X+X+X	9.31	46.26
55)!	6.0*X	3.45	17.14

The first 10 examples were executed 100,000 times, and the percentages are for a total run time of 8.281 minutes. The results show clearly that to double a number, addition should be used. For any other quantity, the product is much faster than a repeated sum.

10.2.2 Fixed Point Addition and Multiplication

		Percent Run Time	Seconds
56)!	I+I	1.10	7.03
57)	2*I	2.50	15.97
58)!	I+I+I	1.34	8.57
59)	3*I	2.01	12.85

ARITHMETIC OPERATORS

		Percent of Run	Seconds
60)	I+I+I+I	2.02	12.92
61)!	4*I	1.98	12.66
62)	I+I+I+I+I	2.65	16.95
63)!	5*I	2.00	12.79
64)	I+I+I+I+I+I	2.99	19.12
65)!	6*I	2.08	13.30
66)	I+I+I+I+I+I+I	3.34	21.36
67)!	7*I	2.16	13.81
68)	I+I+I+I+I+I+I+I	3.81	24.36
69)!	8*I	2.07	13.24
70)	I+I+I+I+I+I+I+I+I	4.23	27.05
71)!	9*I	1.94	12.41
72)	I+I+I+I+I+I+I+I+I+I	4.72	30.18
73)!	10*I	2.10	13.43

Each of the fixed point tests was run 1.8 million times for a total CPU step charge of 10.658 minutes. All the multiplies were executed as multiplies and none as the faster internal instruction. When the multiplier is four or less, repeated addition would be used for best execution performance. Multiplication is preferred when the multiplier is five or more.

10.2.3 Multiplication With Constants of Powers of Two

		Percent of Run	Seconds
74)	I*16	0.94	5.52
75)	16*I	2.09	12.26
76)	I*32	1.01	5.92
77)	32*I	2.16	12.67
78)	I*64	1.11	6.51
79)	64*I	2.14	12.56
80)	I*128	1.10	6.45
81)	128*I	2.43	14.26
82)	I*256	1.80	10.56
83)	256*I	2.59	15.20

Multiplication tests by a constant whose value is a power of two were run for 9.780 minutes and 1.8 million passes through the main program loop. The first cases (74, 76, 78, 80, and 82) cause a shift to be used, whereas the second examples (75, 77, 79, 81, and 83) used the regular multiply. The results speak for themselves. The constant as the second operator appears to always generate the faster shift instruction.

13.3 HOW THE STATEMENTS ARE COMPILED

The compiler may not always generate the internal code exactly the way the FORTRAN program is written. Some subtraction of constants and variables, as well, is performed by loading the additive inverse of the value and adding, rather than subtracting, this reversed quantity. Very little difference is seen in the inverse addition. A notable difference is obtained if an integral quantity is multiplied by a constant, which is two to a positive power, (2, 4, 8, 16, 32, etc.). The compiler recognizes that this may also be done by a different and much quicker instruction. When the code is not complex, the order of the operands will make no difference. If the constant multiplier is placed second, the faster instruction will always be used.

11.0 SUBPROGRAMS

11.1 SUMMARY

Subprograms make the programmer's job easier and the coding more obvious to follow. They also shorten the program by allowing the same code to be executed from many places, rather than writing the same statement sequence many times. There is some overhead involved in transferring control from one subprogram to another. The transferring of variables back and forth between subprograms also requires time and in some cases extra memory for the storage of local variables.

It is advisable to keep the number of calls as small as possible and to keep the number of variables passed small, or even pass none at all. Passing arguments by location should be avoided. Loops which call subprograms should have non-loop dependent calls removed from the loop and use a temporary variable to hold the result(s). Subroutines should, when possible, have the loop placed in the routine and the call removed from the body of the loop. COMMON areas should be used to pass variables between subprograms. Simple variables should be first in a COMMON area followed by arrays. The simple variables are best allocated with the longest type first and the shortest last (COMPLEX*16 to LOGICAL*1). Arrays should be ordered with the one containing the smallest total size (the number of elements times the length of an element) first and the longest ones last.

If the types of variables and arrays are mixed, the programmer will have to ensure that each type starts on the proper boundary. COMPLEX*16 data addresses must be divisible by 16 and always end with a zero. Double word variables (COMPLEX*8 and REAL*8) have a starting address which is divisible by eight (address ends with a 0 or an 8), four byte data (REAL, INTEGER, or LOGICAL) must start at an address divisible by four (last digit of address is a 0, 4, 8, or C). INTEGER*2 data addresses must be divisible by two (last character a 0, 2, 4, 6, 8, A, C, or E). LOGICAL*1 data may fall on any address. When the longest type of variable is placed first, all addressing is properly compiled.

For easiest debugging and program maintenance, the dummy arguments in the SUBROUTINE and FUNCTION statements should, when possible, be called the same name as those in the CALL statement or reference to the function. This also applies to COMMON areas. Using a particular COMMON area is easiest to use when the same variable names appear in all references to that COMMON. EQUIVALENCE statements will allow sloppy coders the chance to change the names, but at the cost of increased complexity, confusion, and reduced optimization (see section 18).

11.2.0 CODE COMPARISON

All of the following examples (84 - 93) were executed nearly 10,000 times, requiring 21.835 minutes of CPU time. The subroutine and function subprograms used are illustrated in Examples 84 and 87 and are the same for Examples 85, 86, 88, and 89.

11.2.1 Subroutine CALL With No Arguments (Passed in Common)

No argument list was used to pass the arguments to the subroutine in this first example.

```
84)!      COMMON/ARGLST/A,B,C,D,E
           .
           .
           .
          CALL SUB
           .
           .
           .
          END
          SUBROUTINE SUB
          COMMON/ARGLST/A,B,C,D,E
          E=-100.0
          DO 1000 I=1,100
      100 E=(A*B-C)/D+A+E
          RETURN
          END
```

The call and subroutine execution time was 1.34 percent or 17.55 seconds. This used only 12 bytes for the CALL statement, and the subroutine took 264 bytes of memory.

11.2.2 Subroutine CALL With Arguments (Passed by Value)

When an argument list is passed, and the subroutine is the same, time increased to 1.38 percent or 18.08 seconds as in Example 85.

```
85)       CALL SUB(A,B,C,D,E)
           .
           .
           .
          END
          SUBROUTINE SUB(A,B,C,D,E)
           .
           .
           .
          RETURN
          END
```

This took 14 bytes for the CALL and 328 bytes for the subroutine. The values of the simple variables were passed to the subroutine and then restored to the locations used in the subroutine. This accounted for the extra time.

11.2.3 Subroutine CALL With Arguments (Passed by Location)

The call by location is the worst, Example 86, and took 2.48 percent (32.49 seconds). The CALL still takes 14 bytes for the instructions, but the subroutine now uses 340 bytes.

```
86)       CALL SUB(A,B,C,D,E)
           .
           .
           .
          END
          SUBROUTINE SUB(/A/,/B/,/C/,/D/,/E/)
           .
           .
           .
          RETURN
          END
```

The same 3 examples were run with functions; passing arguments through COMMON was the best as shown in Example 87. Example 88 was next best by passing arguments in the normal manner (by value). The worst case was the passing by location, Example 89. The functions were overall slower than the same code used in a subroutine.

11.2.4 Function Reference With No Arguments (Passed in Common)

```
87)!      COMMON/ARG/A,B,C,D,E
           .
           .
           .
          E=F(-100.0)
           .
           .
           .
          END
          FUNCTION F(X)
          COMMON /ARG/ A,B,C,D,E
          F=X
          DO 4000 I=1,100
     4000 F=(A*B-C)/D+A+F
          RETURN
          END
```

11.2.5 Function Reference With Arguments (Passed by Value)

The function is similar for Examples 88 and 89.

```
88)       E=F(A,B,C,D)
           .
           .
           .
          END
          FUNCTION F(A,B,C,D)
           .
           .
           .
          RETURN
          END
```

11.2.6 Function Reference With Arguments (Passed by Location)

```
89)       E=F(A,B,C,D)
           .
           .
           .
          END
          FUNCTION F(/A/,/B/,/C/,/D/)
           .
           .
           .
          RETURN
          END
```

		Summary of Examples		Memory Used		
Example	Type	% of Run Time	Seconds	Call	Subprogram	Total
84	subroutine common	1.34	17.55	12	264	276
85	subroutine value	1.38	18.08	14	328	342
86	subroutine location	2.48	32.49	14	340	354
87	function common	2.32	30.39	14	270	284
88	function value	2.63	34.46	14	312	326
89	function location	3.39	44.41	14	328	342

Figure 17 - Comparison of Subprogram Argument Passing

There are two common alternatives to external subprograms, which, while not retaining all the coding advantages, are somewhat quicker.

11.2.7 Statement Function

```
90)       SF(W,X,Y,X)=(W*X-Y)/Z+W
           .
           .
           .
          E=-100.0
          DO 7000 I=1,100
     7000 E=SF(A,B,C,D)+E
```

Example 90 uses a statement function to do the same simple calculation. The names in the statement function definition are dummy and act the same as dummy arguments for subprograms. This allows the flexibility of using different arguments. Each statement function reference is expanded in line. That is, the statement in the statement function definition is substituted for the statement function reference. With many statement functions, the program size will increase as each one is compiled. In Example 90 the in line expansion took 38 bytes and the whole example 58 bytes. The time was only 1.30 percent or 17.03 seconds.

11.2.8 Internal Routine

The quickest, and least flexible substitution for an external subprogram is by using a local section of code which is referenced with a simple GO TO and the return address ASSIGNed to a variable. The assigned GO TO is used to return from the shared code to the proper location. Example 91 required 36 bytes total.

```
91)       ASSIGN 8000 TO K
          GO TO 8001
     8000 ........
           .
           .
           .
     8001 E=-100.0
          DO 8002 I=1,100
     8002 E=(A*B-C)/D+A+E
          GO TO K,(8000, ....)
```

This took only 1.15 percent of the run time for 15.06 seconds for both the ASSIGN, the calculation, and the return.

11.2.9 Optional Return

When returning from a subroutine, the next FORTRAN source statement is usually executed. In some cases data dependent or extraordinary returns are needed. Either an index may be set, Example 92, or use of the conditional return, as in Example 93, may be used.

```
92)       DO 9001 II=1,100
          CALL SUB (A,B,II,J)
          IF (J.EQ.1) E=E*E/E
          IF (J.EQ.2) E=E*E/E
     9001 E=E*E/E
           .
           .
           .
          END
          SUBROUTINE SUB(A,B,II,J)
          A=(A*B)/B
          B=B+A+A
          J=(II/50)+1
          RETURN
          END
```

```
93)         DO 10000 II=1,100
            CALL SUB(A,B,II,&10001,&10002)
            GO TO 10003
      10001 E=E*E/E
            GO TO 10000
      10002 E=E*E/E
            GO TO 10000
      10003 E=E*E/E
      10000 E=E*E/E
             .
             .
             .
            END

            SUBROUTINE SUB(A,B,II,*,*)
            A=(A*B)/B
            B=B+A+A
            J=(II/50)+1
            IF (J.EQ.1) RETURN 1
            IF (J.EQ.2) RETURN 2
            RETURN
            END
```

The 'RETURN digit' form indicates which statement number position of the calling sequence is to be returned to. When the digit is not specified, or is greater than the number of statement numbers indicated in the CALL statement, the statement following the CALL is executed upon return from the subroutine. Otherwise, the statement number in the slot referenced by the digit is executed subsequent to the return. The net effect is a combined CALL and computed GO TO. While these two examples are very simple, a greater difference will be observed in practical use than is demonstrated here. Example 92 used 43.01 percent of the CPU time, or 563.47 seconds. Instructions for the CALL and associated statements were 90 bytes, the subroutine used 314 bytes, for a total of 404 bytes.

Example 93 used 527.05 seconds (40.23 percent) and used 104 bytes for the main program statements, 336 for the subroutine, totaling 440 bytes. The extra length is accounted for by the longer FORTRAN code in this illustration.

11.3.0 HOW THE STATEMENTS ARE COMPILED

11.3.1 Argument Lists

Subprograms which have an argument list will load an address for the arguments to be passed. Each entry in the list, whether array or simple variable, takes four bytes.

Another register is then loaded with the address of the subprogram and the branch taken to the subprogram. If the ID option of the compiler is on, which it is by default (NOID may be specified in

the PARM field of the EXEC JCL statement for the compile step to turn it off), a dummy branch instruction is inserted which contains the ISN (internal statement number) of the CALL statement or the function reference. It is used for debugging information and is printed in the traceback when a program error has occurred which the FORTRAN run time subroutines (the error monitor) trap. The ISN is also available in a dump.

11.3.2 No Argument List

Calling a subroutine with no argument list will load the register which was used for the argument address list pointer with zero which requires only two bytes and executes much faster than the load of the address which takes four bytes.

11.3.3 Returned Values

The single value returned from a function is returned in a register and is always stored. This is not dependent on the argument list but rather is the definition of a function.

Values returned from a subroutine are already in storage, and no additional action is required of the calling program.

11.3.4 Subroutine Initial Action

The initial internal action in the subroutine is different if an argument list exists or is absent. A subroutine always moves the values of all simple variables to a local area in the subroutine. This provides multi-programming capabilities in that the calling and called program units may each work with individual variables within their own workspaces and not interfere with the calculations in the other. Each variable requires eight bytes of memory for the instructions which move the values.

11.3.5 Subroutine Exit Processing

Returning from a subroutine moves the simple variables in the argument list whose values have changed, (appearing on the left side of an equal sign, and marked on the map with an S), back to the calling program's area. This requires eight bytes of memory for the instructions for each simple variable, as well as the storage space for the variables in the subroutine. There is also a constant four byte overhead. Returning from a subroutine with no argument list takes only the four bytes fixed overhead.

11.3.6 Argument Passing

Arrays passed between subroutines or functions and their calling program unit require much less overhead for initialization and clean up. The address of the array (along with the subscript) is passed and loaded by the subprogram to reference the specific location desired. This also requires eight bytes for each array passed.

When simple variables are passed by location (enclosing the dummy arguments in slashes), memory for local variables is saved. The corresponding increase in program time and subroutine length may not be worth that memory. Arrays are always passed by location. The subroutine will load the address of each of the simple arguments passed and store that address in a separate location, taking four bytes. When a value is needed, the address is loaded into a register and then a load from that address is executed. This may at times require extra loads, and the time increases with each reference and complexity of the program. The optimizer will attempt to reduce the number of times the address is loaded but is subject to the same guidelines given in the Common Expression Elimination section (8.0).

An array name in an argument list is represented by the address of the element given in the CALL statement. If only the name is used (no subscript(s)), the address of the beginning of the array is used. This may come in handy when dealing with a row of an array as shown below.

```
      DIMENSION X(100,50)
       .
       .
       .
      CALL ROWMLT(X(1,10),C,100)
       .
       .
       .
      SUBROUTINE ROWMLT(X,C,N)
      DIMENSION X(N)
      DO 1 I=1,N
    1 X(I) = X(I)*C
      RETURN
      END
```

Figure 18 - Passing a Row of an Array

The effect of subroutine ROWMLT is to multiply by C all of the named row in the second subscript of the CALL statement reference for X (the tenth row in this case). This technique is only applicable for rows. The elements in a multi-dimension array are in order in storage as a vector array, by the first index.

11.3.7 Statement Function Expansion

Statement functions are another way to reduce the overhead involved with external subprogram calling. Each occurrence of the statement function is expanded in-line and so requires more memory for its instructions, but the optimizer also has a chance to work on the internal machine language code generated in the proper program unit. (See also section 9.0 on Statement Functions.)

11.3.8 Internal Routine Reference

Assigned GO TO routines use the assigned variable to contain the return location so as to branch directly to the local code. The argument list processing time is saved. (Refer to the Branching section for a description of how the ASSIGN and assigned GO TO statements work, section 12.3.3.)

12.0 BRANCHING

12.1.0 SUMMARY

The normal sequential flow through a program can be altered conditionally or unconditionally. The conditional branching may depend on the value of a switch, a single variable, or the value of an expression. This is accomplished by the simple, computed or assigned GO TO, and the arithmetic or logical IF statements. The efficiency of each type of branching statement depends upon where the branch is performed and the location it is branching to.

12.1.1 Branching Statements Compared

The simple GO TO is the only unconditional branch and translates to one internal machine language instruction. Branching on a switch is best accomplished by the assigned GO TO. The next best testing is the arithmetic IF followed in execution speed by the logical IF, both having integer value compared to a constant. The arithmetic and logical IFs execute in very nearly the same time and are dependent on the statement order in the FORTRAN code. The branching statement with the most overhead is the computed GO TO. When used for its designed purpose, it is the quickest for the application.

12.1.2 Statement and Expression Ordering

Statement ordering is of importance and the following guidelines should produce the best coding possible. With assigned GO TO statements, the statements containing the referred statement numbers should not follow the assigned GO TO. The statement ordering is of no importance for the computed GO TO. Arithmetic IF statements should have the most often executed branch as the first statement number in the list of three. It should not directly follow the IF. Testing for a value of TRUE with a logical IF containing a simple expression (only one relational operator) is the best, and results in the execution of the appended statement. When a logical variable is tested, the best branching is done when the value is false and the appended statement is not executed. Complex expressions, using ANDs, ORs, and NOTs, are discussed later.

12.1.3 Index Branching

Checking an index or variable for a value may be accomplished by a series of arithmetic or logical IF statements or one computed GO TO. When the value of the index is to be six or less, a series of arithmetic IFs proved the best. When the value of the index may exceed seven, a computed GO TO statement is by far the best. A series of logical IF statements proved to be the worst of the three methods tested.

12.1.4 Complex Logical IFs

Complex logical IF statements with several NOTs, ANDs, or ORs should be avoided when possible; several separate IFs are better. Multiple conditions should be tested as described below. A series of relational operations with only AND operators should test for the condition which will fail first, causing fewer tests, and branch to the most often executed statement as the one following the IF. The appended statement should be the least often executed, i.e., the exceptional case. ANDs should be used when most of the conditions will be false.

When only OR operators are used with relational operands, the appended statement should be executed most often. The condition most often false, causing the branch to fall through, should be the last condition in the list. ORs are best used when the conditions are usually true and an extra statement is to be done. The relation tested (GT,GE,LT,LE,EQ,NE) causes no difference in execution speed.

The NOT operator used previously with relational operands or mixed ANDs and ORs will require the entire expression to be evaluated before any action is determined. No difference is apparent in either doing or skipping the appended statement. Using several simple IF statements to separate the ANDs and ORs improves the execution speed. NOTs are to be avoided.

Logical IF statements with logical variables (typed explicitly in a LOGICAL statement), with all ANDs or all ORs in a single expression, are executed the way outlined above for relational operands. NOT used with either operator (all ANDs or all ORs) in a statement merely tests the reverse true or false value, but the logic is the same as mentioned before. With a NOT and AND series the first test should cause the fall through. NOT with OR should cause the appended statement to be executed as soon as possible and the most likely true variable placed first. The NOT operator will not force the entire statement to be executed with logical variables. Mixed ANDs and ORs, with or without NOTs, will force the entire expression to be calculated before the final result can be analyzed to cause the execution of or skip the appended statement. There is little difference in which condition is tested.

12.1.5 Index Testing in a Loop

When a conditional logical IF is to be placed in a loop, the logical value part of the expression which does not change may be set outside the loop and tested in the loop. In logical IFs, most of the time is spent in the evaluation of the expression. This causes an improvement over the repeated calculations in the loop for each test. The variable should be declared in a LOGICAL type statement.

12.2.0 CODE COMPARISONS

12.2.1 Switch Setting and Testing

94)
```
ASSIGN 1001 TO K
 .
 .
 .
ASSIGN 1002 TO K
 .
 .
 .
GO TO K,(1001,1002)
```
Best when most often branched to statement does not follow GO TO.

95)
```
I=0
 .
 .
 .
I=1
 .
 .
 .
IF (I) 2001,2002,2001
```
Best when most often branched to statement does not follow IF.

96)!
```
I=0
 .
 .
 .
I=1
 .
 .
 .
IF (I.EQ.1) GO TO 3001
```
Best when branch is executed.

97)
```
I=0
 .
 .
 .
I=1
 .
 .
 .
IF (I.NE.1) GO TO 3002
```

98)
```
L=.FALSE.
 .
 .
 .
L=.TRUE.
 .
 .
 .
IF (L) GO TO 4001
```
Best when branch not taken.

```
99)     I=0
          .
          .
          .
        I=1
          .
          .
          .
        GO TO (5001),I

100)    I=1                          Best when most often executed
          .                          statement does not follow.
          .
          .
        I=2
          .
          .
          .
        GO TO (6001,6002),I
```

Example 94 shows setting and branching based on a switch. In the test program, one set and one branch were executed two million times and required only 0.5 percent of the 5.079 minute execution time for 1.52 seconds when the statement branched to did not follow the GO TO statement. When the branch was next to the target statement, 1.45 percent or 4.42 seconds were taken for the same 2,000,000 passes through the program. In this example, if K were ASSIGNed 1001, most of the time it would be better to have statement number 1001 not immediately follow the GO TO statement.

Example 95, using an arithmetic IF to test if a switch is zero or not took 0.48 percent of the execution time for 1.46 seconds when the most likely statement number did not follow the IF. When the most likely statement followed the IF, 1.26 percent, or 3.84 seconds were used for the two million passes.

Example 96 uses a logical IF statemett to test for a single value of a flag. This took 0.48 percent of the run time, 1.46 seconds, with the branch always executed. When the opposite condition was tested as in Example 97, 2.5 seconds (0.82 percent) were used when the branch was not taken and the following statement executed.

Example 98 shows setting a logical variable to TRUE (represented as non-zero, usually one) or false (zero). When the condition was FALSE, 3.07 percent or 9.36 seconds of the run time was used. If the condition is true, 3.38 percent of the run time was used, or 10.30 seconds.

Example 99 uses a computed GO TO to branch to a single statement number or fall through to the next sequential statement when the index is not one. The time required for this type of branch was 12.52 seconds or 4.11 percent of the same 5.079 CPU minute execution of the two million passes through the main loop.

Example 100 uses a computed GO TO. When the most likely statement does not follow the GO TO, 3.87 percent (11.79 seconds) was expended branching. When the statement which was most often executed branched to follow the GO TO statement, 4.43 percent (13.50 seconds) was spent in execution.

12.2.2 Simple Expression Testing

```
101)!       IF (I.LT.20) GO TO 7001
            L=L+1
             .
             .
             .
      7001  L=L+1

102)        IF (I.GE.20) GO TO 8002
            L=L+1
             .
             .
             .
      8002  L=L+1

103)        IF (L-20) 9001,9002,9003
      9003  L=L+1
            GO TO 9004
             .
             .
             .
      9002  L=L+1
            GO TO 9004

104)!       IF (I-20) 10001,10002,10003
     10001  L=L+1
            GO TO 10004
     10002  L=L+1
            GO TO 10004
     10003  L=L+1
            GO TO 10004

105)        GO TO (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1),I
            L=L+1
          1 .....

106)        ASSIGN 12001 TO K
            IF (I.LT.20) ASSIGN 12003 TO K
            GO TO K,(12001,12003)
     12001  L=L+1
             .
             .
             .
     12003  L=L+1
```

```
107)!       ASSIGN 13001 TO K
            IF (I.GE.20) ASSIGN 13003 TO K
            GO TO K,(13001,13003)
      13003 L=L+1
              .
              .
              .
      13001 L=L+1
```

All the above examples (101 - 107) were run 2,000,000 times for a total time of 5.079 minutes CPU time. The variable I varied from one to 2,000,000 so it was greater than 20 most of the time. The examples all do the equivalent calculations, except for Example 104 which shows the additional testing possible with the arithmetic IF.

Examples 101 and 102 show the best way to conditionally branch on the value of an expression. Example 101 is better when the condition falls through, and used 5.09 seconds, or 1.67 percent of the run time. When the branch was taken, as in Example 102, the time expended was 6.58 seconds or 2.16 percent.

The arithmetic IF statements were executed in very nearly the same time, but checked on the 'equal to' condition as well, and did more work in the same time as the logical IF. Example 103, where the most often executed statement did follow the testing of the expression, used 8.41 seconds, 2.76 percent of the step execution time. Example 104 with the most likely statement not following the test, expended 8.26 seconds or 2.71 percent of the run time.

Example 105 is included to show the unorthodox use of a computed GO TO. If only the first of a number of conditions is to be tested, this would be a possible way to code the test. The example took 3.74 percent of the time and was executed for 11.40 seconds. (Also see the following sub-section covering Index Branching, 12.2.5.)

Examples 106 and 107 use a flag set to determine the object of the branch. This sort of scheme would work best when the assigned GO TO was in the range of an inner loop and the ASSIGN and the LOGICAL IF were outside that loop. Example 107 would be the best choice. In this test case, both the GO TO and the ASSIGN with the logical IF statement were executed.

12.2.3.0 Complex Expression Testing

All of the examples (108 - 134) were executed 40,000,000 times, half with the conditions failing on the first test and half requiring all of the expression to be evaluated. This required 12.041 minutes of CPU time. Each type of testing was performed on relational and logical operands with the appended statement being a simple GO TO or adding one to an index. The results, in summary, indicated the best relational tests are done with a series of simple logical IF statements. Complex IFs with only ANDs or ORs are significantly worse. Mixed ANDs and ORs are worse yet and NOTs are at the bottom of the group tested. The relational operands were always slower than the logical operands.

The values used in the tests were such that B, C, D, E, and F were not changed from their value of ten. A was 11 or -11. L1 switched between .TRUE. and .FALSE. while L2, L3, L4, L5, and L6 were always .TRUE..

12.2.3.1 'NOT' With 'AND' Operators

```
108)     IF (.NOT.(A.GE.B).AND..NOT.(A.GE.C).AND..NOT.
        1(A.GE.D).AND..NOT.(A.GE.E).AND..NOT.(A.GE.F)) GO TO 14100

109)     IF (.NOT.(A.GE.B).AND..NOT.(A.GE.C).AND..NOT.
        1(A.GE.D).AND..NOT.(A.GE.E).AND..NOT.(A.GE.F)) I=I+1

110)!    IF (.NOT.L1.AND..NOT.L2.AND..NOT.L3.AND..NOT.L4.AND.
        1.NOT.L5.AND..NOT.L6) GO TO 14300

111)!    IF (.NOT.L1.AND..NOT.L2.AND..NOT.L3.AND..NOT.L4.AND.
        1.NOT.L5.AND..NOT.L6) I=I+1
```

The relational operators were slow, using 4.34 percent (31.35 seconds) for Example 108 (GO TO as the appended statement) and 4.20 percent (30.34 seconds for Example 109 (add as the appended statement)). The logical operands were faster; Examples 110 and 111 both used 2.44 percent, or 17.62 seconds of the 12.041 minutes CPU run time.

12.2.3.2 'NOT' With 'OR' Operators

```
112)     IF (.NOT.(A.LT.B).OR..NOT.(A.LT.C).OR..NOT.(A.LT.D).OR.
        1.NOT.(A.LT.E).OR..NOT.(A.LT.F)) GO TO 15100

113)     IF (.NOT.(A.LT.B).OR..NOT.(A.LT.C).OR..NOT.(A.LT.D).OR.
        1.NOT.(A.LT.E).OR..NOT.(A.LT.F)) I=I+1

114)     IF (.NOT.L1.OR..NOT.L2.OR..NOT.L3.OR..NOT.L4.OR.
        1.NOT.L5.OR..NOT.L6) GO TO 15300

115)!    IF (.NOT.L1.OR..NOT.L2.OR..NOT.L3.OR..NOT.L4.OR.
        1.NOT.L5.OR..NOT.L6) I=I+1
```

In these examples the unconditional GO TO as the appended statement of the relational operators was the quickest, with Example 112, 4.12 percent (29.76 seconds). The add, Example 113, 4.29 percent (30.99 seconds) was just slightly worse. The logical operators were faster, and the GO TO was the slower case. Example 114 used 2.53 percent (18.28 seconds), and Example 115, with an appended add, used 2.45 percent for 17.70 seconds.

12.2.3.3 Mixed 'AND' and 'OR' Operators

Mixed ANDs and ORs faired better than NOTs. No great amount of execution time difference was noted if the ANDs and ORs were grouped together as in Examples 116 - 119, or interspersed as in Examples 120 - 123.

```
116)     IF (A.GE.B.AND.A.GE.C.AND.A.GE.D.OR.A.LT.E.OR.
        1A.LT.F) GO TO 16100

117)     IF (A.GE.B.AND.A.GE.C.AND.A.GE.D.OR.A.LT.E.OR.
        1A.LT.F) I=I+1

118)!    IF (L1.AND.L2.AND.L3.AND.L4.OR.L5.OR.L6) GO TO 16300

119)     IF (L1.AND.L2.AND.L3.AND.L4.OR.L5.OR.L6) I=I+1

120)     IF (A.GE.B.AND.A.GE.C.OR.A.GE.D.AND.A.GE.E.OR.
        1A.LT.F) GO TO 17100

121)     IF (A.GE.B.AND.A.GE.C.OR.A.GE.D.AND.A.GE.E.OR.
        1A.LT.F) I=I+1

122)!    IF (L1.AND.L2.OR.L3.AND.L4.OR.L5.AND.L6) GO TO 17300

123)     IF (L1.AND.L2.OR.L3.AND.L4.OR.L5.AND.L6) I=I+1
```

With the ANDs and ORs separated by relational operators (Examples 116 and 117), both took the same time, 3.79 percent for 27.38 seconds of the total step time. The logical operators with the simple GO TO as the appended statement, Example 118, received 2.09 percent of the run time for 15.01 seconds. Example 119, logical operator with an appended add, took 2.10 percent (15.17 seconds) of the 12.041 minutes run time.

With the ANDs and ORs interspersed (Examples 120 - 123), the relational operators took 3.74 percent and 3.83 percent (27.01 and 27.67 seconds) with the appended GO TO, 120, and the appended add, 121, respectively. With the logical operators, the appended GO TO was again the slightly faster. Example 122 used 2.14 percent (15.46 seconds). Example 123, with the add statement, used 2.22 percent for 16.04 seconds of the measurement time.

12.2.3.4 Separate 'AND' and 'OR' Operators

The usage of ANDs or ORs exclusively in a statement was better yet than mixed operators.

```
124)     IF (A.GE.B.AND.A.GE.C.AND.A.GE.D.AND.A.GE.E.AND.
        1A.GE.F) GO TO 18100

125)     IF (A.GE.B.AND.A.GE.C.AND.A.GE.D.AND.A.GE.E.AND.
        1A.GE.F) I=I+1

126)     IF (L1.AND.L2.AND.L3.AND.L4.AND.L5.AND.L6) GO TO 18300

127)     IF (L1.AND.L2.AND.L3.AND.L4.AND.L5.AND.L6) I=I+1
```

```
128)     IF (A.LT.B.OR.A.LT.C.OR.A.LT.D.OR.A.LT.E.OR.
        1A.LT.F) GO TO 19100

129)     IF (A.LT.B.OR.A.LT.C.OR.A.LT.D.OR.A.LT.E.OR.
        1A.LT.F) I=I+1

130)     IF (L1.OR.L2.OR.L3.OR.L4.OR.L5.OR.L6) GO TO 19300

131)     IF (L1.OR.L2.OR.L3.OR.L4.OR.L5.OR.L6) I=I+1
```

Examples 124 and 125 used 2.22 percent (16.04 seconds) and 2.37 percent (17.12 seconds) for the two relational tests for the simple AND case. The OR examples took 2.21 percent (15.96 seconds) for Example 128, and Example 129 took 2.52 percent (18.21 seconds). The logical operators were as usual faster. Example 126 used 2.04 percent or 14.74 seconds, and Example 127 used 2.19 percent or 15.82 seconds. The logical operands with the ORs took 2.02 percent, or 14.59 seconds for Example 130 and 2.25 percent, or 16.25 seconds for Example 131.

12.2.3.5 Separate IF Statements for 'AND'

The best testing for relational operators was achieved by separating each test into a series of separate IF statements.

```
132)          IF (A.GE.B) GO TO 20000
              IF (A.GE.C) GO TO 20000
              IF (A.GE.D) GO TO 20000
              IF (A.GE.E) GO TO 20000
              IF (A.GE.F) GO TO 20000
              I=I+1
        20000 .....

133)!         IF (A.GE.B) GO TO 21000
              IF (A.GE.C) GO TO 21000
              IF (A.GE.D) GO TO 21000
              IF (A.GE.E) GO TO 21000
              IF (A.GE.F) GO TO 21000
              GO TO 21001
        21000 I=I+1
        21001 .....

134)          IF (A.GE.B) GO TO 22000
              GO TO 22005
        22000 IF (A.GE.C) GO TO 22001
              GO TO 22005
        22001 IF (A.GE.D) GO TO 22002
              GO TO 22005
        22002 IF (A.GE.E) GO TO 22003
              GO TO 22005
        22003 IF (A.GE.F) GO TO 22004
              GO TO 22005
        22004 I=I+1
        22005 .....
```

These three examples show the separate tests for AND, OR, and a poor case of AND respectively. Example 132 is significantly the best ANDing with 1.49 percent or 10.76 seconds of the 12.041 minute execution time. The separate ORing, as shown in Example 133, used 1.46 percent or 10.55 seconds of the same total run time. Example 134 shows a bad case of using separate IFs to accomplish an AND. This required 2.67 percent for 19.29 seconds.

12.2.4 Multiple IF Statements

```
135)!       IF (I.LT.20) GO TO 100
            L=L+1
            L=L+2
            L=L+3
            L=L+4
            L=L+5
        100 .....

136)        IF (I.GE.20) L=L+1
            IF (I.GE.20) L=L+2
            IF (I.GE.20) L=L+3
            IF (I.GE.20) L=L+4
            IF (I.GE.20) L=L+5
```

These two examples were also executed 2,000,000 times for a total charge of 5.079 minutes for the measurement step. The variable I ranged from one to 2,000,000 so that the addition statements were executed most of the time. Example 135 was better with 5.74 percent, or 17.49 seconds. Example 136 used almost twice the time with 10.15 percent for 30.95 seconds. While these two examples may look ridiculous, programs which are often and carelessly modified may contain some symptoms of the above examples.

12.2.5 Index Branching

```
137)        IF (J.EQ.1) GO TO 25101
            IF (J.EQ.2) GO TO 25102
            IF (J.EQ.3) GO TO 25103
            IF (J.EQ.4) GO TO 25104
              .
              .
              .
      25101 .......
              .
              .
      25102 .......
              .
              .
      25103 .......
              .
              .
      25104 .......
```

```
138)!       IF (J-1) 25240,25201,25210
      25210 IF (J-2) 25202,25202,25220
      25220 IF (J-3) 25203,25203,25230
      25230 IF (J-4) 25204,25204,25240
      25240 .......
              .
              .
      25201 .......
              .
              .
      25202 .......
              .
              .
      25203 .......
              .
              .
      25204 .......

139)        GO TO (25301,25302,25303,25304), J
              .
              .
              .
      25301 .......
              .
              .
      25302 .......
              .
              .
      25303 .......
              .
              .
      25304 .......

140)        IF (J.EQ.1) GO TO 26101
            IF (J.EQ.2) GO TO 26102
            IF (J.EQ.3) GO TO 26103
            IF (J.EQ.4) GO TO 26104
            IF (J.EQ.5) GO TO 26105
            IF (J.EQ.6) GO TO 26106
      26101 .......
              .
              .
      26102 .......
              .
              .
      26103 .......
              .
              .
      26104 .......
              .
              .
      26105 .......
              .
              .
      26106 .......
              .
              .
```

```
141)!       IF (J-1) 26260,26201,26210
      26210 IF (J-2) 26202,26202,26220
      26220 IF (J-3) 26203,26203,26230
      26230 IF (J-4) 26204,26204,26240
      26240 IF (J-5) 26205,26205,26250
      26250 IF (J-6) 26206,26206,26260
      26260 .......
                .
                .
      26201 .......
                .
                .
      26202 .......
                .
                .
      26203 .......
                .
                .
      26204 .......
                .
                .
      26205 .......
                .
                .
      26206 .......

142)        GO TO (26301,26302,26303,26304,26305,26306), J
                .
                .
                .
      26301 .......
                .
                .
      26302 .......
                .
                .
      26303 .......
                .
                .
      26304 .......
                .
                .
      26305 .......
                .
                .
      26306 .......

143)        IF (J.EQ.1) GO TO 27101
            IF (J.EQ.2) GO TO 27102
            IF (J.EQ.3) GO TO 27103
            IF (J.EQ.4) GO TO 27104
            IF (J.EQ.5) GO TO 27105
            IF (J.EQ.6) GO TO 27106
            IF (J.EQ.7) GO TO 27107
            IF (J.EQ.8) GO TO 27108
```

```
                .
                .
                .
          27101 .......
                .
                .
          27102 .......
                .
                .
          27103 .......
                .
                .
          27104 .......
                .
                .
          27105 .......
                .
                .
          27106 .......
                .
                .
          27107 .......
                .
                .
          27108 .......
                .
                .

144)            IF (J-1) 27280,27201,27210
          27210 IF (J-2) 27202,27202,27220
          27220 IF (J-3) 27203,27203,27230
          27230 IF (J-4) 27204,27204,27240
          27240 IF (J-5) 27205,27205,27250
          27250 IF (J-6) 27206,27206,27260
          27260 IF (J-7) 27207,27207,27270
          27270 IF (J-8) 27208,27208,27280
          27280 .......
                .
                .
          27201 .......
                .
                .
          27202 .......
                .
                .
          27203 .......
                .
                .
          27204 .......
                .
                .
          27205 .......
                .
                .
```

```
        27206 .......
                .
                .
        27207 .......
                .
                .
        27208 .......
                .
                .
145)!           GO TO (27301,27302,27303,27304,27305,27306,27307,27308), J
                .
                .
                .
        27301 .......
                .
                .
        27302 .......
                .
                .
        27303 .......
                .
                .
        27304 .......
                .
                .
        27305 .......
                .
                .
        27306 .......
                .
                .
        27307 .......
                .
                .
        27308 .......
                .
                .
```

Examples 137 to 139 were executed 1.8 million times for a total time of 9.149 minutes. Example 137 took 10.91 percent of that time for 59.89 seconds. The multiple arithmetic IF statements in Example 138 took only 8.40 percent and 46.11 seconds and are always better than multiple logical IF statements. Example 139 used 61.15 seconds or 11.14 percent of the same run time.

Examples 140 to 142 executed each type of branch, in groups of six, 40,000 times. This required 3.999 minutes for the measurement step. The computed GO TO (Example 142) is still the worst with 31.67 seconds (13.20 percent). Example 140 was the next best and took 29.10 seconds (12.13 percent). Example 141 used 27.16 seconds (11.32 percent).

Examples 143 through 145 test for eight specific values, and the computed GO TO statement is now the best (Example 145, 11.18 percent, 26.36 seconds). Each of the examples 143 - 145 was executed 300,000 times for a total test program execution time of 3.929 minutes. The arithmetic IFs (Example 144, 14.22 percent, 33.52 seconds) are still better than the logical IFs (Example 143, 15.77 percent, 37.18 seconds).

The savings are more dramatic as the break at seven tests is moved away from. Examples 95, 96, 97, and 100 show the result of one or two comparisons. A test was also run with ten index values. The 200,000 executions took 16.370 minutes for the test program. The logical IF statements required 3.32 percent, or 32.61 seconds. The arithmetic IFs used 2.86 percent or 28.10 seconds. Significantly better was the computed GO TO which required 1.93 percent of the run time for 18.96 seconds.

12.2.6 Expression Reduction of Complex Logical IF Statements

There are several ways to evaluate logical IFs which test conditions. The simplest condition setting, using the rules outlined in the opening paragraphs of this section, should be followed.

```
146)        IF (.NOT.(L.GT.M).AND..NOT.(I.GT.J)) K=.FALSE.
```

This example is the worst way to set K and took 5.52 percent, or 32.39 seconds, of the 9.780 minute CPU time for 1.8 million executions. The examples 146 through 152 were also executed the same number of times, and the total run time was also 9.870 minutes. The values of L and I in each example were negative to start with and reversed each pass through the loop so there was no preferred order to influence the tests. Example 146 evaluated the entire expression and required two NOTs.

```
147)        IF (.NOT.(L.GT.M.OR.I.GT.J)) K=.FALSE.
```

This took 5.15 percent of the run time (30.22 seconds). The saving was achieved by only one NOT operator being used.

```
148)        IF (L.GT.M) GO TO 21001
            IF (I.GT.J) GO TO 21001
            GO TO 21002
      21001 K=.TRUE.
      21002 ........
```

Here the two conditions are split apart, and the time is reduced to 13.14 seconds, 2.24 percent of the 9.870 minute run time.

```
149)        IF (L.LE.M) GO TO 22001
            GO TO 22002
      22001 IF (I.LE.J) K=.FALSE.
      22002 ........
```

Example 149 requires somewhat less branching and shows a minimal improvement over Example 148. This test took 2.19 percent for 12.85 seconds.

```
150)        IF (L.GT.M.OR.I.GT.J) K=.TRUE.
```

Example 150 combines the two statements and does not use ANDs or NOTs. This took 2.18 percent of the run time for 12.79 seconds.

```
151)!       IF (L.LE.M.AND.I.LE.J) K=.FALSE.
```

This, the best (2.07 percent, 12.14 seconds), is the preferred method.

When the values are not changed in the loop, some terms of a logical expression may be calculated outside the loop at a significant saving.

```
152)        LOGICAL K1
              .
              .
              .
            K1= L.EQ.M.AND.I.LE.J
              .
              .
              .
            DO 23001 N=1,NN
              .
              .
              .
            IF (K1) K=.FALSE.
```

Example 152 required 9.56 seconds (1.63 percent) of the 1.8 million executions for the IF statement alone. The combination of both statements was also measured at 29.10 seconds, or 4.96 percent.

12.3.0 HOW THE STATEMENTS ARE COMPILED

12.3.1 Machine Language Branching

All conditional branching statements result in the setting of two bits in the program status work (PSW), called the condition code, and testing their value. There are 16 test combinations of the four possible condition code values which may be tested for (zero to 15). A conditional branch checks which bits are on and takes action from there. If no bits are to be checked, the branch is never taken, and the next sequential instruction is always executed. An unconditional branch to the specified address is taken when the check asks all combinations to be checked. All other combinations of bits specified in the conditional branch must match the condition code value in the PSW field before the branch is taken. If the condition code does not match the combination of bits, instruction processing 'falls through' to the next instruction in sequence.

12.3.2 Simple GO TO Statement

The unconditional GO TO statement is a single unconditional branch instruction to the correct location. This requires four bytes.

12.3.3 ASSIGN and Assigned GO TO Statements

Each of these statements use eight bytes. The ASSIGN statement loads the address of the statement which is specified by its statement number to a register and then stores it in the variable location named. The assigned GO TO statement loads the address for the branch from the variable location to a register and then uses an unconditional branch to the address in the register.

12.3.4 Computed GO TO Statement

The processing of a computed GO TO statement first checks the value of the index to see if its value exceeds the number of executable statement numbers in the FORTRAN statement. The constant of the number of statement numbers in the list is loaded into a register as is the index. These are compared, and the condition code set. The index is then always shifted left two places, causing a multiply by four, for use when the index is acceptable. The conditional branch is set to branch to the statement following the computed GO TO statement (fall through) when the index is greater than the number of statement numbers in the list. Assuming the index is in the proper range, the modified index is used to space down the proper number of entries into a table which contains the addresses of the statements named in the list by their statement numbers. Each address entry is four bytes long--the reason for the shift. When the index is one, the address four bytes from the beginning of the list is used; when the value is two, the address eight bytes down is used, etc. When the index is zero, the address at the start of the list is used. This first address points to the statement which follows the computed GO TO in the FORTRAN source. No checking is done for a negative index, and it probably will cause an addressing error. The processing required for this statement is somewhat involved and accounts for its slowness when the range of the index is small. It has a fixed number of instructions (24 bytes) which are executed no matter how long the list of statement numbers is. This is advantageous when many values of the index are possible. The length of the address list is four bytes plus four times the number of entries in the computed GO TO list. This list is created for every computed GO TO even if the list is the same in more than one statement.

12.3.5 Arithmetic IF Statement

After the expression which is enclosed in parentheses is evaluated, the condition code is tested according to the pattern of the statement numbers.

The order in which the condition code is tested depends on the statements which follow the arithmetic IF. When all the statement numbers are different and one follows the IF statement, that condition is not checked. Otherwise the checking procedure follows from left to right. The following illustration should make the point clear:

IF (I-1) 1,2,3	IF (I-1) 1,2,3	IF (I-1) 1,2,3
1	2	3
2	1	2
3	3	1
test and branch on equal to or greater than; fall through for less than	test and branch on less and greater than; fall through on equal to	test and branch on less than and equal to; fall through when greater than

Figure 19 - Arithmetic IF Statement Ordering

When one of the named statement numbers follows, a conditional branch is used to check all three possible condition code settings. When two of the statements numbers are the same, the checking is again order dependent, but the position with the repeated statement numbers encountered first, from right to left, has the condition code test altered to reflect the dual code. If the statement which follows the IF is numbered as one of the targets from the IF, that condition code(s) is not checked and becomes the fall through condition.

For these reasons it is best to avoid placing the most often branched to statement after the IF statement. The condition should also be set up so that the most often branched to statement number is the one which occurs first in the statement number list that does not follow the IF statement.

The setting of the condition code may be done through the arithmetic statements. In the three illustrations shown above, the subtraction will set the condition code. This is true for any expression. If a single variable is to be checked, a special instruction whose main purpose is to set the condition code is used. This instruction uses a register and does not access memory (although the initial load of the register may obtain the value from storage). The condition code checking branches are then executed, no matter how the condition code was set.

12.3.6.0 Logical IF Statement

12.3.6.1 Single Expression

The logic is the same for relational operators or logical variables when only ANDs or only ORs are used in a single expression. ANDs, for relational operators, test on the reverse condition from that coded. The branch from each test is to the statement which followed the IF in the FORTRAN source code. Only when all the conditions are met is the appended statement executed. This logic also holds true for logical variables preceded with NOT.

As each relational operator is evaluated, the condition code is set and tested. If the condition code testing is true, the rest of the testing is skipped and the following statement executed. When the appended statement is a simple GO TO, the final test in the series is as coded and the appended GO TO executed. The following statement is then 'fallen through' from the unsuccessful last test.

Logical operands use the same instruction which is used for controlling short running or simple DO loops. First a register is cleared (to zero), and the value to be tested is placed in another register. The test is done (not using the condition code); and when the index is zero (non-zero for NOT), the next statement is branched to. When the index is non-zero (zero for NOT), the next condition is tested. When the appended statement is a simple GO TO, the final test is reversed and causes the appended GO TO to be taken.

12.3.6.2 Multiple Expressions

Multiple ORs in an IF statement try to branch to the appended statement after each test of the relational operators. When the appended statement is an unconditional GO TO, the branch is to the specified statement rather than a separate GO TO statement. When the appended statement is not a GO TO, the last test is reversed and the statement following the IF in the FORTRAN source is branched to if the condition code is matched, skipping the appended statement. Ordinarily the condition code is tested as written in the FORTRAN program. Logical operators with a string of ORs use the same logic as described for relational operators, but the instructions are those as described for AND operators. A NOT prefix only changes the instruction used for testing the index and not the logic flow.

Using NOT with the relational operators forces the entire expression to be evaluated before the appended statement is executed or skipped. The relational operator is evaluated and a zero or one loaded into a register to record the true or false result, respectively, of each relational test. The NOT operator causes the resulting value of the relational test to have one subtracted from the value and then the number reversed (1s complemented) in the register. The results of these two operations are ANDed or ORed together and the final result tested. This last test, to execute the appended statement or skip its execution, uses the same instructions as the logical variable does. When the appended statement is a simple GO TO, the test will branch to the named statement number when the final truth value is true. The branch is to skip the appended statement with a false condition in any other case.

Compound and mixed ANDs and ORs with logical variable operands use the machine language instructions. The final result for logical variables is evaluated as outlined previously. NOT is evaluated with the same two instructions used for relational operators, without using the condition code testing.

13.0 INPUT/OUTPUT

13.1 SUMMARY

Passing information between external storage and the processing unit is the slowest operation of a computer program. Significant reductions in time (CPU, I/O, as well as wall clock) can be realized if a little care and forethought is exercised. Data which is intermediate, used only by programs and not viewed by human readers, should be kept in internal form, i.e., not formatted, using either FORTRAN or FTIO, described in Appendix D. Data to be presented for human consumption should be kept as simple as possible, the list of variables short, and formatting instructions explicit. Unformatted direct access, or random access I/O should be used only when required. The advantages of DAIO and FORTXDAM (Appendix C) should be explored.

13.2.0 CODE COMPARISONS

The following tests, unless otherwise noted, were made by placing a READ statement in a loop which was executed 25 times. The READ consistently transferred the equivalent of an array 20 by 1000 of the same floating point numbers. Test results with WRITE statements show essentially the same results, with some changes in the time spent formatting the data.

13.2.1.0 Formatted I/O

13.2.1.1 Element Transfer

153)
```
      DIMENSION A(20,1000)
          .
          .
          .
      DO 10 N=1,25
      DO 10 I=1,20
      DO 10 J=1,1000
   10 READ (10,20) A(I,J)
   20 FORMAT (F4.1)
```

In this example each element of the array is read individually, and the I/O routines are called for each element. This required 7.190 minutes CPU time and 9.453 minutes I/O time.

13.2.1.2 Row Transfer by Implied Loop

154)
```
      DIMENSION A(20,1000)
          .
          .
          .
      DO 10 N=1,25
      DO 10 I=1,1000
   10 READ (10,20) (A(J,I),J=1,20)
   20 FORMAT (20F4.1)
```

The use of an implied loop reduced the calls to the library routines and took half the CPU time, 3.189 minutes and saved over 15 times the I/O charges, 0.538 minutes.

13.2.1.3 Row Transfer by Subroutine

```
155)!            DIMENSION A(20),AA(20,1000)
                      .
                      .
                      .
                 DO 10 N=1,25
                 DO 10 I=1,1000
              10 READ (10,20) A
                 CALL FMOVE(AA(1,I),80,A)
              20 FORMAT (20F4.1)
```

The amount of overhead involved with the implied DO loop is again reduced, and the intent of the READ clearer to the library functions. The call to FMOVE makes the examples exactly alike and required 0.16 seconds CPU time. The same number of calls to the I/O support routines were made, but this method used only 2.700, excluding FMOVE, CPU minutes and 0.535 I/O time. A savings of 6 percent in CPU time and no change (<1 percent) in the I/O time over the implied loop.

13.2.1.4 Array Transfer by Name

```
156)             DIMENSION A(20,1000)
                      .
                      .
                      .
                 DO 10 N=1,25
              10 READ (10,20) A
              20 FORMAT (20F4.1)
```

This simplest setup uses 2.765 minutes of CPU time and 0.538 I/O minutes. These are essentially the same as before but with less coding, and all the data has been placed in the proper location in the array. The slight increase in CPU time is probably attributed to the generation of the second subscript being less efficient when implied. It should be specified.

The double implied loop is slower as shown in Example 5.

13.2.1.5 Array Transfer by Implied Loop

```
157)             DIMENSION A(20,1000)
                      .
                      .
                      .
                 DO 10 N=1,25
              10 READ (10,20) ((A(I,J),I=1,20),J=1,1000)
              20 FORMAT (20F4.1)
```

This used 3.161 minutes CPU time and 0.543 I/O minutes.

13.2.1.6 Effect of JCL

The DCB parameters coded on the DD statement for the file affect the I/O times as shown in the following example. Examples 153 through 157 were run using:

```
DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

Example 158 uses the same code as Example 156 but has:

```
158)   DCB=(RECFM=F,BLKSIZE=80)
```

The CPU time used was 2.852 minutes, a 3 percent increase, but the I/O time was increased by 13.208 minutes, a 2400 percent rise. All I/O should be blocked if possible. In this case the additional memory required to contain the buffer was 6240 bytes, not a considerable amount.

13.2.2.0 Unformatted I/O

Examples 155 and 156 were also run without formatting and showed dramatic savings as illustrated below in Examples 159 and 160. Example 159 has the call to FMOVE to make the results of both examples exactly the same. The call accounted for 1.4 CPU seconds.

13.2.2.1 Row Transfer by Subroutine

```
159)            DIMENSION A(20),AA(20,1000)
                    .
                    .
                    .
                DO 10 N=1,25
                DO 10 I=1,1000
                READ (10) A
              10 CALL FMOVE (AA(1,I),80,A)
```

13.2.2.2 Array Transfer by Name

```
160)!           DIMENSION A(20,1000)
                    .
                    .
                    .
                DO 10 N=1,25
              10 READ (10) A
```

Example 159 required 0.265 total CPU minutes and 0.537 minutes I/O time, saving 11 times the CPU as Example 155 and 11 percent of the I/O time. Example 160 displayed more spectacular savings. CPU time was reduced to 0.097 minutes, a saving of 96 percent (or 30 times faster), and the I/O time of 0.498 minutes is a reduction of 14 percent.

13.2.3 Simplifying I/O Lists

The same savings realized by reading in an array with a single item in the I/O list can also be used to read in various kinds of data. The data is read into an array and EQUIVALENCEd to the proper variables as shown in Examples 161 and 162.

161) Long I/O List

```
      DO 10 N=1,25
      DO 10 I=1,1000
 10   READ (10,20)A1,B1,C1,D1,E1,F1,A2,B2,C2,D2,E2,F2,
     1A3,B3,C3,D3,E3,F3,A4,B4,C4,D4,E4,F4
 20   FORMAT (F4.1,A4,A3,Z3,1X,I3,A2,F4.1,A4,A3,Z3,
     11X,I3,A2,F4.1,A4,A3,Z3,1X,I3,A2,F4.1,A4,A3,
     2Z3,1X,I3,A2)
```

162)! Array for Long I/O List

```
      DIMENSION A(24)
      EQUIVALENCE (A(1),A1),(A(2),B1),(A(3),C1),(A(4),D1),(A(5),E1),
     1(A(6),F1),(A(7),A2),(A(8),B2),(A(9),C2),(A(10),D2),(A(11),E2),
     2(A(12),F2),(A(13),A3),(A(14),B3),(A(15),C3),(A(16),D3),(A(17),E3),
     3(A(18),F3),(A(19),A4),(A(20),B4),(A(21),C4),(A(22),D4),
     4(A(23),E4),(A(24),F4)
          .
          .
          .
      DO 10 N=1,25
      DO 10 I=1,1000
 10   READ (10,20) A
 20   FORMAT (F4.1,A4,A3,Z3,1X,I3,A2,F4.1,A4,A3,Z3,1X,I3,A2,
     1F4.1,A4,A3,Z3,1X,I3,A2,F4.1,A4,A3,Z3,1X,I3,A2)
```

Example 161 took 2.199 CPU minutes whereas Example 162 used 1.756 minutes, saving 0.443 minutes or 20 percent. The I/O time was identical.

13.2.4 Variable or Execution Time Formats

Variable formats are useful but can be expensive. The following is the same as Example 161 but uses a dynamic format.

163)

```
      DIMENSION A(20,1000)
      REAL*8 FMT(12)
      DATA FMT/'(F4.1,A4',',A3,Z3,1','X,I3,A2,','F4.1,A4,',
     1'A3,Z3,1X',',I3,A2,F,'4.1,A4,A','3,Z3,1X,',','I3,A2,F4',
     2'.1,A4,A3',',Z3,1X,I','3,A2)'/
          .
          .
          .
      DO 10 N=1,25
 10 READ (10,FMT)A1,B1,C1,D1,E1,F1,A2,B2,C2,D2,E2,F2
     1A3,B3,C3,D3,E3,F3,A4,B4,C4,D4,E4,F4
```

Example 163 took 3.550 CPU minutes and 0.558 I/O minutes compared to 2.199 and 0.540 for Example 161. This is an increase of 38 percent.

```
164)          DIMENSION A(20,1000)
              DO 10 N=1,25
              DO 10 I=1,1000
           10 CALL FREAD (A(1,I),10,80,&99,&98)
```

This required 0.143 minutes CPU time and 0.531 minutes I/O time, a savings of 33 percent CPU over Example 159. FTIO, which performs unformatted I/O, is described in Appendix D.

13.2.5 Direct Access I/O

Direct access I/O was tested in a program which wrote 1000 records, each 7200 bytes, to each of four different data sets. The records were checked for accuracy by comparing the first element read with the calculated value.

Three I/O packages were tested: FORTRAN, DAIO (a locally written replacement package), and FORTXDAM (an IBM written asynchronous I/O package, see Appendix E). DAIO provides the same direct access I/O functions as FORTRAN, see Appendix D. FORTXDAM enables the user to start an I/O operation and then resume his program processing. The calling program must pause when the data being read is to be used, or the data being written is to be changed, until the I/O operation is complete. The test program was altered to go round-robin between the four files and to keep count of the number of times calculations could have been done while waiting for I/O operations to complete. Each file maintained its own buffer as a program array for reading and writing. The results of the tests are shown below.

		CPU MIN	I/O MIN	% DATA SET WAIT	MEMORY USED BYTES
165)	FORTRAN	2.341	10.799	82.40	146K
166)	DAIO	0.707	7.236	93.25	78K
167)	FORTXDAM	6.257	10.389	27.80	78K

Figure 20 - Direct Access I/O Comparison

The data set wait figure indicates the percent of time the program was waiting on a busy device.

For general purpose use, DAIO shows significantly better I/O time and remarkably better CPU time than FORTRAN. FORTXDAM was marginally better at I/O than FORTRAN, 3 percent, but was able to do over 2.62 CPU minutes of other work. The PPE reports show 57 percent of the CPU time spent in FORTXDAM and 42 percent in the main program. The wait counters in the program totalled 10,123,841 for both read and write for all four files indicating the number of times additional CPU work could have been done.

13.3 HOW THE STATEMENTS ARE COMPILED

Any I/O request causes a call to IBCOM#, an entry in IHCECOMH, which is the extended communications handler. Each item in the I/O list generates a call, hence, reducing the number of variables in the I/O list causes less CPU time to be used. IHCECOMH processes information between the user's program and the I/O device by utilizing other I/O package programs to call the system and data management services. Encoded with each call to IHCECOMH is information about the options in use, END= or ERR=, the buffer location, and memory location, descriptions and formatting, if required. IHCEFIOS (extended FORTRAN input/output system) is the interface module to the system supplied data management routines for sequential reading, writing, and file positioning. IHCEDIOS (extended direct access input/output system) does the same job as IHCEFIOS for direct access data transfer. If formatting is required, IHCFCVTH translates the data and moves it, otherwise IHCECOMH moves the data. Other modules used are IHCERRM, the error monitor, which is called when an error occurs to print the messages. It also determines what options for recovery have been set by looking in IHCUOPT (user option table). IHCEFNTH is used to patch up arithmetic errors such as overflow, underflow, and divide checks. If trace back information is to be printed, IHCETRCH prints this information. The table of default unit information for READ, PRINT, and PUNCH statements as well as file descriptor information and buffer addresses are held in IHCUATBL (unit assignment table).

Formatting time can be considerable. Print space which is not used should be skipped by using the X format specification, not wide format fields. In order, by the quickest formatting conversion routine, first are alphameric, hexadecimal, logical, integer and floating point (F, E, D, G, and C all nearly the same). Variable formats require more time for processing during execution for data to be transferred. Each time an I/O statement is executed the format is verified and translated to internal code.

The breakdown of time spent in each module for the different examples is given on the next page.

Example	Seconds MAIN	Seconds IHCECOMH	Seconds IHCFCVTH	Seconds IHCEFIOS	Seconds IHCFIOS2	Minutes CPU	Minutes I/O*
153	4	131	134	111	31	7.190	9.453
154	4	50	129	6	2	3.198	0.538
155	0	23	128	5	2	2.722	0.535
156	0	21	135	4	2	2.765	0.538
157	7	46	130	4	2	3.161	0.543
158	0	15	104	5	1	2.852	13.498
159	0	4	0	6	2	0.265	0.537
160	0	8	0	2	2	0.097	0.498
161	1	71	52	6	2	2.199	0.540
162	0	44	53	5	2	1.765	0.540
163	1	152	52	1	2	3.550	0.558
164	0	FREAD = 6	0	0	0	0.143	0.531

		Seconds MAIN	Seconds I/O Handler	Seconds System Routines	Minutes CPU	Minutes I/O	% Data Set Wait
165	FORTRAN	37	60	36	2.341	10.799	82.40
166	DAIO	28	13	0	0.707	7.236	93.25
167	FORTXDAM	160	214	1	6.237	10.389	27.80

* Adjusted to show only I/O charges to input tape. The time to write the measurement tape has been subtracted from the reported time.

Figure 21 - Summary of I/O Examples

The direct access routines read a specified record from direct access storage. The records may be read or written in any order and do not require spacing over the previous records as would have to be done with sequential operations. At the front of each record is identifying information. This is used to verify that the proper record is being read or written and insure that the entire record is transferred. FORTRAN and FORTXDAM require that the entire data set is preformatted. FORTRAN does this automatically, and FORTXDAM requires a special call to be made by the user. DAIO gains some of its savings by only formatting those records actually used.

FORTXDAM should have the data sets it accesses on separate channels so it is physically possible to access the data sets simultaneously without interfering with other accesses. This is done by coding SEP=ddname in the UNIT field of the DD statement. After the new data sets have been formatted, I/O operations are started. If planning has been done carefully, then the calling program should be able to do other processing while waiting for the I/O operation to be completed. For read operations the data is not available until the completion of the request, and for writing, the data should not be altered. When all other work is done, the state of the I/O request may be tested or the calling program placed in a wait state until the completion of the I/O request. The main program will be restarted automatically in the latter case. With some thought and programming utilizing double buffering, it is possible to overlap quite a bit and realize savings in elapsed time.

The DCB parameters specified for any kind of data set may affect the amount of time charged to the execution of the program. The buffers are used as an intermediate storage location between the system I/O functions and the user's program. The user must allow space for the buffers in his region. FORTRAN does no overlapped I/O and therefore uses only one buffer, of the two default, at a time. BUFNO=1 coded in the DCB subparameters will save the number of bytes used by the second buffer. All data sets should be blocked if possible by including the letter B in the RECFM field and adding the LRECL subparameter. LRECL specifies the length of the logical record for fixed length records (F in the RECFM field) or the longest possible logical record for variable length records (V in the RECFM field). The BLKSIZE specifies the amount of space to be allocated for buffers and the size of the physical records. It is the number of physical records transferred which determines the I/O charges. The larger the blocking factor, BLKSIZE:LRECL, the less I/O time charged, the more region used. The largest block size for 2314 disks is 7294 and 32767 for tape. The BLKSIZE choosen should be a compromise between the frequency of I/O requests to the data set and the amount of region required. When an I/O request is made, the data management routines check to determine if the logical record is available in the buffer. If it is not, a physical I/O operation for a physical block is made. All subsequent logical I/O requests can be filled from memory until that buffer is exhausted. The larger buffer requires less physical I/O time.

14.0 FLOATING POINT ARITHMETIC

The relationship between the structure of floating point arithmetic and the way a programmer codes an algorithm is at best clouded. There are two main reasons why this is so:

1. Even with the "most sensible" of definitions for floating point arithmetic operations, the usual laws of real arithmetic fail to hold in many cases.

2. Most floating point architectures found in real computers do not conform completely to the "most sensible" definitions.

We shall concentrate our discussion on point one since the situation is bad enough in this case. We shall content ourselves with one example of how point two causes problems.

Let us begin with some terminology. We shall assume that our computer words are composed of 32 bits; these bits are numbered zero through 31. The usual representation of a floating point number is as follows:

$$\begin{array}{cccc} \pm & \leftarrow e \rightarrow & \leftarrow f \rightarrow & \\ \overline{0} & \overline{1} \ldots . \overline{7} & \overline{8} \ldots \overline{31} & \end{array}$$

i.e, the zero bit contains the sign; bits one through seven contain a non-negative binary represented integer e called the exponent such that $0 \leqslant e \leqslant 127$; bits eight through 31 contain a non-negative binary represented integer f called the fraction such that $0 \leqslant f \leqslant 2^{25}-1$.

Such a computer word represents the real number whose magnitude is

$$16^{e-64}\left(\frac{f}{16^6}\right)$$

Note that as the definition stands, the representation of a given real number is not unique, e.g.

(i) $0 = 16^{e-64}\left(\frac{0}{16^6}\right)$

for any permissible value of e

(ii) $16^{e-64}\left(\frac{f}{16^6}\right) = 16^{(e-1)-64}\left(\frac{16f}{16^6}\right)$

for any e, e-1, f, 16f in their respective permissible ranges.

This problem is eliminated by the stipulation that the representation be normalized, i.e.,

(i) 0 is represented by e = f = 0 (the sign convention varies on different machines)

(ii) $16^{15} \leqslant f < 16^6$ or $\frac{1}{16} \leqslant \frac{f}{16^6} < 1$

This means that the hexadecimal digit formed by bits eight through 11 is non-zero for a non-zero number.

Note that while hexadecimal and binary arithmetic seem to be equivalent, once normalization enters the picture this is definitely not so; binary normalization demands that bit eight = 1, i.e.,

$$\frac{1}{2} \leqslant \left(\frac{f}{16^6}\right) < 1.$$

The ramifications of this difference in normalization will not be pursued since it is not applicable to our discussion.

Floating point numbers shall either be represented as six significant hexadecimal digits or as an exponent-fraction pair (e,f). The floating point sum of x and y will be denoted by x plus y to distinguish it from x + y, the real arithmetic sum. Similarly for x - y we write x minus y, for x * y write x mul y, and finally for x / y write x div y.

In order to introduce our "most sensible" definitions of arithmetic operators, we need one more definition:

Given any non-negative real number x (a portion of any floating point number) we define rd(x) as follows:

If $16^{e-1} \leq x < 16^e$,

$rd(x) = 16^{e-6} *$ greatest integer less than or equal to $(16^{6-e}\ x + 1/2)$

$rd(0) = 0$

If $x < 0$, $rd(+x) = -\ rd(-x)$

What this amounts to is that rd(x) is "x rounded to six hexadecimal digits".

With these definitions we can now define:

For floating point numbers x, y:

(1)
$$\begin{aligned} x \text{ plus } y &= rd(x+y) \\ x \text{ minus } y &= rd(x-y) \\ x \text{ mul } y &= rd(x*y) \\ x \text{ div } y &= rd(x/y) \end{aligned}$$

whenever the appropriate real arithmetic operation leads to a number "roundable" to a floating point number.

There are two ways this condition can be violated. Let z be the result of real arithmetic operation. Overflow occurs if $|z| > 16^{63}(1-16^{-6})$ and underflow occurs when $|z| \neq 0$ but $|z| < 16^{-65}$. The actual result of such operations on a given machine will depend on the hardware and the setting of specified "masking bits" in a certain location. For our purposes, such results are undefined. Henceforth we assume that no operations lead to overflow or underflow unless specifically mentioned.

Besides the algebraic "closure" property which we have just seen does not hold, real arithmetic assumes five basic laws:

FLOATING POINT ARITHMETIC

associativity	$x + (y + z) = (x + y) + z$ $x * (y * z) = (x * y) * z$
commutativity	$x + y = y + x$ $x * y = y * x$
distributivity	$x * (y + z) = x * y + x * z$
existence of additive inverse	for each x, there is a unique -x such that $x + (-x) = 0$
existence of multiplicative inverse	for each $x \neq 0$ there is a unique $\frac{1}{x}$ such that $x * \frac{1}{x} = 1$

The law of existence of additive inverse implies the only solution to $x + y = x$ is $y = 0$. The law of existence of multiplicative inverse implies the only solution to $x * y = x$ $(x \neq 0)$ is $y = 1$.

Let us examine each of these laws for floating point arithmetic.

We can dispense with commutativity quickly since it is the only one of the five laws to hold, viz,

$$x \text{ plus } y = rd(x + y) = rd(y + x) = y \text{ plus } x$$

$$x \text{ mul } y = rd(x * y) = rd(y * x) = y \text{ mul } x$$

Unfortunately the discussion of the other four laws will be centered on showing that they do not hold for floating point arithmetic.

Let us begin with associativity:

(i) (111113. plus -111111.) plus 7.51111=2. plus 7.51111 = 9.51111
but 111113. plus (-111111. plus 7.51111) = 111113. plus
rd(-111109.AEEEF) = 111113. plus (-11110A) = 9.

(ii) (4.00001 mul 1.70001) mul 9.0000A = rd(5.C00570001) mul 9.0000A
= 5.C0005 mul 9.0000A = rd(33.C006680032) = 33.C006 but 4.00001 mul
(1.70001 mul 9.0000A) = 4.00001 mul rd(C.F00176000A) = 4.00001 mul
C.F0017 = rd(33.C0068F0017) = 33.C007

It is possible to concoct examples where overflow or underflow results from one sequence of operations but not from the other.

It should also be noted that in a sense associativity of addition "fails more egregiously" than associativity of multiplication, i.e., it happens more often and the relative discrepancy between answers is larger.

Distributivity perhaps fails worst of all and the next example shows that relying on the distributivity law can lead to disasterous consequences:

(iii) 200000. mul (F.00001 plus -F.) = 200000. mul .00001 = 2.
but (200000. mul F.00001) plus (200000. mul -F.) = rd(1E00002.) plus -1E00000. = 0.

This example also shows that floating point arithmetic is not an integral domain, i.e. it is possible for u mul v = u mul w but $u \neq 0$, $v \neq w$.

Next let us consider counterexamples to the additive and multiplicative inverse laws:

(iv) If (e_1,f_1) and (e_2,f_2) are such that $e_1 \geqslant e_2 + 8$ then (e_1,f_1) plus $(e_2,f_2) = (e_1,f_1)$. (On some machines $e_1 \geqslant e_2 + 7$ is enough to make this happen.)

Similarly,

(v) .100001 mul .FFFFFF = rd(.100000EFFFFF) = .100001

The lack of regularity exhibited in the previous five examples can surface in many subtle ways in particular programs. We present two examples where verification is left to the reader.

(vi) In real arithmitic $(x + y)^2 \leqslant 2(x^2 + y^2)$. (This formula is the basis for the fact that variances are always non-negative.) In floating point arithmetic this need not be true.

(vii) In real arithmetic, for each $x \leqslant y, x \leqslant \frac{x+y}{2} \leqslant y$ (i.e., geometrically the midpoint of an interval lies between the end points.) Again, in floating point arithmetic this need not be true.

Lest the reader think that there are no positive results concerning floating point operates, we present a theorem which provides limited information about floating point addition.

Let x, y be floating point numbers.

Let x' = (x plus y) minus y

Let y'' = (x plus y) minus x'

(Note that x',y'' are both able to be calculated effectively.) then:

$$(x + y) - (x \text{ plus } y) = (x \text{ minus } x') \text{ plus } (y \text{ minus } y'').$$

Of course if associativity held, then x' would be x and y'' would be y and the theorem would say that x + y = x plus y.

While the previous theorem gives a scheme for discerning the difference between real and floating addition, it is too cumbersome to apply in large scale programs. The interested reader is referred to [1] or [2] for further reading on the subject. [1] is written from the point of view that floating arithmetic is merely "inexact" real arithmetic. [2] represents the point of view that floating arithmetic is an exact branch of mathematics, albeit, with fewer helpful properties than real arithmetic. [2] also contains an extensive bibliography.

It has been mentioned that troublesome as the definitions given in (1) are, most machines do not even completely conform to them. We limit ourselves to an example involving addition:

Suppose the hardware of the floating point adder on a given machine operates as follows. (Again we neglect overflow and underflow.)

1. The fraction adder keeps seven hexadecimal digits.
2. The fraction of the number with lesser exponent is right shifted until exponents match.
3. Fractions are algebraically added.
4. Resulting fraction is left shifted if necessary to normalize it.
5. Fraction is rounded to six hexadecimal digits.

Let us see an example where such an adder will not get x plus y for the sum of x and y.

Let x = (54., - .800001)
y = (5B., .100000)

Then x plus y = (5A., .FFFFFF), but if one follows the adder rules just espoused, the eighth digit of x will be shifted out before the fraction normalization takes place. Hence the adder will get (5B., .1) as the sum. Although the relative error is small, the absolute error is 16^{14}.

Thus in summary, the programmer must be aware that under the best circumstances he must be wary of interchanging floating point algorithms that are algebrically equivalent. While the subject of floating point arithmetic is finally being treated in a positive rather than negative fashion, there is still little of a quantitative nature to guide him. For the time being, analysis of floating point arithmetic is more of an art than a science.

REFERENCES

1. R. Hamming, Numerical Methods for Scientists and Engineers — Chapter 2. McGraw Hill Book Company Inc., 1962.

2. D. Knuth, The Art of Computer Programming — Vol. II, Chapter 4, Section 4.1 - 4.3 Addison Wesly Publishing Company, 1969.

15.0 FUNCTIONS AND APPROXIMATIONS

The use of the FORTRAN library of transcendental functions is very convenient but is alsc very costly. The simple statement

```
Y = EXP(X)
```

invokes a function subprogram with over 30 statements. While this is immaterial in situations where the total execution time is small, in large scale programs it can become an unnecessarily large expense. Let us examine three ways to circumvent the use and hence the cost of the FORTRAN library functions.

1. Common subexpression elimination either by the compiler or by the programmer is of paramount importance when transcendental functions are involved. (See Comm.on Expression Elimination.) For example the pair of statements:

```
Z1 = EXP(X) * COS(Y)
Z2 = EXP(X) * SIN(Y)
```

should be written as:

```
TEMP = EXP(X)
Z1 = TEMP * COS(Y)
Z2 = TEMP * SIN(Y)
```

whether it is done implicitly by the compiler or explicitly by the programmer.

2. The use of algebraic identities that exist among certain classes of functions can lead to considerable savings of execution time. Indeed, in some cases, it can also lead to increased accuracy, since formal manipulation before evaluation is roundoff error free. Let us look at an easy example.

Suppose $\cos^2 x - \sin^2 x$ is to be calculated. As it stands, this expression involves two trigonometric evaluations, two multiplications and one subtraction. Of course, using the well known identity:

$$\sin^2 x + \cos^2 x = 1,$$

the expression can be reduced to the form:

$$1 - 2\sin^2 x.$$

which can be evaluated with one trigonometric evaluation one addition and one subtraction. This is not only faster but more accurate.

Certainly the preceeding example is a straw man that we set up so that we could knock down. However, the number of transcendental function identities is large, and vigorous effort should be made to use these identities to optimize code whenever possible. Note that the word "optimize" was used, not "speed up". Often one is faced with a tradeoff of accuracy for speed. Sometimes it is not clear exactly what kind of tradeoff is involved. A general rule of thumb is that if a substitution of an identity leads to more floating point algebra, a loss of accuracy can be expected. We complete our discussion of identities with an example that illustrates why it is impossible to make hard and fast rules.

Suppose one wishes to calculate cos(.001n) and sin(.001n) for n = 1,4000. A straight forward way to code this is:

```
(i)        T = 0.0
           DO 9000 I = 1,4000
           T = T + .001
           X(I) = COS(T)
      9000 Y(I) = SIN(T)
```

If one recalls that:

$$\cos((n+1)x) = \cos(nx) * \cos(x) - \sin(nx) * \sin(x) \text{ and}$$
$$\sin((n+1)x) = \cos(nx) * \sin(x) + \sin(nx) * \cos(x)$$

it is not hard to see that the following code is equivalent algebraically to the previous code:

```
(ii)       XA(1) = COS(.001)
           YA(1) = SIN (.001)
           DO 9000 I = 1,3999
           XA(I + 1) = XA(I) * XA(1) - YA(I) * YA(1)
      9000 YA(I + 1) = XA(I) * YA(1) + YA(I) * XA(1)
```

See also note 1 at the end of this section. Before looking at the results, one would ordinarily comment that code (ii) should be must faster, but less accurate than code (i). Now let us look at some of the actual results of these codes in the

following table. The column "Actual Value" was derived from "Eight Place Tables of Trigonometric Functions" by Pitey, published by Edwards Bros. Inc., 1939. Columns E_1 and E_2 represent the absolute error $*10^4$ for codes (i) and (ii) respectively.

All values are rounded to seven decimal digits.

	Actual Value	Code (i)	Code (ii)	E_1	E_2
cos(.01)	.9999500	.9999500	.9999500	0	0
cos(0.1)	.9950041	.9950043	.9950039	.002	.002
cos(0.3)	.9553364	.9553375	.9553351	.011	.013
cos(0.5)	.8775826	.8775853	.8775766	.027	.060
cos(0.7)	.7648418	.7648476	.7648360	.058	.058
cos(1.0)	.5403023	.5403126	.5402973	.103	.050
cos(2.0)	-.4161468	-.4156359	-.4160949	5.109	.519
cos(3.0)	-.9899925	-.9898351	-.9898616	1.574	1.309
cos(4.0)	-.6536437	-.6548992	-.6535568	12.555	.869
sin(.01)	.0099960	.0099998	.0099998	.038	.038
sin(0.1)	.0998324	.0998327	.0998313	.003	.011
sin(0.3)	.2955204	.2955171	.2955070	.033	.134
sin(0.5)	.4794255	.4794204	.4794015	.051	.240
sin(0.7)	.6442182	.6442113	.6441829	.069	.353
sin(1.0)	.8414710	.8414643	.8414217	.067	.493
sin(2.0)	.9092975	.9095311	.9092358	2.336	.617
sin(3.0)	.1411201	.1422198	.1411141	10.997	.057
sin(4.0)	-.7568024	-.7557162	-.7566550	10.862	1.474

Figure 22 - Accumulated Error in Repeated Function Evaluation

As per our original comment, code (ii) executes approximately 25 times faster than code (i), on the 360/95. If one examines the accuracy of the two methods for arguments ≤ 0.7, it is true that code (i) is more accurate than code (ii). For arguments in this range, one would have to decide whether the loss of accuracy is fatal. However, for larger arguments a strange phenomenon occurs, namely code (ii) has an absolute error that grows much more slowly than code (i). Thus for arguments between one and four code (ii) is 25 times faster and more accurate.

The reason for this seemingly anomolous behavior is as follows: The number .001 can certainly be expressed exactly with a one decimal digit fractic .. However, in the hexadecimal number system this number cannot be expressed exactly, no matter what finite number of hexadecimal digits one has. For $.001_{10} = .004189374BC6A7\ldots_{16}$. Thus if a floating point variable T is "set

equal to" $.001_{10}$, it will actually equal $.00418937_{16} = .000999999_{10}$. This seemingly insignificant difference is the reason code (i) gets much worse for larger values of the argument. Indeed adding together 4000 .001's, the exact FORTRAN answer is 3.99833965. The discrepancy is now larger than the step size! A possible solution to this problem would be to recall the sin and cos function after 500 evaluations to maintain accuracy.

Thus each individual problem must be treated with great care. The trigonometric identities provide the programmer with options. The ability to chose among the options is an art, an art the programmer must cultivate.

We present a list of the most common elementary identities for sin, cos, exp and in functions in appendix 1. Those interested in an expanded coverage of such information are invited to see [1]. Those who have occasion to use identities that exist among the so called "special function" of mathematical physics such as Bessel, Legendre, hypergeometric etc. should consult [2], [4] and [6]. In [4] a bibliography of other sources can be found. [2] contains a concise derivation of many of these advanced identities.

3. The final method for lowering the cost of calculating transcendental functions is the method of approximations. Here we touch upon one of the most far reaching branches of mathematics. From Linear Interpolation to Functional Analysis, Approximation theory encompasses a huge field. Thus we will limit ourselves to a few useful formulas and general remarks about other approximation methods.

The most common method of approximation for transcendental functions is the method of Taylor polynomials. The rationale for this is based on Taylor's theorem which in its entirety can be found in any good elementary differential calculus text. We limit ourselves to the Taylor polynomials for $\sin x$, $\cos x$, e^x, $\ell n(1 + x)$ $\sqrt{1 + x}$, about $x_0 = 0$, on an interval of radius $A < 1$, i.e. on the interval $(-A,A)$.

$$\text{(i)} \quad \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + E_1$$

where:

$$|E_1| \leq \frac{A^{2n+2}}{(2n+2)!}$$

(ii) $$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + E_2$$

where:

$$|E_2| \leqslant \frac{A^{2n+1}}{(2n+1)!}$$

(iii) $$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + E_3$$

where:

$$|E_3| \leqslant \frac{e^A A^{n+1}}{(n+1)!}$$

(iv) $$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots + (-1)^{n+1} \frac{x^n}{n} + E_4$$

where:

$$|E_4| \leqslant \frac{A^{n+1}}{n+1}$$

(v) $$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{4} + \frac{3x^3}{8} - \ldots + (-1)^{n+1} \frac{3*5*7\ldots*(2n-3)x^n}{2^n * n!} + E_5$$

where:

$$|E_5| \leq \frac{3*5*7*\ldots*(2n-1)\,A^{n+1}}{2^{n+1}\,(n+1)!}$$

Several remarks are in order concerning these formulas.

First, although formulas (i), (ii) and (iii) hold for any $A > 0$, they are not much use for $A > 1$ since the error term does not converge to 0 as n gets large.

Second, if one has an a priori error bound that must be satisfied, the size of n can be determined to make E smaller than the error bound, e.g.

To approximate cos x on (-1,1) to within 10^{-4}, one chooses n such that

$$\frac{1}{(2n+1)!} < 10^{-4} \quad \text{i.e. } n \geq 3$$

To approximate cos x on (-½, ½) to within 10^{-4}, one chooses n such that

$$\frac{(½)^{2n+1}}{(2n+1)!} < 10^{-4} \quad \text{i.e. } n \geq 2$$

The same techniques may be used with the other formulas.

Third, if one is interested in an interval about some other point besides $x_0 = 0$, one can rederive the Taylor polynomials about the new x_0. (See a good calculus text). If values are needed on an interval of radius greater than one, it is often possible to use identities to reduce oneself to the case of unit radius, e.g., to approximate e^x on the interval (-2,2), note:

$$e^x = e^{2*x/2} = (e^{x/2})^2$$

hence

$$\left(1 + \frac{x}{2} + \frac{\left(\frac{x}{2}\right)^2}{2!} + \ldots + \frac{\left(\frac{x}{2}\right)^n}{n!}\right)^2$$

approximates e^x. As usual when one uses identities, one must be careful about accuracy. (See discussion of identities in the first part of this section).

Finally, when coding polynomial approximations, one should optimize the algorithm for calculating the polynomial. (See the section on polynomial evaluation).

There are other polynomial approximation schemes besides Taylor polynomials. In particular, the use of Chebyschev polynomials is recommended in certain instances. For such considerations, the reader is referred to [3].

Non-polynomial approximations, such as rational functions, continued fractions, Fourier series, etc., are beyond the scope of this document. The interested reader is referred to [5] which is a handy introduction to this subject and a useful bibliography.

One final word of caution is in order; approximations are just that, approximations. Indeed, the FORTRAN library itself consists of approximations, albeit, of a very sophisticated form. The user should be wary of properties of approximations he may not desire. For example, in general polynomials of degree n have (n-1) relative maxima and minima, i.e., they oscillate. An error estimation may miss the fact a certain program is sensitive to such oscillations. In this case, high degree Taylor polynomials are worse than useless.

NOTE 1: Also if calculating $\sin(x + \delta)$ and $\cos(x + \delta)$ many times where δ is relatively small,

$$\sin(x_0 + \delta) = \sin x_0 \cos\delta + \cos x_0 \sin\delta \quad \text{and}$$

$$\cos(x_0 + \delta) = \cos x_0 \cos\delta - \sin x_0 \sin\delta$$

hence $\sin x_0 + \cos x$ need be called but once. Use Taylor series, to approximate to required degree for accuracy, for $\sin\delta$ and $\cos\delta$. The same may be done for

$$\exp(x_0 + \delta) = \exp x_0 \exp\delta$$

as well.

REFERENCES

1. L. Ayers — Trigonometry, Schaum's Outline Series.

2. R. Courant — Courant and Hilbert, Mathematical Physics, Vol. I, Interscience, 1953.

3. R. Hamming — Numerical Methods for Scientists and Engineers, Part II, McGraw Hill, 1962.

4. Magnus, Oberhettinger — Formulas and Theorems for the Functions of Mathematical Physics, Chelsea, 1949.

5. J. Rice — Approximations of Functions, Chapter 6, Addison Wesley, 1964.

6. M. Spiegal — Advanced Mathematics for Engineers and Scientists, Schaum's Outline Series.

16.0 EVALUATION OF POLYNOMIALS

At first glance many programmers may be surprised to find a section devoted to polynomials. After all, what could be simpler? Indeed, FORTRAN was designed to make polynomial evaluation an easy task to program, viz:

$$P(x) = a_0 x^4 + a_1 x^3 + a_2 x^2 + a_3 x + a_4$$

becomes:

```
P = A0*X**4+A1 *X**3+A2*X**2+A3*X+A4
```

However if execution time optimization (and accuracy) are crucial questions, then a more sophisticated approach is demanded. The study of efficient means of polynomial evaluation goes back to 200 B.C. hence predating the electronic computer. But with the advent of large scale problems on high speed computers this study has blossomed into a branch of mathematics in its own right.

We shall start with the problem of evaluating the general polynomial

$$P(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$$

for a "random" input value of x. We shall assume that the coefficients of P are coded as follows:

$$C = a_0, \; A(I) = a_i, \; i = 1,n.$$

We shall assume that F(I) is the floating point variable whose value is I, I = 1,n.

Let us start with the most naive and perhaps worst method of evaluating P. Consider:

```
      NM1 = N - 1
      P = C * X ** F(N)
      DO 9000 I = 1, NM1
9000  P = P + A(N - I) * X ** F(I)
      P = P + A(N)
```

which evaluates P correctly, but is highly wasteful of execution time. Most programmers will quickly see that the following code is a substantial improvement:

```
      NM1 = N - 1
      P = C * X ** N
      DO 9000 I = 1, NM1
9000  P = P + A(N - I) * X ** I
      P = P + A(N)
```

The fact that a fixed point variable, rather than a floating point variable is used as an exponent, allows the use of a faster exponentiation routine.

However, the preceeding code is far from optimum. Indeed it is many times slower than necessary. Consider:

```
      P = C
      DO 9000 I = 1,N
9000  P = P * X + A(I)
```

The reader may show that the results of this code are (algebraically) the same as the previous code. First, there is no explicit exponentiation. Second, there are only as many multiplications and additions as in the previous code. Finally, the indices of A are simpler. This simpler arthmetic is manifested in (usually) enormous execution time savings.

This method of evaluating P(x) as:

$$P(x) = ((\ldots ((a_0 x + a_1)x + a_2) \ldots)x + a_n$$

is called Horner's method (although it was known to Newton). Although Horner's method represents a tremendous improvement over naive evaluations, when special circumstances hold, it can be improved still further. We present three examples of special techniques.

1. On a machine whose architecture allows pipelining or parallelization of arithmetic operations, it is possible to devise higher order Horner methods to make full use of this capability. Let us suppose that P(x) is of even degree $n = 2m$. (The reader may supply details for $n = 2m + 1$.) If we let $y = x^2$, we can write P(x) as:

$$P(x) = (a_0 y^m + a_2 y^{m-1} + \ldots + a_{n-2} y + a_n)$$

$$+ (a_1 y^{m-1} + a_3 y^{m-2} + \ldots + a_{n-1}) x$$

$$= ((\ldots ((a_0 y + a_2) y + a_4) \ldots + \ldots) y + a_n)$$

$$+ ((\ldots ((a_1 y + a_3) y + a_5) \ldots) y + a_{n-1}) x$$

This translates to FORTRAN as:

```
      P1 = A(1)
      P2 = C
      Y = X * X
      NM3 = N - 3
      DO 9000 I = 1, NM3, 2
      P1 = P1 * Y + A(I + 2)
9000  P2 = P2 * Y + A(I + 1)
      P = P1 * X + P2 * Y + A(N)
```

The separation of calculations of the even (P2) and odd (P1) terms allows a machine, such as the 90 series 360's and 370's, to make better use of the reservation stations in the hardware.

Actual test cases of calculating a polynomial of degree 10 by the naive code (with integer exponents), by Horner's method and by the just mentioned Horner's method of second order revealed that the naive code is approximately 25 times slower than either Horner's method, and that Horner's method of second order is at least several percent faster than the original Horner's method, on the 360/95.

2. There are several occasions where one wishes to calculate several polynomials which are related in such a way that intermediate information can be "shared" by more than one polynomial. For instance suppose one wishes to calculate P(x) and its derivative.

$$P'(x) = n a_0 x^{n-1} + \ldots + 2a_{n-2} + a_{n-1}$$

The following code calculates P and DP (=P') without the explicit calculation of $\{k a_{n-k}\}$.

```
      P = C
      DP = 0.
      DO 9000 I = 1, N
      DP = DP * X + P
9000  P = P * X + A(I)
```

3. In many problems of numerical approximations one wishes to calculate P(x) on a sub-set of an arithmetic progression, i.e. on equally spaced data $\{x_0, x_0 + h, x_0 + 2h, \ldots\}$. If this is to be done for a large number of values compared to the degree of the polynomial, it is worthwhile to set up difference tables, for one is then able to calculate values of P using only n additions and no multiplications per value after a transient phase.

This technique is based on the following fundamental theorem.

Let P be a polynomial of degree n, $(\Delta P)(x) = P(x + h) - P(x)$, $(\Delta^{k+1} P)(x) = \Delta(\Delta^k P)(x)$ $k = 1, n - 1$. Then $(\Delta^n P)(x) = a_0 (n!) h^n$ for all values of $x = x_0, x_0 + h, x_0 + 2h, \ldots$.

We now define the difference table of P. (This construction is applicable to all functions, not just polynomials. But for a non-polynomial function, no column is ever constant.)

x	P	ΔP	$\Delta^2 P$	etc
x_0	y_0	$y_1 - y_0$	$(y_2 - y_1) - (y_1 - y_0)$	etc
$x_0 + h$	y_1	$y_2 - y_1$	$(y_3 - y_2) - (y_2 - y_1)$	etc
$x_0 + 2h$	y_2	$y_3 - y_2$	$(y_4 - y_3) - (y_3 - y_1)$	etc
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

The basic feature of this table for our purposes is that the sum of any two consecutive horizontal values yields the value under the leftmost of the original two values. Thus if one can calculate the top row and the rightmost column, one can generate the whole table by successive additions. In particular, the leftmost column yields the function values derived. (In many applications, the high order differences also play an important role. From our point of view, they are an unexpected bonus.)

One example is worth a thousand words in getting the "hang" of using the table.

Let $P(x) = x^3 + 1$, $x_0 = 0$, $h = .1$, (The simplicity of the polynomial in no way affects the validity or complexity of the table construction).

$$P(0) = 1.000,\ P(.1) = 1.001,\ P(.2) = 1.008,\ P(3) = 1.027$$

Let us start the table:

x	P	Δ P	Δ^2P	Δ^3P
0.0	1.000	.001	.006	.006
0.1	1.001	.007	.012	
0.2	1.008	.019		
0.3	1.027			

Note that $.006 = 1 * 3! * (.1)^3$ and that the theorem previously quoted guarantees that the entries in the Δ^3P column are .006.

Let us fill out the table

x	P	Δ P	Δ^2P	Δ^3P
0.0	1.000	.001	.006	.006
0.1	1.001	.007	.012	.006
0.2	1.008	.019	A	.006
0.3	1.027	B	C	.006
0.4	D	E	F	.006
0.5	G	H	I	.006

as follows:

$$A = .012 + .006 = .018$$
$$B = .019 + A = .037$$
$$C = A + .006 = .024$$
$$F = C + .006 = .030$$
$$E = B + C = .061$$
$$P(.4) = D = 1.027 + B = 1.064$$
$$I = F + .006 = 1.094$$
$$H = E + F = .091$$
$$P(.5) = G = D + E = 1.125$$

The table may now be continued downward as far as one pleases. Each row depends only on three additions. The lack of multiplication makes for a marked speed improvement over Horner's method.

Let us point out that we have barely scratched the surface of the subject of polynomial evaluation. In particular we have not mentioned the technique of coefficient "adaptation." Further references on this subject and the difference calculus in general can be found in [1], [2] and [3].

Finally, note that there has been no discussion of accuracy. This is a thorny problem entwined with the general problem of floating point arithmetic. We make a few general comments.

(i) Horner's methods are at least as accurate as naive evaluation.

(ii) In evaluating P(x) and P'(x) simultaneously, implicitly ka_{n-k} is evaluated as

$$\underbrace{a_{n-k} + a_{n-k} + \ldots + a_{n-k}}_{k \text{ terms}}$$

These two floating point operations need not give the same answer.

(iii) In constructive difference tables, the calculation of $\Delta^n p = a_0 (n!) h^n$ should be done with extreme accuracy, since errors in this quantity propagate in meaningful form back to the polynomial values (see [1] p 27).

REFERENCES

1. R. Hamming — Numerical Methods for Scientists and Engineers, McGraw Hill, 1962.

2. D. Knuth — The Art of Computer Programming, Chapters 4 - 6, Addison Wesley, 1969.

3. L. Milne — Thomson Calculus of Finite Differences, McMillan, 1933.

17.0 OVERLAY PROGRAMS

17.1 CONSIDERATIONS

This section will cover minimization of the region size needed to execute a given algorithm on a 360 computer. Covered topics will include overlay construction, specific linkage editor techniques, overlay aids such as OVLY, LOADMAP and reusability and reentrancy considerations which the FORTRAN programmer must consider when coding his routine to make overlaying practical. Since reading a dump from an overlayed load module is just a bit more difficult some hints on dump reading are included.

17.2 STRUCTURE

OS supports a reusable multiple tree overlay system where each tree consists of one node (called the root segment) and the remaining nodes (segments) partitioned into sets of nodes each of which appears as a tree. This support can be visualized with a simple example common to most programs. Consider a typical program with three main service areas:

1. Input parameter and data verification.
2. Main processing.
3. Output formatting and printing.

These would typically be organized in memory as a simple block (a one node tree) such as:

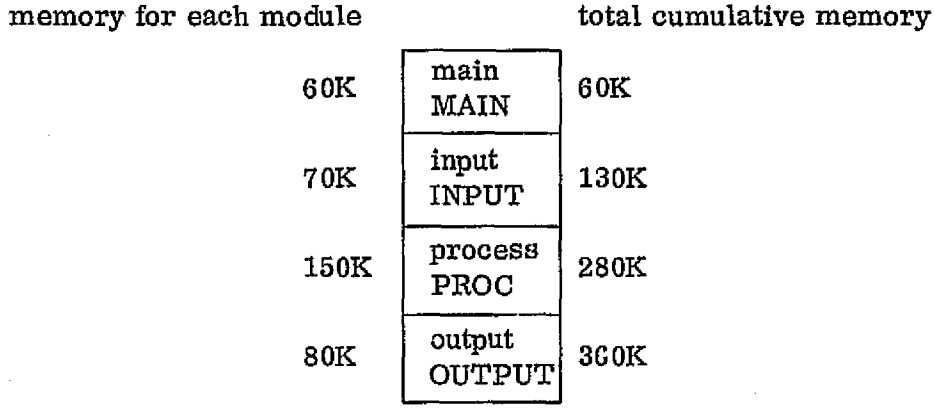


Figure 23 - A Simple Program

This program can be logically viewed as a driver (root segment) and three additional processing segments (input, processing, and output) only one of which needs to be in memory at any given time.

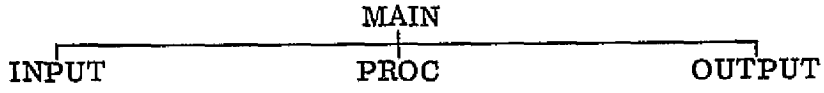


Figure 24 - A Simple Program Tree

Now, if core requirements are considered for this simple tree:

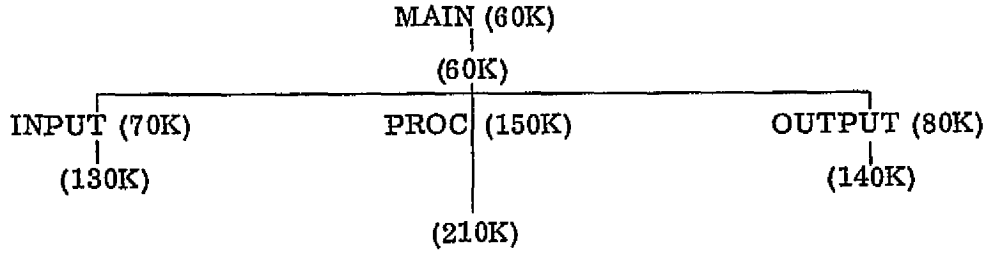


Figure 25 - A Simple Overlay

Two things should be apparent from figure 25. The program now occupies only 210K, a saving of 150K (or about six hours turnaround on a busy 360/95 day) and, more subtly, code added to INPUT or OUTPUT is "free" in that the program will get no longer until INPUT or OUTPUT exceeds PROC in its memory demands. It is possible to reduce the needed region further still by viewing the PROC routine as the root segment of another tree and logically partition the PROC algorithm into its component parts, BOUND (20K) and INVERT (30K).

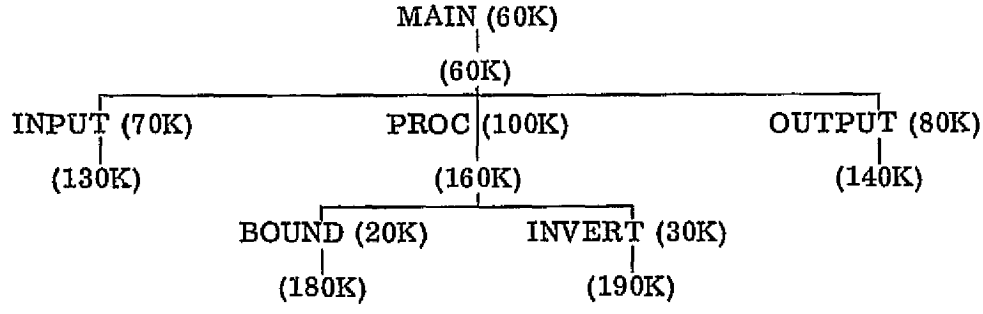


Figure 26 - A Near Minimum Size Overlay

The total core required for this structure is now 190K and BOUND can be added to the list of routines which can be "freely" expanded without increasing the size of the module.

17.3 CHANGES REQUIRED FOR OVERLAYING

Under ideal conditions where each branch of a program flow is executed once such as in Figure 25 an overlay program will take no more I/O time and only minimal more CPU time than the non-overlayed version of the same routine. The system will load each segment once as it is needed. When the FORTRAN statement CALL INPUT in MAIN is executed it automatically calls the segment containing the INPUT subroutine into memory. When INPUT completes its processing and returns to MAIN, its code remains unaltered in memory and can be recalled from MAIN without a further I/O charge. In the processing portion of the program PROC calls both BOUND and INVERT to perform the necessary calculations. Since BOUND and INVERT share the same memory they are never in memory simultaneously and cannot call each other at any time. Any attempt to form a call between routines located exclusively, such as BOUND and INVERT will result in a linkage editor diagnostic IEW0182. If this diagnostic messages is circumvented by coding the LET option the first call from BOUND to INVERT will operate properly but if INVERT attempts to return (in a FORTRAN sense, by the RETURN statement) to BOUND an 0Cx abend will occur in the part of INVERT where BOUND used to be located.

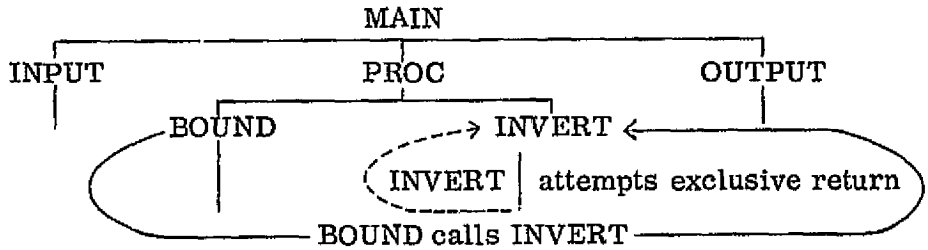


Figure 27 - An Invalid Exclusive Call with LET Specified

If, however, INVERT is an ALC routine which does not return to BOUND but branches instead to somewhere in PROC or MAIN it is legal to form the overlay structure as shown in Figure 27 and the message IEW0172 is issued. After verifying the program branching, the XCAL option in the PARM field to the linkage editor can be coded which will cause it to check the validity of the exclusive call and issue an IEW0161 warning message.

PROC can, however, call BOUND and INVERT (but not INPUT or OUTPUT since they are mutually exclusive) as often as required. Alternate calls between two (or more) exclusive routines will cause the called routine to be brought into memory replacing the existing routine. Each time one of the segments is

loaded into memory there is an I/O charge of 32 milliseconds for each approximately 6K of loaded code. In our example:

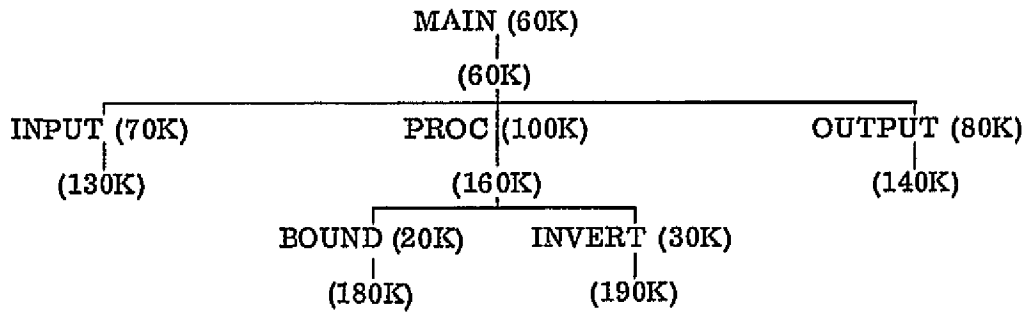


Figure 28 - Some Overlay Timings Estimates

A return from BOUND and a call from PROC to INVERT would bring five "blocks" into memory costing 160 milliseconds of I/O time and a RETURN from INVERT followed by a call from PROC to BOUND would bring in about four blocks costing 132 milliseconds of I/O time. A limited amount of calling back and forth clearly justifies the memory savings but is impractical for an iterative routine where convergence requires repeated execution of exclusive routines not loaded in memory simultaneously. That is:

```
      DO 100 I = 1, 50
      CALL BOUND
100   CALL INVERT
```

Figure 29 - A Simple Driver Program

The example in Figure 29 would require almost 30 seconds of I/O time for loading the necessary segments into memory. This can only be justified on the longest path of a long running program.

The above examples illustrate the essential steps in deciding how to construct an overlayed load module. First, and clearly the most important step, is to draw the logical flow of the module in tree form as shown in the above figures. Include in this drawing the sizes of the modules, taken from a prior link edit map, and determine the size of each segment by totaling the module sizes within that segment. Next to each name note the length of the module. Below the name show the length of the path from the root segment.

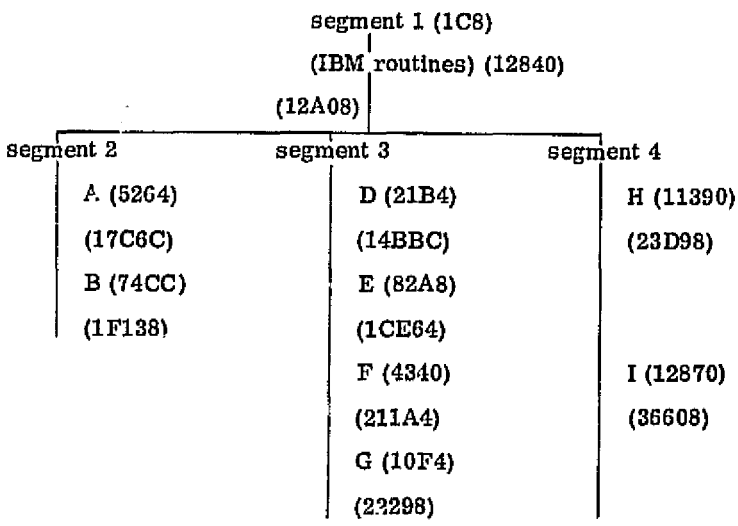


Figure 30 - Some Sample Overlay Segment Sizes

From Figure 30 it is clear that segment four is the longest since routines H and I are much longer than their exclusive cousins D, E, F and G. Resist the urge to overlay D, E, F and G any further, even though they, as small routines, are probably simpler than H and I. The goal should be to balance the tree as much as possible. In this example the module is about 230K as overlayed in Figure 30. If routines H and I can be overlayed into separate segments by converting segment four in segments four and five, the total module will drop to about 150K.

17.4 CODING CONSIDERATIONS FOR OVERLAYED PROGRAMS

Most fully debugged programs can be overlayed with no source code or logic alternations. Counters and initialization flags may have to be moved to a common block under some circumstances. For example:

```
      SUBROUTINE DEMO
      LOGICAL FIRST
      DATA FIRST/.TRUE./
      IF (FIRST) GO TO 100
C
C     COMPUTATION CODE
C
      GO TO 200
  100 CONTINUE
C
C     INITIALIZATION CODE
C
      FIRST = .FALSE.
  200 CONTINUE
```

Figure 31 - Common Block Initialization

The first time the routine in Figure 31 is executed that DATA statement will make FIRST true and the initialization code will be executed. Each future pass through subroutine DEMO will find FIRST false and only the computation code will be executed. If, however, the subroutine is in an overlay structure such that it is repeatedly loaded, the DATA statement will cause FIRST to always be TRUE with each new load of the routine.

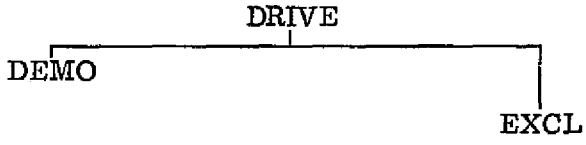


Figure 32 - Exclusive Segments

When the DRIVE routine calls DEMO it will automatically be loaded with FIRST .TRUE. and the initialization code will execute properly. If DRIVE then calls EXCL, a routine exclusive to DEMO, and recalls DEMO at a later time the overlay supervisor will automatically load EXCL and reload DRIVER as needed. The reloaded DEMO will again have FIRST set to TRUE from the DATA statement and the initialization code will be re-executed, probably erroneously. To preserve the reusability of DEMO, the DATA statement must be either removed and FIRST passed as an argument from DRIVER which has a DATA statement to initialize FIRST, or a common block and BLOCKDATA routine must be used to initialize FIRST. The common block must be sufficiently close to the root so as not to be overlayed during the "life" of FIRST. Another difficulty can be created with the use of counters in routines which may be overlayed and recalled. If a counter is initialized to zero in a DATA statement and incremented each time the routine is entered it must be in the argument list or a common block if it is to survive the overlay process when the routine is sharing address space with another exclusive segment and being reloaded. The FORTRAN compiler automatically provides reusability for such things as DO loop ranging variables and they cause no difficulty in overlaying routines.

The ALC programmer can, in general, overlay routines with the same considerations as above. He must be careful not to issue an OPEN without the corresponding CLOSE in a segment which is to be overlayed. If he does, any future I/O to the DCB in the overlayed segments will result in an abend with a very difficult dump to debug. Similarly, a GETMAIN should have a corresponding FREEMAIN issued or special care should be taken to preserve the addresses necessary to 'free' the storage later. With these considerations reentrant routines should present no usage difficulty.

17.5 THE MECHANISMS OF THE LINKAGE EDITOR

The linkage editor numbers the segments of code consisting of one or more routines (subprograms, common areas, or CSECTS) from top to bottom and left to right as shown in figure 33 below.

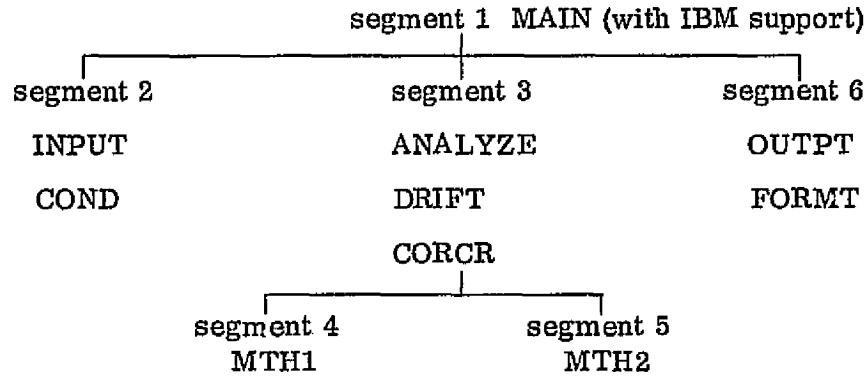


Figure 33 - Overlay Segments

The primary tool for describing the overlay structure is the INSERT card. It says to the linkage editor: "take the routine named from segment one (the root) and place it in the segment where I currently am". An OVERLAY card is used to indicate the beginning of a new segment and identical names on OVERLAY cards start at the same level in the tree as the first time the name appears. For figure 33 the required cards are:

```
OVERLAY   TOP
INSERT          INPUT
INSERT          COND
OVERLAY   TOP
INSERT          ANALYZ, DRIFT, CORCR
OVERLAY   MID
INSERT          MTH1
OVERLAY   MID
INSERT          MTH2
OVERLAY   TOP
INSERT          OUTPT
INSERT          FORMT
```

Figure 34 - Overlay Control Cards

The INSERT and OVERLAY cards must not begin in column one; more than one name can be placed on an INSERT card. The three OVERLAY TOP cards define the first major branch and the two OVERLAY MID cards define the next level of overlaying. There is no effective limit on the number of levels it is possible to have but storage is reserved for the longest leg found by the linkage editor and no advantage is gained by overlaying other legs which are not the longest.

17.6 OVERLAY TOOLS

17.6.1 OVLY program - to draw a tree

A program is available which will take an existing overlayed load module and produce a tree, such as the ones drawn in this section, and optionally print or punch the necessary control cards to reconstruct the tree. The program provides useful information when trying to optimize an existing overlay structure or debugging an overlay program where the suspected bug is in the overlay structure itself. As with all standard programs a procedure is available which operates on a catalogued library, or data set, to produce the desired picture and control cards. The documentation is contained in appendix G.

17.6.2 LOADMAP - to map a load module and list cross references

LOADMAP produces a listing of all the routines in a specified load module. A linkage editor map and two cross references listings which show all the routines a specific routine calls and all the routines that call a specific routine. Common area references are likewise cross referenced. This is useful to a program which is to be overlayed so that a tree may be drawn. The documentation for this program is in appendix F.

17.7 OPTIMIZATION OF AN EXISTING OVERLAY

The techniques described above will produce a substantial core saving in most programs which are not now overlayed. Through the use of the OVLY and LOADMAP programs some additional memory saving can usually be realized with only a slight increase in execution time. It is also possible to decrease the complexity of most large overlay structures with no increase in memory necessary for program execution. The two principal rules to remember are:

1. Combine short legs into one larger leg, being careful that it does not become the longest leg, and

2. Carefully search the longest leg for any routines which can be relocated to a shorter leg.

For example:

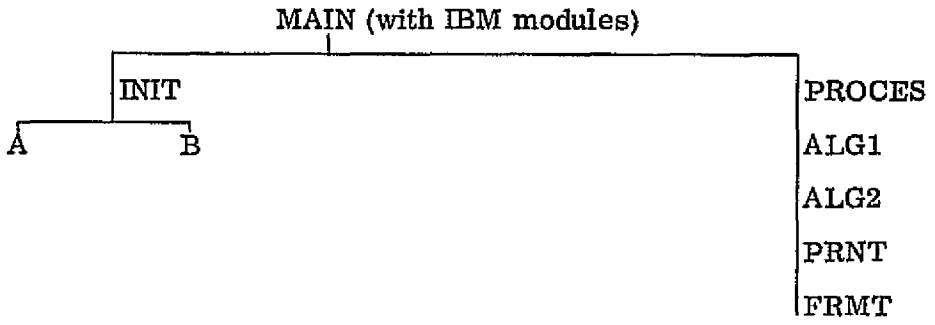


Figure 35 - A Candidate for Overlay Optimization

The routines A and B can be combined into the INIT segment and the segment containing PROCES will still be the longest in the program. On careful examination of the segment containing PROCES we see that the routines ALG1 and ALG2 are not used in the same run thus the overlay tree can be redrawn as:

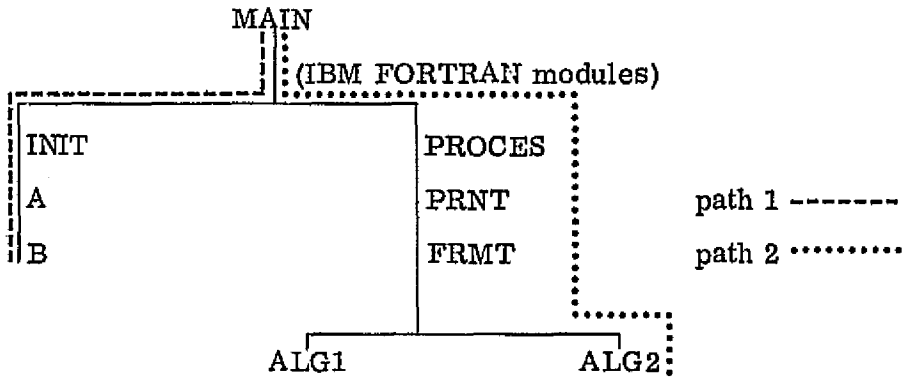


Figure 36 - A Balanced Overlay Tree

where the total length of path one is approximately equal to path two. No further overlay optimization is likely to occur unless a more advanced technique, outlined below, is employed.

17.8 MULTIPLE REGION OVERLAYS

Occasionally there is an opportunity for a further reduction in the required region that can be obtained from the processes outlined above. If two or more support routines are used by two or more major legs of the program, where one of these legs is the longest leg and the two routines do not call each other, they can be relocated from the root segment and placed parallel to each other in a second base area of the overlay structure called, confusingly, a region.

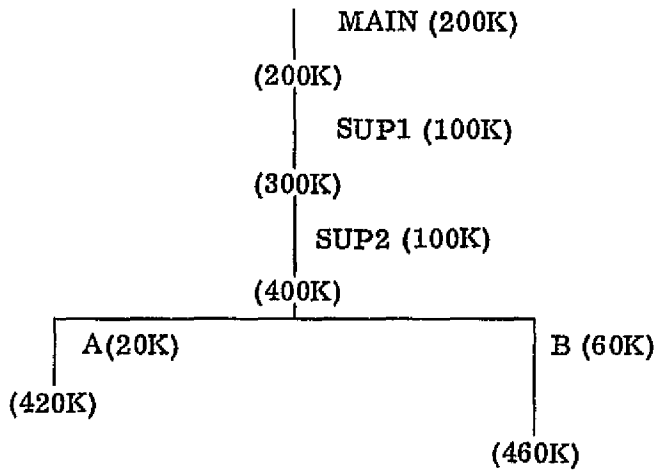


Figure 37 - A Candidate for Multiple Region Overlaying

The entire module in Figure 37 may take 460K as shown, a saving of 20K from the straight line linking with no overlay. Since SUP1 and SUP2 do not call each other but are called by A and B they normally reside in the root segment but can be relocated in a second region with the following control cards:

```
OVERLAY   ONE
INSERT    A
OVERLAY   ONE
INSERT    B
OVERLAY   SUPT (REGION)
INSERT    SUP1
OVERLAY   SUPT
INSERT    SUP2
```

Figure 38 - A Multi-region Control Card Deck

The control cards as shown in Figure 38 would generate a tree which would look like:

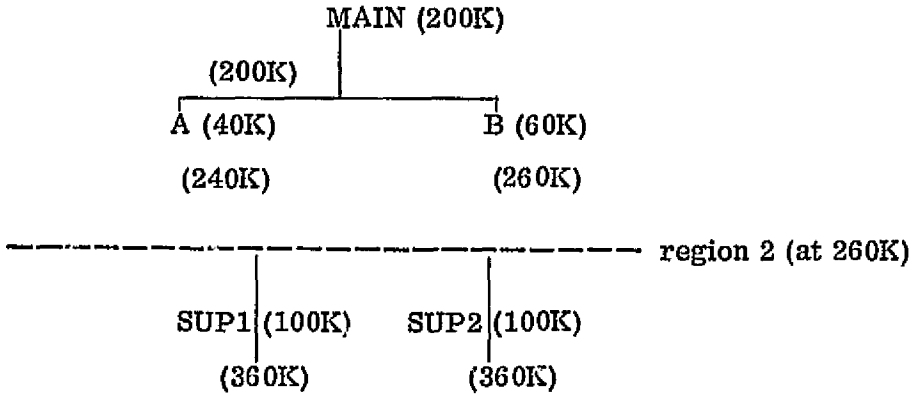


Figure 39 - Multi-region Overlay Tree

The above multi-region overlay gives a saving of 100K.

A total of four regions can be designated and each of the regions must be complete before the next region is begun with an OVERLAY name (REGION) control card. The IBM routine IHCERRM can be moved easily from the root segment to a second region provided confidence exists that there is no arithmetic error, such as an underflow, since it is possible for IHCERRM to be invoked for all FORTRAN routine errors.

17.9 BUGS, DUMPS, HAZARDS AND PIT FALLS

Do not overlay a routine having a FORTRAN DEFINE FILE statement until all processing for the associated unit is complete. Do not overlay a routine containing an ALC OPEN until the corresponding CLOSE is issued.

Be especially careful about DATA statements used to initialize counters, they will be reset each time the routine is called after being overlayed. Use BLOCK DATA and common areas to be sure. Insure the common areas are in the root segment, or high enough in the tree so they are not overlayed at the wrong time.

There is only one serious dump "caused" by an overlay structure.

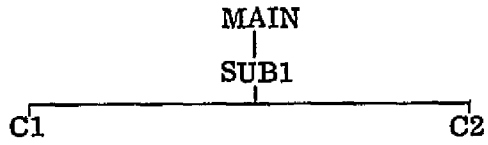


Figure 40 - A Structurally Caused Failure

In Figure 40 MAIN calls C1, C1 calls SUB1 which calls C2. The linkage editor will not detect the exclusive call and C1 will be overlayed by the code for C2. The resulting 0Cx traceback will show C2 being called by SUB1 having been called by a possible invalid reference while the forward trace will show MAIN calling C1. You can determine the segment in storage from the following table which is always at the beginning of your load module:

+8 - Last segment currently in region 1

+9 - Highest segment of region 1

+A - Last segment currently in region 2

+B - Highest segment of region 2

+C - Last segment currently in region 3

+D - Highest segment of region 3

+E - Last segment currently in region 4

+F - Highest segment of region 4

Figure 41 - Segment Table ($SEGTAB) Format

Normally you only need to examine byte nine of your load module and look up the segment number found (shown in hexadecimal) in the linkage editor map or LOADMAP link map. If the routine in the trace table is not in the segment shown or in its path to the root segment, you have made an illegal exclusive reference and must reexamine your overlay structure.

Further information and format information can be found in the IBM manuals LINKAGE EDITOR AND LOADER GC28-6538, and SYSTEM CONTROL BLOCKS GC28-6628.

18.0 The Different IBM FORTRAN Compilers

The following discussion relies heavily on Appendix H, "FORTRAN IV (H) Optimization Facilities" from the IBM manual "FORTRAN IV (G and H) Programmers's Guide", GC28-6817.

18.1 COMPARISONS

FORTRAN IV G, G1, and H with OPT=0 generate code with approximately the same level of sophistication. It is very straightforward and each type of FORTRAN statement creates a specific set of assembly instructions. The compilation times are the lowest and execution times the longest of the level of optimization available with FORTRAN H. These should not be used for other than syntax scanning, exceptionally quick or one shot programs. The G and G1 compilers are almost the same. G1 is usable from TSO and supports list directed I/O and the TEST option. G and H do not support these new features. FORTRAN H allows the following: (a) arithmetic operations with one byte variables and options for generating optimized code, (b) producing a structured source listing, (c) a cross reference list of variables and statement numbers, (d) controlling the amount of storage used when the compiler is attached, and (e) allowing the compiled source code to execute even if there were source errors. The H compiler does not support the DEBUG facility which is available with G and G1.

18.2 FORTRAN H OPT=1 OPTIMIZATION

When OPT=1 is specified, the compiler execution time increases slightly, but a large savings is evidenced in the execution of the compiled code. The improvements in the generated code are:

1. Placing often used variables in registers and retaining the value for later use.

2. The same is done for FORTRAN generated values (base registers for data areas, COMMON, or table addresses).

3. Use of branching instructions which utilize registers.

The code generated is still very similar to unoptimized code but makes better use of the registers and uses several faster instructions.

18.3 FORTRAN H OPT=2 OPTIMIZATION

OPT=2 requires more compile time but generates even better code than OPT=1. The following are done in addition:

4. All values are attempted to be held in registers (variables, constants, and FORTRAN generated values).

5. Recognition of redundant calculations and use of registers to hold values of intermediate results.

6. Moving code ahead of loops which is not changed within the body of the loop.

7. Removing calculations which are not used.

8. Generate the fastest possible branch and logical testing instructions.

OPT=2 generated code is more sophisticated than any other FORTRAN generated code and requires the programmer to be alert to possible errors which may be generated. These are discussed below.

18.4 COMPILE AND EXECUTION SPEED TIMINGS

A general purpose test program was compiled and executed utilizing the various compilers and optimization levels. FORTRAN G timed out after 14.25 CPU minutes and was excluded from the tests.

	CPU MINUTES OF EXECUTION	COMPILER TIME IN MINUTES CPU	I/O
FORTRAN G1	13.861	0.057	0.123
FORTRAN H OPT=0	14.123	0.011*	0.074
FORTRAN H OPT=1	8.502	0.011*	0.078
FORTRAN H OPT=2	2.303	0.085	0.180

* Includes a BLOCKDATA routine

Figure 42 - Compiler Comparison Timings

18.5 OPT=2 WARNINGS

OPT=2 causes analysis of the program structure. Blocks of code are analyzed as a unit for the most active values. A block starts with a labeled statement, or the first statement in the program unit, and ends with another labeled statement, a branch statement (including READ with END or ERR specified), or a CALL. Within the body of a block, registers can be fully utilized and intermediate results, partial calculation of expressions, and base addresses are generated once and reused from their high speed positions. Excessive numbers of branches or referenced statement labels will reduce the effectiveness of the optimization by reducing the scope of a block. Optimization is also reduced when a block starts with an IF statement, conditional GO TO, a READ statement with END or ERR, and a CONTINUE to end a loop where other than a DO loop follows and no values are initialized. These statements or combination of statements provide a second path which the compiler views as equally likely and must save or set up values again before leaving or entering the block at the implied path. (For example, a computed GO TO may fall through, READ with END or ERR may not fall through, or the start of a loop may be obscured when there are no values initialized after a CONTINUE.)

Errors or code which executes differently than intended may be generated with OPT=2. The following are things to watch for:

1. Code is moved from inside a loop to the initialization of the loop when all values in an expression or subexpression are not changed in the loop. This occasionally will not give the expected results.

For example:

```
      DO 11 I=1,10
      DO 12 J=1,10
      IF (B(I).LT.0) GO TO 11
12 C(J)=SQRT(B(I))
11 CONTINUE
```

The IF statement contains no expression relying on J, the index, or a value calculated in both loops and therefore is rearranged as though the following was written:

```
      DO 11 I=1,10
      TO1=SQRT(B(I))
      DO 12 J=1,10
      IF (B(I).LT.0) GO TO 11
12 C(J)=TO1
11 CONTINUE
```

It is now apparent that the computation of the square root of B(I) is always perfomed before B(I) is tested for a valid value. The compiler recognized that the computation of SQRT(B(I)) does not depend on the inner loop index, J. To preserve the intent, the code should be rearranged as shown below:

```
      DO 11 I=1,10
      IF (B(I).LT.0) GO TO 11
      DO 12 J=1,10
12 C(J)=SQRT(B(I))
11 CONTINUE
```

Other checks made to ensure the successful execution of statements following the one with the test may be moved to a useless place. Adding and subtracting a value in the loop will cause retention of the statement in its proper place (IF (B(I)+J-J.LT.0). This should only be done when a problem really exists.

2. Assigned GO TO statements with an incomplete statement number list may not compile properly. Be sure to have an accurate list of all possible branches.

3. When a user subprogram has the same name as a FORTRAN supplied subprogram, errors may occur if: 1) variables are remembered from one call of the subprogram to the next, 2) I/O is performed, 3) the subprogram saves into COMMON or its arguments. Avoid the problem by explicitly declaring the name of the subprogram in an EXTERNAL statement. The FORTRAN supplied subprogram may not be referenced in that program unit.

4. Since values are held in registers certain relationships may not be known outside the body of the physical loop and rarely after the completion loop. These are implied equivalences, indices from DO loops and implied loops, and FORTRAN generated temporary variables. An implied equivalence is illustrated by:

```
      COMMON/COMMON/A(10),B,C
      DIMENSION E(12)
      EQUIVALENCE (A,E)
         .
         .
         .
      DO  10 I=1,N
         .
         .
         .
      E(11)=D+G
      F=B+G
         .
         .
         .
   10 CONTINUE
```

The data in memory would be as follows:

A(1)	A(2)	A(3)	...	A(9)	A(10)	B	C
E(1)	E(2)	E(3)	...	E(9)	E(10)	E(11)	E(12)

E(11) and B occupy the same location as do C and E(12). In the example it is possible that B will not contain the just calculated value of E(11). The optimization is done by name, not by location. In general, variables in EQUIVALENCE statements are marked so they are not moved or partial results calculated using them. This may cause serious downgrading of optimization.

5. Call by value arguments, enclosed in slashes, may not be passed properly unless placed in COMMON.

19.0 COMPILER INTRINSIC FUNCTIONS

The FORTRAN IV H compiler provides a number of built-in pseudo-functions which are useful for logical operations and bit manipulation. The logical operations pseudo-functions are coded as regular functions but generate instructions in-line, which utilize assembler code to do the precise operation requested. These functions are extremely fast. To make the implementation of these functions as in-line code requires that the XL option be on (by specification on the PARM field or by default).

19.1 BOOLEAN AND SHIFT PSEUDO-FUNCTIONS

The pseudo-function and its use is described below. The operation treats the data as a bit string and pays no attention to any particular numeric format. The correct use is the responsibility of the programmer.

Operation	Number of Arguments	Argument Type	Function
LAND	2	1, 2, or 4 byte	logical and
LOR	2	1, 2, or 4 byte	logical or
LXOR	2	1, 2, or 4 byte	logical exclusive or
LCOMPL	1	1, 2, or 4 byte	logical complement (1's)
SHFTL	2	4 byte *	logical shift left
SHFTR	2	4 byte *	logical shift right

* The second argument is an integer which indicates the number of bits to shift.

The following truth tables give the results of the first four pseudo-functions.

LAND	0	1
0	0	0
1	0	1

LOR	0	1
0	0	1
1	1	1

LXOR	0	1
0	0	1
1	1	0

LCOMPL	
0	1
1	0

Individual bits may be tested by using the TBIT pseudo-function. It uses two arguments. The first is the variable to be tested and is four bytes or less. The second indicates the bit position to test, the left-most bit being zero. No checking is performed to insure the bit position requested falls within the length of the variable. The result is a four byte logical value of .TRUE. or .FALSE.

Another special purpose function is the MOD24 function. Its form is A=MOD24(A), where A must be a four-byte integer variable. This function returns the value of its argument except that the high-order byte is set to zero. The resulting value will be declared INTEGER*4.

19.2 BIT PSEUDO-FUNCTIONS

The bit setting facilities are not pseudo-functions but are used as statements. The pseudo-function must be set to a variable and use that same variable as the first argument. It may be subscripted but both references should be identical. The second argument specifies the bit to set and must be an integer with a value of zero to 63 inclusive. The bit facilities are:

```
V=BITON(V,K)   to turn on bit K
V=BITOFF(V,K)  to turn off bit K
V=BITFLP(V,K)  to reverse the value of bit K
```

19.3 EXAMPLES

Find the value of I and J, ORed, ANDed, and exclusive ORed together.

```
168)          DATA I,J/3,15/
              K=LAND(I,J)
              L=LOR(I,J)
              M=LXOR(I,J)
```

The results from I=3 (0011_2) and J=15 (1111_2) are K=3 (0011_2), L=15 (1111_2), M=12 (1100_2).

Find the logical complement of I.

```
169)          DATA I/0/
              J=LCOMPL(I)
```

The result in J is all bits on or -1.

Shift I 6 bits to the right and 15 bits to the left.

```
170)          DATA I/64/,N/15/
              K=SHFTR(I,6)
              L=SHFTL(I,N)
```

The results are K=1 (0001_2) and L=2097152 (2^{21}).

Test each bit in a four byte variable, F, and call a subprogram it the bit is off.

```
171)          DO 10 I=1,32
              IF (.NOT.TBIT(F,I-1)) CALL NOBIT
          10  ...
```

Test the first bit in each byte of a double precision variable, D; and if off, flip the first 2 bits, turn off the next 2 bits, and turn off the last 4 bits.

```
172)          DO 10 I=1,64,8
              IF (TBIT(D,I-1)) GO TO 10
              D=BITFLP(D,I-1)
              D=BITFLP(D,I)
              D=BITOFF(D,I+1)
              D=BITOFF(D,I+2)
              D=BITON(D,I+3)
              D=BITON(D,I+4)
              D=BITON(D,I+5)
              D=BITON(D,I+6)
        10    CONTINUE
```

19.4 STRUCTURE STATEMENT

STRUCTURE//V_{11},V_{12},V_{13},...//V_{21},V_{22},V_{23},...//V_{n1},,V_{n2},V_{n3},...$V_{n\ n}$

WHERE: V_{11},V_{12},V_{13},...V_{21},V_{22},V_{23},...$V_{n\ n}$

Represent names of variables that will be equated to displacement values. If these variables are declared in a Type statement, this statement must precede the STRUCTURE statement.

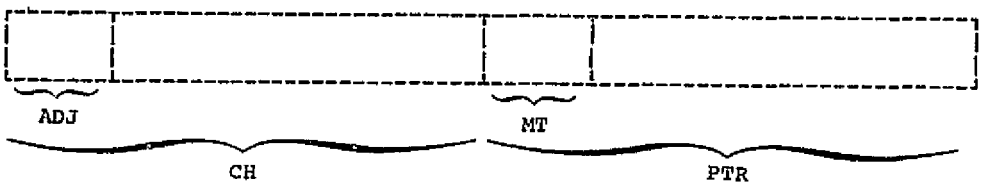
Note: The // immediately following the word STRUCTURE may be omitted.

The variables may be implicitly or explicitly declared as any type or length. They must not be dimensioned and must not appear in COMMON or EQUIVALENCE statements. A variable may appear more than once in STRUCTURE statements within a single program or subprogram provided it is given the same displacement by each program.

If D is the name of a structured variable, it must always appear in an executable statement with a single subscript, e.g., D(I). An expression such as D(I) refers to a variable of the type specified for D which is located in main storage at the base address specified by the value of the subscript expression, I, plus a displacement equal to the total number of bytes in the length specification of all the variables preceding D in the STRUCTURE statement in which it appears. For the object program to execute successfully, it is essential that the value of the subscript plus the displacement always be an integral multiple of the length of the referenced field. Displacements may not exceed 255. The subscript expression must be declared as integer or logical.

```
173)    LOGICAL*1     ADJ, MT
        INTEGER       CH, PTR
        STRUCTURE     CH, PTR//ADJ//CH, MT
```

Here the STRUCTURE statement shown in Example 173 is used to define a 2-word structure where the high-order byte of each word is overlapped by a 1-byte field.



If J contains a pointer to such a structure, its fields may be referenced as ADJ(J), CH(J), MT(J), and PTR(J).

If a structured variable is used incorrectly, the compiler may issue a diagnostic message.

Appendix A

DAIO

DAIO, direct access input/output, is a locally supported replacement package for FORTRAN's direct access I/O. A complete description can be found in the current version of the M&DO 360 User's Guide.

Appendix B

FMOVE

FMOVE is an assembly subroutine used to move data from one field to another. These fields may overlap. The subroutine makes use of the MVC instruction which moves up to 256 bytes, the equivalent of 64 four-byte variables with one instruction. When more than 256 bytes are to be moved, the subroutine loops moving the specified amount of data. FMOVE uses 172 bytes.

The calling sequence is:

```
CALL FMOVE(to,length,from)
```

"to" is the storage area which is to receive the data. It may be specified as a simple variable, an array name, or a subscripted variable.

"length" is the number of bytes of data to move. The four-byte integer may be specified as a constant, a variable, or an expression.

"from" is the storage area where the data is copied from. It may be specified as a simple variable, an array name, or a subscripted variable.

The effect of a call is to move "length" bytes of data from "from" to "to". To obtain the proper length in bytes, determine the number of variables to be moved and multiply by the length of each of the variables (1, 2, 4, 8, or 16 bytes).

FMOVE is located in SYS2.GSFCLIB and is automatically linked into the user's load module when the LINK, LINKGO, or LOADER procedure is executed.

To zero an array, or set it to any other specific value:

```
174)            DIMENSION A(2000)
                    .
                    .
                    .
                A(1)=0.0
                CALL FMOVE(A(2),7996,A(1))
```

The move is done one byte at a time, and the variables are filled A(2)=A(1) then A(3)=A(2) until all the remaining portion of the array is filled.

Move mixed data into the middle of a work array from a COMMON area:

```
175)      COMMON/COMMON/A(50),B(50),I(10),C(20),H(80)
          INTEGER*2 H
          LOGICAL*1 C
          DIMENSION   TEMP(300)
             .
             .
             .
          CALL FMOVE(TEMP(150),50*4+50*4+10*4+20+80*2,A)
```

The length has been expressed as the sum of each variable length times the number of elements. Since the "from" field is in COMMON, all the arrays are contiguous in storage.

Appendix C

FORTXDAM

1 FUNCTIONS

FORTRAN extended direct access method, FORTXDAM, is a subprogram with seven entry points. It moves unformatted fixed length blocks of data between disk storage and memory with no buffering and returns control to the user once the read or write operation is started. The amount of data which may be moved can be from one byte to one cylinder, 145880 bytes. Each record starts at the beginning of a track. To use space more efficiently, the record length should be as close to a multiple of the track length as is possible, 7264. No blocks or records are split across cylinders. The following table shows the relationship of record lengths to tracks and cylinders for 2314 disk storage. There is some work space required in each track.

Record Length	Tracks/Block	Blocks/Cylinder
1-7264	1	20
7265-14528	2	10
14529-21792	3	6
21793-29056	4	5
29057-36320	5	4
36321-43584	6	3
43585-72640	7-10	2
72640-145280	11-20	1

Figure 43 - Space Requirements for FORTXDAM Data Sets

2 ARGUMENTS

The subroutine is contained in SYS2.GSFCLIB and will be included in the load module automatically when the LINK, LINKGO, or LOADER procedure is used. It uses 1408 bytes of memory. FORTXDAM is re-entrant except when entry points XDOPEN, XDFORM, and XDCLOS are active.

The seven entry points, the calling sequence, and functions are documented below. The following names are used as symbolic arguments:

'field' - The area where the data is transferred. It must be large enough to contain all the data requested and may be specified as an array name or a simple or subscripted variable.

'length' - The record size in bytes to read or write. It must be a four byte integer value and may be specified as a constant, a simple or subscripted variable, or an expression.

'ddname' - The eight character left justified name of the JCL DD statement which defines the data set FORTXDAM is to read or write. All eight characters must be specified using blanks to pad the name on the right. It may be specified as a literal constant, a simple or subscripted variable name, or an array name.

'flag' - A four byte integer variable which contains the completion code of the previous operation. It may be a simple or subscripted variable.

'block' - A four byte integer value specifying the number of the block (or record) to transfer. It may be specified as a constant, an expression, or a simple or subscripted variable.

'unit' - A four byte integer variable which contains the internal file identification information.

3.0 CALLING SEQUENCE AND FUNCTION

3.1 XDOPEN

CALL XDOPEN(unit,length,ddname) is called first and prepares the data set control blocks for input/output operations. 'length' bytes are always transferred on subsequent access to the file 'unit'. It also acts as a flag with the following meanings:

a positive value - the data set was opened successfully

-1 - the data set was not opened successfully, probably a DD statement error

-2 - insufficient memory to open the data set, increase the region size

-4 - the data set record length is wrong, greater than 145280 bytes, or the SPACE field of the DD statement did not specify CYL

The value of 'unit' should not be changed once XDOPEN has executed successfully. If the length of the records in the data set is to be changed, it must be closed and reopened.

3.2 XDFORM

CALL XDFORM(unit,flag,field) formats a new data set for subsequent operations by FORTXDAM. XDOPEN must have been successfully executed. The entire data set is written with the data stored in 'field'. In this way the unused records may be flagged, set to a particular value, or certain fields initialized. If used in a multi-tasking environment, it should be noted that the eight bytes before 'field' are altered; upon completion of XDFORM they are restored. The values of 'flag' are:

a positive value - the data set was successfully formatted and may contain a maximum of 'flag' blocks (records)

a negative value - a write error occurred as explained:

-2 - a wrong length record condition was found, check 'length' in the call to XDOPEN

-4 - an uncorrectable error occurred

-8 - an unidentified error occurred

All data sets that are written for the first time must be formatted. If the length is being changed, then it must be reformatted.

3.3 XDWRIT

CALL XDWRIT(unit,block,field) starts the transfer of 'length' bytes from 'field' to record 'block' in file 'unit' (assigned by XDOPEN) on disk. A call to XDCHEK must precede any other I/O operation to 'unit'. The memory area 'field' should not be changed until the completion of XDWRIT as the values may be changed before the transfer takes place.

3.4 XDREAD

CALL XDREAD(unit,block,field) starts to transfer 'length' bytes from disk record 'block' of file 'unit' (assigned by XDOPEN) to 'field'. XDCHEK must be called before any other I/O operation to 'unit'. The memory locations 'field' should not be used until the completion of the I/O operation since the data may not yet be present.

3.5 XDTEST

CALL XDTEST(unit,flag) tests the progress of the I/O operation last requested on 'unit'. The calling program continues after the test is made. The meanings of 'flag' are:

1 - no I/O operations are active, XDCHEK has been called, and 'unit' is ready to read or write

0 - the previous operation is complete and XDCHEK needs to be called

-1 - an I/O operation is currently active

3.6 XDCHEK

CALL XDCHEK(unit,flag) completes an I/O operation to 'unit'. If the I/O operation is still proceeding, the calling program waits for it to complete. 'flag' indicates the status of the completed operation:

1 - all operations are complete

0 - record 'block' was successfully transferred

-1 - 'block' is too large, the file does not contain that many records

-2 - a wrong length record condition was found, and the number of bytes of data transferred is uncertain, check 'length' in XDOPEN

-4 - an uncorrectable I/O error occurred

-8 - an unidentifiable error occurred

3.7 XDCLOS

CALL XDCLOS(unit) is not required, unless changing the record length of a file. The system will automatically close all data sets used at the end of the program's execution.

4 JCL

The DD JCL statement for a FORTXDAM data set accessed via FORTXDAM should only specify the UNIT, SPACE, and optionally the DSN and DISP keywords. The SPACE parameter must be in the form:

```
// ... SPACE=(CYL,n,,CONTIG)
```

where 'n' is the number of cylinders needed to hold the records of the data set, see Figure 44. If full advantage is being made of the asynchronous input/output capabilities, the SEP subparameter of the UNIT field should be coded for new or work data sets. This will try to place the data sets on channels which are logically independent from one another and allows full physical overlapping of I/O operations.

5 EXAMPLES

Create a new file and reference it in the same program.

176)

```
C       ALLOCATE ARRAY SPACE AT NEAR 1 FULL TRACK(4*1800=7200)
        DIMENSION RECORD(1800),LOC(400)
C       INITIALIZE FILE TO ZEROS
        DATA RECORD/1800*0.0/
                              .
                              .
                              .
C       OPEN FILE AND ASSIGN FORTXDAM UNIT REFERENCE NUMBER
        CALL XDOPEN(IUNIT,7200,'FT01F001')
                              .
                              .
                              .
```

```
C       PREFORMAT DATA SET FOR USE AND INITIALIZE UNUSED RECORDS
        CALL XDFORM(IUNIT,NFLAG,RECORD)
C       CHECK TO BE SURE FILE PROPERLY FORMATTED
        IF (FLAG.GE.0) GO TO 10
        WRITE (6,20)NFLAG
   20   FORMAT (' *** FORMATTING ERROR CODE = ',I3/
      1         ' PROGRAM ENDED ***')
        STOP 4
                              .
                              .
                              .
C       WRITE ENTIRE FILE IN THIS LOOP, 'NUMREC' IS LESS THAN 401
   10   DO 100 I=1,NUMREC
                              .
                              .
                              .
C       START TO WRITE RECORD - 'LOC' ARRAY CONTAINS BLOCK NUMBERS
        CALL XDWRIT(IUNIT,LOC(I),RECORD)
C       DO OTHER CALCULATION WHICH DO NOT USE RECORD
                              .
                              .
                              .
C       CHECK STATE OF WRITE AND WAIT FOR COMPLETION
C       WAIT PLACED HERE SINCE RECORD ABOUT TO BE USED
        CALL XDCHEK(IUNIT,NFLAG)
        IF (NFLAG.LT.0) GO TO 200
   50   RECORD( ) = ...
                              .
                              .
                              .
        GO TO 100
  200   WRITE (6,210)NFLAG,LOC(I)
  210   FORMAT (' *** WRITE ERROR CODE ,'I3,' FOR BLOCK',I3)
        GO TO 50
                              .
                              .
  100                         .
C       START INPUT OPERATION FOR BLOCK II
        CALL XDREAD(IUNIT,II,RECORD)
                              .
                              .
                              .
C       NOW NEED TO USE RECORD - HALT PROGRAM UNTIL I/O DONE
        CALL XDCHEK(IUNIT,NFLAG)
        IF (NFLAG.GE.0) GO TO 300
        WRITE (6,220)NFLAG,II
  220   FORMAT (' *** ERROR CODE = ',I3,' READING BLOCK',I3)
        GO TO 100
  300
```

The JCL required for the data set would be:

```
//FT01F001 DD UNIT=2314,SPACE=(CYL,20,,CONTIG)
```

The number of cylinders was calculated as 400 records; one track one record. 400 tracks divided by 20 tracks per cylinder is 20 cylinders.

Copy one old file to another old file.

177)

```
C       ALLOCATE RECORD SPACE FOR TWO 14400 BYTE BUFFERS
                DIMENSION R1(3600),R2(3600)
                              .
                              .
                              .
C       OPEN FILES TO USE
        CALL XDOPEN(IUNITA,LEN,'FIRST   ')
        CALL XDOPEN(IUNITB,LEN,'SECOND  ')
                              .
                              .
                              .
                              .
                              .
                              .
C       READ IN FIRST RECORD FROM UNIT A INTO R1
        CALL XDREAD(IUNITA,1,R1)
                              .
                              .
                              .
C       SET UP LOOP - READ & WRITE DONE IN PAIRS OF
C       WRITE CURRENT RECORD AND READ NEXT RECORD
C       LOOP THEREFORE GOES BY TWOS AFTER INITIAL
C       RECORD READ
        NM1=NUMREC-1
        DO 240 I=1,NM1,2
                              .
                              .
                              .
C       WAIT FOR PREVIOUS READ TO FINISH TO WRITE RECORD FROM BUFFER R1
        CALL XDCHEK(IUNITA,IFLAG)
        IF (IFLAG) 100, 200, 200
  100   WRITE (6,105)IFLAG,I
  105   FORMAT (' ERROR CODE',I3,' AT FIRST CHECK IN LOOP, RECORD = ',I3)
C       DUMP CURRENT RECORD
  200   CALL XDWRIT(IUNITB,I,R1)
C       START TO READ NEXT RECORD INTO R2 BUFFER
        CALL XDREAD(IUNITA,I+1,R2)
C       NOW WAIT FOR UNIT B TO COMPLETE WRITING WHILE UNIT A IS READING
        CALL XDCHEK(IUNITB,IFLAG)
        IF (IFLAG.GE.0) GO TO 220
        WRITE (6,210)IFLAG,I
  210   FORMAT (' ERROR WRITING RECORD, CODE IS',I3,' RECORD = ',I3)
C       WAIT FOR READ TO FINISH BEFORE STARTING WRITE OF BUFFER R2
  220   CALL XDCHEK(IUNITA,IFLAG)
        IF (IFLAG.GE.0) GO TO 230
        WRITE (6,225)IFLAG,I
```

```
  225  FORMAT (' ERROR AT STATEMENT 220, CODE = ',I3,' FOR
     1RECORD',I3,'+1')
C       NOW START TO WRITE BUFFER R2 AND READ BUFFER R1
  230  CALL XDWRIT(IUNITB,I+1,R2)
       CALL XDREAD(IUNITA,I+2,R1)
C       CHECK FOR THIS READ AT TOP OF 240 LOOP
  240  CONTINUE
```

The JCL for the two DD cards would be:

```
//FIRST DD  DISP=SHR,SPACE=(CYL,10,,CONTIG),DSN=ORIGINAL
//SECOND DD UNIT=2314,SPACE=(CYL,10,,CONTIG),DSN=COPY,
//          DISP=(NEW,CATLG)
```

Save the results of calculations during an iteration when there is I/O time available. The previous results will be accessible later for restarting. An old FORTXDAM data set is to be used. At most save 10 results.

```
178)

C         ALLOCATE A WORK SPACE AND AN I/O SPACE
          DIMENSION COEFF(100,100,2),RHS(100,2)
                                    .
                                    .
                                    .
C         OPEN FILE TO BE USED
          CALL XDOPEN(IUNIT,40400,'SAVEDATA')
                                    .
                                    .
                                    .
C         PREPARE LOOP TO DO 1000 ITERATIONS AND INDICATOR FOR WHICH
C         AREA TO USE
          IDUMP=0
          J=2
          K=1
          DO 10 I=1,1000
          CALL ITER(COEFF(1,1,J),RHS(1,J))
C         CHECK IF FILE AVAILABLE FOR WRITE
          CALL XDTEST(IUNIT,N)
C         CONTINUE LOOP IF BUSY
          IF (N.LT.0) GO TO 10
C         FILE IS FREE WRITE RECORD, FINISH I/O OPERATION
          CALL XDCHEK(IUNIT,N)
          IF (N.GE.0) GO TO 5
          WRITE (6,6)N,IDUMP
      6   FORMAT (' ERROR ON INTERMEDIATE OUTPUT, ERROR IS',
         1I3,' RECORD IS',I3)
          GO TO 10
C         SET BLOCK NUMBER
      5   IDUMP=IDUMP+1
C         CHECK IF MORE THAN 10, IF SO RESET
          IF (IDUMP.GT.10) IDUMP=1
```

```
C         SET VALUE TO ADJUST BUFFER LOCATION SUBSCRIPT
          K=K*-1
          CALL XDWRIT(IUNIT,IDUMP,COEFF(1,1,J))
         1I3,' RECORD IS',I3)
C         MOVE DATA FROM BUFFER TO NEXT CALCULATION AREA
          CALL FMOVE(COEFF(1,1,J),COEFF(1,1,J+K),40000)
          CALL FMOVE(RHS(1,J),RSH(1,J+K),400)
C         RESET BUFFER ADDRESS
          J=J+K
     10   CONTINUE
C         WRITE LAST SAVED RECORD NUMBER
          WRITE (6,20) IDUMP
     20   FORMAT ('0 RECORD OF LAST SAVE IS',I3)
```

The JCL would be:

```
//SAVEDATA DD DSN=FORTXDAM.DATA,DISP=SHR,SPACE=(CYL,4,,CONTIG)
```

The number of cylinders of space required is calculated by using Figure 43. A record length of 40400 bytes takes 6 tracks; 3 records stored in each cylinder. Four cylinders will hold the required 10 records.

Appendix D

FTIO

FTIO, a FORTRAN callable subprogram, supports unformatted sequential I/O. Backspacing is not permitted, and the data to transfer must be continuous in storage. There are nine entry points which function as follows.

1 ENTRY POINTS AND FUNCTIONS

FREAD - read a record

FREADB - read a file backwards, last record first, etc.

FWRITE - write a record

REWIND - close the file and position at the start of the same file

UNLOAD - dismount the tape and free space used for controlling the file and buffers

POSN - position to the start of a specified tape file

LEAVE - close a file, free some file control space and buffers, and position at the end of the current file

MOUNT - mount a tape and optionally advance to a particular tape file

MEMBER - locate a member in a partitioned data set on direct access

2 HOW TO USE

The subprogram is located in SYS2.GSFCLIB and is automatically included when the LINK, LINKGO, or LOADER procedure is used. It requires 2589 decimal bytes.

In the discussion which follows, a record is one continuous group of individual data items. A data set is a related collection of records. A file is the manner in which a program refers to a data set. A file may consist of one or more data sets by concatenation. The FORTRAN unit number is the file name and is coded as the ddname on the DD statement. A tape file is a given data set on tape. There may be more than one stored per tape. When referenced through JCL, the physical sequential position on the tape is specified in the first field of the LABEL parameter of the DD statement.

A particular file may not be referenced both by FORTRAN and FTIO at the same time since certain system information within control blocks is different. Control blocks are created when a data set is

opened. When a data set is closed, certain pointers are reset and the buffers are freed. FTIO and FORTRAN may use the same files but only if the file has been closed by the first I/O package used to perform the operations before the other package opens the file. Data sets are implicitly opened by accessing the file. In FORTRAN the READ and WRITE statements cause an open, in FTIO calls to FREAD, FWRITE and FREADB open a data set. Closes are done by ENDFILE and REWIND statements in FORTRAN and by calls to REWIND, UNLOAD, POSN, LEAVE, MOUNT, and MEMBER.

FREAD, FWRITE, and REWIND are used just like the FORTRAN statements READ, WRITE, and REWIND. Several of the other calls allow the program to handle certain functions usually assigned to the JCL. Specific tapes may be dismounted (UNLOAD) and mounted (MOUNT). I/O operations may be directed at a specific physical tape file without a separate DD card for each tape file and dynamically changed (POSN and MOUNT). A tape file may be closed and logically positioned at the end of the physical tape file (LEAVE). For disk data sets which are contained in a partitioned data set, a specific member may be transferred and dynamically altered (MEMBER). Files may also be read backwards (FREADB), that is, read the records in reverse order. The contents of the record are unchanged.

3.0 EXAMPLES

The specific argument lists for each of the calls are shown below. Some entry points may have more than one form. Only the calls as shown are legal.

3.1 FREAD

```
CALL FREAD(record,unit,length,&end,&err)
CALL FREAD(record,ddname,length,Ø,&end,&err)
```

This will cause the number of bytes returned as 'length' to be read from 'unit' (FTunitFØØ1) or 'ddname' into memory at the location starting with 'record'. If an I/O error occurs, statement 'err' is passed control upon exiting FREAD. When the end of a file is read, statement 'rr' will have control. The data set will be opened if necessary.

3.2 FREADB

```
CALL FREADB(record,unit,length,&end,&err)
CALL FREADB(record,ddname,length,Ø,&end,&err)
```

The function is the same as FREAD except the records are read backwards. That is, the last record is read first until the first record. The data in each record is in its proper order. The record format of the file must be fixed blocked or unblocked (F or FB specified in the RECFM subparameter of the DCB operand). The data set will be opened if necessary.

3.3 FWRITE

```
CALL FWRITE(record,unit,length)
CALL FWRITE(record,ddname,length)
```

FWRITE will take 'length' bytes starting at location 'record' and write them to the file specified by 'unit' (FTunitFØØ1) or 'ddname'. The data set will be opened if necessary.

3.4 REWIND

```
CALL REWIND(unit)
CALL REWIND(ddname)
```

REWIND positions the file to the first record in the file referenced by 'unit' (FTunitFØØ1) or 'ddname'. The data set is closed if necessary.

3.5 UNLOAD

```
CALL UNLOAD(unit)
CALL UNLOAD(ddname)
```

For tape files only. The tape referenced by file 'unit' (FTunitFØØ1) or 'ddname' is dismounted and physically removed from the tape drive. All control block space is freed for reuse. The data set is closed if necessary.

3.6 POSN

```
CALL POSN(option,unit,tfile)
CALL POSN(option,ddname,tfile)
```

For tapes only. The tape mounted on file 'unit' (FTunitFØØ1) or 'ddname' is positioned at the start of physical tape file 'tfile'. 'option' specifies the type of I/O operation to be performed next. The data set is closed if necessary.

3.7 LEAVE

```
CALL LEAVE(unit)
CALL LEAVE(ddname)
```

The file referred to by 'unit' (FTunitFØØ1) or 'ddname' is positioned at the end of the current physical sequential file being processed. The data set is closed if necessary.

3.8 MOUNT

```
CALL MOUNT(option,unit,volume)
CALL MOUNT(option,unit,volume,tfile)
CALL MOUNT(option,ddname,volume)
CALL MOUNT(option,ddname,volume,tfile)
```

MOUNT will place the tape labelled as 'volume' on the tape drive assigned to file 'unit' (FTunitFØØ1) or 'ddname'. Optionally the tape may be positioned to the physical sequence tape file 'tfile'. Default is to the first tape file when not specified. 'option' specifies the type of I/O operation to perform next. The file will be closed if necessary.

3.9 MEMBER

```
CALL MEMBER(option,unit,member)
CALL MEMBER(option,ddname,member)
```

The next I/O operation will take place at the start of the member specified by 'member'. The operation will be as described by 'option'. The partitioned data set is referenced by file 'unit' (FTunitFØØ1) or 'ddname'. The data set will be closed if necessary.

4 ARGUMENTS

In the description of the calls to each of the entry points, the following symbols are used to represent the arguments.

record - A continuous area of storage in which the I/O transfer takes place. It may be an array name or a simple or subscripted variable and have 'length' bytes of storage following.

unit - The unit number of the file to be referenced. The four byte integer value must be between 1 and 50 inclusive. The name of the file is generated according to the rules of FORTRAN. It may be specified as a simple or subscripted variable or a constant or expression.

ddname - Is an eight byte literal which specifies the DD name for the file to be read. Trailing blanks must be included. It may be coded as a literal constant, a simple or subscripted variable, or an array name.

length - The number of bytes of data to be transferred. The four byte integer value may be coded as a constant, expression. or a simple or subscripted variable. The 'length' is calculated by multiplying the length in bytes of the data item (1, 2, 4, 8, or 16) by the number of items of each length.

option - Is a four byte integer value which specifies the type of I/O transfer which will be done. It may be given by a constant, expression, simple or subscripted variable. The values and their meanings are:

1 for input, read
2 for output, write
3 for input backwards, read backwards

tfile - Is a four byte integer value which specifies the physical tape file to which to position. It may be given in a constant, simple or subscripted variable, or an expression.

volume - Gives the tape volume serial number. The alphanumeric field is left justified and should contain trailing blanks to fill the six byte field. It may be written as a literal constant, a simple or subscripted variable, or an array name.

member - Is an eight byte name which is left justified and contains trailing blanks. It may be specified in any manner 'volume' is.

end - Specifies a statement number. This statement is given control when a read is issued and there are no more records in the file. It must be given as a one to five digit number which appears as the label of an executable or CONTINUE statement. It is coded with a leading ampersand, as shown in the description of the calls and in the examples.

err - Specifies a statement number. This statement is given control when an I/O error has occurred. It is specified as 'end' is.

5 RETURN CODES

If an invalid request is made of FTIO, the user condition code is set for the job step, and the step is terminated. The codes and their meanings are:

201 - 'unit' is out of range, larger than 50 or less than 1

202 - the file referenced is being used for direct access, rather than sequential input/output

210 - 'option' is invalid, greater than 3 or less than 1

220 - 'length' is invalid, check with the value coded in the LRECL subparameter of the DCB operand for FTunitF001 or 'ddname'

230 - the DD card for FTunitF001 or 'ddname' is missing

6 PROGRAM EXAMPLES

Create a file and use it later. The data will be written to FT10F001. Each record contains 1000 real variables.

```
179)                DIMENSION A(1000)
                        .
                        .
                        .
              C  WRITE OUT A RECORD
                    CALL FWRITE (A,10,4000)
```

```
                .
                .
                .
      C  RESET FILE TO READ FROM START OF FILE
            CALL REWIND (10)
                .
                .
                .
      C  READ IN A RECORD
            CALL FREAD (A,10,L,&99,&98)
                .
                .
                .
      C  END OF DATA SET FOUND
         99 CONTINUE
                .
                .
                .
      C  AN I/O ERROR FOUND
         98 CONTINUE
```

Read data into an array. When file DATAIN is all read, process the data.

```
180)              DIMENSION RECORD(80,100)
                      .
                      .
                      .
       C  READ IN UP TO 100 RECORDS
            DO 10 I=1,100
            CALL FREAD(RECORD(I),'DATAIN  ',L0,0,&100,&50)
          10  CONTINUE
      C       MORE THAN 100 RECORDS PRESENT - SKIP REST
                      .
                      .
                      .
          50  WRITE (6,60)I
          60  FORMAT ('  READ ERROR ON RECORD',I3,
             1' OF DATAIN - RECORD SKIPPED')
              GO TO 10
                      .
                      .
                      .
      C       PROCESS DATA
        100   CONTINUE
```

Read a record into a COMMON area. Process the individual variables and stop the program when all the data is read. The unit number and tape volume serial number are read on file five.

```
181)                REAL*8 VOL
                    COMMON /DATA/ A(3),B,I,L(6),X(9)
                         .
                         .
                         .
                    READ (5,5) IUNIT,IFILE,VOL
                  5 FORMAT (2I4,A6)
                    CALL MOUNT(1,IUNIT,VOL,IFILE)
                         .
                         .
                         .
                 10 CALL FREAD(A,IUNIT,L,&99,&30)
                         .
                         .
                         .
               C    PROCESS DATA
                         .
                         .
                         .
                    GO TO 10
                 30 WRITE (6,40) IUNIT
                 40 FORMAT ('ERROR READING UNIT',I3)
                    GO TO 10
                 99 STOP
                    END
```

Appendix E

ICMPAR

ICMPAR is an assembly language function used to compare up to 256 bytes of data. The function uses 96 bytes and makes use of a CLC instruction. ICMPAR is in SYS2.GSFCLIB and is automatically included when the LINK, LINKGO, or LOADER procedures are used. This instruction stops its left to right byte-by-byte comparison as soon as an inequality is found. The fields to compare may overlap. The value returned by the function depends on the relationship of the comparands. It is a full word integer which may also be treated as a four byte logical value.

The calling sequence is:

```
ICMPAR(field1,field2,length,offset1,offset2)
```

or,

```
ICMPAR(field1,field2,length)
```

"field1" is the first data string to compare. It may be specified as a simple variable, an array name, or a subscripted variable.

"field2" is the second data string to compare. It may be specified in any of the ways that "field1" is specified.

"length" is an integer value in four bytes which specifies the number of bytes of data to compare in "field1" and "field2". It may be a constant, simple, or subscripted variable or an expression. If the value is not in the range 1 to 256, the value used is taken as modulo 256.

The next two arguments are optional and may either both be left out or both be included in the argument list. If not used, both values default to zero.

"offset1" is the number of bytes to skip in the first data field before starting the comparison. The four byte integer quantity may be specified in any manner as outlined for "length". A value of zero skips no data and starts with the first byte of the data area given in "field1".

"offset2" has the same function as "offset1" but for "field2".

The result of the function is:

```
-1 or .FALSE. = "field1" is less than "field2"
 0 or .TRUE.  = "field1" is equal to "field2"
 1 or .FALSE. = "field1" is greater than "field2"
```

The sorting sequence for alphameric data is given on the lines below:

```
 ¢.<(+|&!$*);¬-/,%_>?:#@'="ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

Figure 44 - Standard IBM Collating Sequence

Compare the first five characters of A with the first five characters of B.

```
182)  REAL*8 A/'ABCDEFGH'/,B/'WXYDEF12'/
            .
            .
            .
      J=ICMPAR(A,B,5)
```

J will have a value of -1 or .FALSE..

If the three character string starting with the fourth character was done with A and B as in Example 175; the following would be used to set J:

```
183)  J=ICMPAR(A,B,3,3,3)
```

and J would be zero or .TRUE..

To compare the first eight values in C with the last eight, the code might be:

```
184)  REAL*8 C(100)
            .
            .
            .
      IF(ICMPAR(C,C,64,0,92*8))1,2,3
```

Note: The second offset is the element number minus one, since the first offset is zero, times the length of an element of the array.

Compare the fifth through tenth elements with the sixth through eleventh elements in array L.

```
185)  LOGICAL*1 L(20)
            .
            .
            .
      IF(ICMPAR(L(5),L(6),6))100,200,300
```

Appendix F

LOADMAP

LOADMAP produces the following: 1) a linkage editor map, 2) an alphabetic listing of all CSECT and entry point names, 3) a list of all unreferenced names, 4) cross-reference listings of called entry points and CSECT's as well as entry points and CSECT's which are called, and 5) a list of CSECT and entry point names with internal identification. A CSECT, or control section, is a main program, subprogram (subroutine or function), COMMON area, STATIC EXTERNAL area, or a pseudo-register vector.

The listings are controlled by keywords in the PARM field of the EXEC JCL statement. Any option not desired should be prefixed by the two letters NO. The following describes the function of the keywords with the default underlined. If only the defaults are to be used, no PARM field need be coded.

ID/NOID - list CSECT name, address, type, length, segment number, and internal identification number

MAP/NOMAP - produce a linkage editor map

LIST/NOLIST - produce an alphabetized listing of all CSECT and entry point names with the same information as ID provides

UNREF/NOUNREF - list all unreferenced, not called or used, CSECT and entry point names with the same information as ID provides

XREF/NOXREF - produce cross-reference listings to show all external references a CSECT makes (calls or references to COMMON, STATIC EXTERNAL areas, or pseudo-register vectors) by the calling CSECT and a list of where a CSECT, or entry point, is referenced by the called CSECT or entry point. Both listings are alphabetized by name.

LINECNT=82 - specify the total number of lines per page to be used for the reports. The two-digit quantity must be between 13 and 82 or the default will be used. Space for headings and footings is included.

The heading on each page includes: the report title, LOADMAP version number, time, date, page number, the contents of the PARM field, entry point address of the load module (in decimal), user region required for the program (exclusive of buffers and dynamically loaded modules) in decimal K, first volume serial number, DD name and load module attributes. The reports are multicolumn and read down the columns.

The program requires 46K + additional table space which is dependent upon the number of CSECT and entry point names and the number of external references. Most programs can be mapped in the default region or at most 100K and require 1/2 minute for both CPU and I/O

time on the 360/95. The amount of memory required to complete processing is reported on the bottom line of the next to last page of the report for each load module mapped. If not enough memory is assigned, the amount required to finish processing that phase of execution is given.

More than one load module may be mapped per execution, but the program options remain unchanged. A named DD card, which may be chosen to comment on the load module, is included for each load module and must include both the data set and member names. The reports are produced on SYSPRINT.

LOADMAP is in SYS1.LINKLIB and may be executed by either PGM= or the LOADMAP procedure. LIB is the symbolic name for the data set and MEM for the member name. Any other load modules to be mapped should have their own DD cards behind the EXEC card and not use the DD name SYSLIB.

186)
```
//MAP EXEC PGM=LOADMAP
//ONE DD DSN=USRID.XYZ.LOAD(MEMBER),DISP=SHR
//BACKUPV2 DD DSN=BACKUP.LOAD(VERSION2),DISP=SHR
//SYSPRINT DD SYSOUT=A

//MAP EXEC LOADMAP,LIB='USRID.XYZ.LOAD',MEM=MEMBER
//BACKUPV2 DD DSN=BACKUP.LOAD(VERSION2),DISP=SHR
```

Appendix G

OVLY

OVLY produces a tree diagram of an overlayed program from a load module. Each segment is shown with all of the CSECT's it contains and the length, in hexadecimal, of the segment. A CSECT, control section, is a main program, subprogram (either subroutine or function), COMMON area, STATIC EXTERNAL area, or a pseudo-register vector. Optionally a list or deck of linkage editor control cards is produced.

Two keywords may be specified in the PARM field of the EXEC JCL statement. NAME= is required and supplies the member name of the load module to be illustrated from the partitioned data set. DECK, if specified, supplies the overlay control cards which can regenerate the overlay structure.

The DD statements required are: SYSPRINT for the tree diagram, SYSPUNCH to contain the overlay control cards when DECK is specified, SYSLIB to point to the load module data set, and SYSUT1 which defines a scratch partitioned data set used as a work area.

The program is stored in SYS1.LINKLIB and also has a procedure to call it. The procedure assumes a listing of the overlay control cards is desired. LIB is the symbolic name used for the data set name, MEM for the load module member name, and PUNCH=B will punch the overlay control cards. OVLY uses 46K and illustrates most programs in 1/2 minute for both CPU and I/O time. The load module may either be a regular or multi-regi n overlay.

The following examples punch an overlay control card deck.

```
187)   //TREE      EXEC  PGM=OVLY,PARM='NAME=MEMBER,DECK'
       //SYSPRINT  DD    SYSOUT=A
       //SYSPUNCH  DD    SYSOUT=B
       //SYSUT1    DD    UNIT=2314,SPACE=(TRK,(2,1,1))
       //SYSLIB    DD    DSN=USRID.LOADMOD.LOAD,DISP=SHR
```

Using the procedure the example would be:

```
188)   //TREE  EXEC  OVLY,LIB='USRID.LOADMOD.LOAD',
       //       MEM=MEMBER,PUNCH=B
```

Appendix H

TIMING SUMMARY

The table below lists all of the examples presented in this guide. The examples are grouped according to the tests performed with a blank line separating each test. The last column indicates which examples may be compared since they were in the same job step or are intended to show equivalent code.

Appendix H

Example	CPU Time (Sec)	Percent of Total Run	Total Run Time (Min)	# Exec.	Examples Which May be Compared
1	96.0	6.96	23.000	70,000	1-10
2	27.6	2.00			
3	89.4	6.48			
4	28.8	2.09			
5	367.2	26.61			
6	328.2	23.78			
7	79.2	5.74			
8	27.6	2.00			
9	70.8	5.13			
10	55.8	4.04			
11	40.67	7.80	8.693	300,000	11-26
12	7.53	1.44			
13	-	-			
14	-	-			
15	-	-			
16	40.67	7.80			
17	10.07	1.93			
18	80.74	15.48			
19	76.17	14.60			
20	76.78	14.72			
21	80.19	15.37			
22	25.06	4.80			
23	73.70	14.13			
24	1.67	0.32			
25	1.43	0.27			
26	6.92	1.33			
27	19.2	-		1,440,000	27-30
28	16.8	-			
29	4.2	-			
30	3.6	-			
31	-	15.46			31-32
32	-	9.20			
33	75.0			2,000,000	33-34
34	25.8				
35	138.33	19.90	11.586	50,000	35-39
36	208.40	29.98			
37	103.23	14.85			
38	-	-			
39	51.65	7.43			

Example	CPU Time (Sec)	Percent of Total Run	Total Run Time (Min)	# Exec.	Examples Which May be Compared
40	-				
41	-				
42	-				
43	128.52	38.00	5.637	2,500,000	43-44
44	124.26	36.74			
45	-				
46	8.55	1.72	8.281	100,000	46-55
47	10.84	2.18			
48	21.32	4.29			
49	13.61	2.74			
50	29.41	5.92			
51	14.01	2.82			
52	36.72	7.39			
53	15.80	3.18			
54	46.26	9.31			
55	17.14	3.45			
56	7.03	1.10	10.658	1,800,000	56-73
57	15.97	2.50			
58	8.57	1.34			
59	12.85	2.01			
60	12.92	2.02			
61	12.66	1.98			
62	16.95	2.65			
63	12.79	2.00			
64	19.12	2.99			
65	13.30	2.08			
66	21.36	3.34			
67	13.81	2.16			
68	24.36	3.81			
69	13.24	2.07			
70	27.05	4.23			
71	12.41	1.94			
72	30.18	4.72			
73	13.43	2.10			

"Page missing from available version"

Example	CPU Time (Sec)	Percent of Total Run	Total Run Time (Min)	# Exec.	Examples Which May be Compared
103	8.41	2.76			
104	8.26	2.71			
105	11.40	3.74			
106	-				
107	-				
108	31.35	4.34	12.041	40,000,000	108-134
109	30.34	4.20			
110	17.62	2.44			
111	17.62	2.44			
112	29.76	4.12			
113	30.99	4.29			
114	18.28	2.53			
115	17.70	2.45			
116	27.38	3.79			
117	27.38	3.79			
118	15.01	2.09			
119	15.17	2.10			
120	27.01	3.74			
121	27.67	3.83			
122	15.46	2.14			
123	16.04	2.22			
124	16.04	2.22			
125	17.12	2.37			
126	14.74	2.04			
127	15.82	2.19			
128	15.96	2.21			
129	18.21	2.52			
130	14.59	2.02			
131	16.25	2.25			
132	10.76	1.49			
133	10.55	1.46			
134	19.29	2.67			
135	17.49	5.74	5.079	2,000,000	135-136
136	30.95	10.15			94a-107
137	59.89	10.91	9.149	1,800,000	137-139
138	46.11	8.40			
139	61.15	11.14			

Example	CPU Time (Sec)	Percent of Total Run	Total Run Time (Min)	# Exec.	Examples Which May be Compared
140	29.10	12.13	3.999	40,000	140-142
141	27.16	11.32			
142	31.67	13.20			
143	37.18	15.77	3.929	300,000	143-145
144	33.52	14.22			
145	26.36	11.18			
146	32.39	5.52	9.780	1,800,000	
147	30.22	5.15	9.870	1,800,000	147-152
148	13.14	2.24			
149	12.85	2.19			
150	12.79	2.18			
151	12.14	2.07			
152	9.56	1.63			

Example	Execution Time (Minutes) CPU	I/O	# of Passes	Examples Which May be Compared
153	7.190	9.543	1000	153-157
154	3.189	0.538	1000	153-157
155	2.700	0.535	1000	153-157,159,160
156	2.765	0.538	1000	153-157,159,160
157	3.161	0.543	1000	153-158
158	2.852	13.208	1000	157
159	0.265	0.537	1000	155,159,160,164
160	0.097	0.498	1000	156,159,160
161	2.199		1000	161,162,163
162	1.756		1000	161
163	3.550	0.558	1000	161
164	0.143	0.531	1000	159
165	2.341	10.799	4000	166,167
166	0.707	7.236	4000	165,167
167	6.257	10.389	>10,000,000	165,166