NASA

Technical Memorandum 81995

Comparison of and Conversion Between Different Implementations of the FORTRAN Programming Language

Lloyd Treinish

(NASA-TM-81995) COMPARISON OF AND CONVERSION BETWEEN DIFFERENT IMPLEMENTATIONS OF THE FORTRAN PROGRAMMING LANGUAGE (NASA) 55 p HC A04/MF A01 CSCL 09B

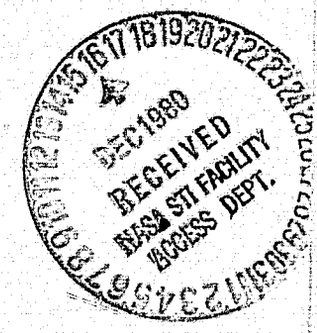
N81-12762

Unclas
G3/61 39830

AUGUST 1980

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771



TM 81995

COMPARISON OF AND CONVERSION BETWEEN DIFFERENT IMPLEMENTATIONS OF THE FORTRAN PROGRAMMING LANGUAGE

Lloyd Treinish

Information Extraction Division

NASA/Goddard Space Flight Center

Greenbelt, Maryland

August 1980

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

COMPARISON OF AND CONVERSION BETWEEN DIFFERENT IMPLEMENTATIONS OF THE FORTRAN PROGRAMMING LANGUAGE

Lloyd Treinish

Information Extraction Division

NASA/Goddard Space Flight Center

Greenbelt, Maryland

ABSTRACT

This document is intended to be used as a guideline for computer programmers who may need to exchange FORTRAN programs between several computers. The characteristics of the FORTRAN language available on three different types of computers are outlined, and procedures and other considerations for the transfer of programs from one type of FORTRAN to another are discussed. In addition, the variance of these different FORTRAN's from the FORTRAN 77 standard are discussed.

CONTENTS

COMPARISON OF AND CONVERSION BETWEEN DIFFERENT IMPLEMENTATIONS OF THE FORTRAN PROGRAMMING LANGUAGE

INTRODUCTION

This document is intended to be used as a guideline for scientific programmers at Goddard Space Flight Center who may need to exchange FORTRAN programs between several computers. The characteristics of the FORTRAN language available on three different types of computers are outlined, and procedures and other considerations for the transfer of programs from one type of FORTRAN to another are discussed. Specifically, FORTRAN from IBM (e.g., SACC System 360/91 and 360/75, M&DO 360/95 and 360/75), Xerox (e.g., AE Sigma 9) and DEC (e.g., DCAC PDP 11/70) are examined.

The constructs and syntax acceptable to the FORTRAN's from the three computer manufacturers outlined below have many mutual incompatibilities as well as a considerable number of differences and inadequate capabilities with respect to the FORTRAN 77 standard (ANSI X3.9-1978, see references 11 and 12). If FORTRAN programs developed on any one of the aforementioned types of computers are to be at all portable then as many of the "machine-peculiar" FORTRAN features should be avoided as possible. However, because of the lack of a standard that encompasses all three types of FORTRAN adequately, no specific guidelines for "machine-independent" code can be realistically offered.

IBM FORTRAN

IBM has marketed many different versions of the FORTRAN programming language. For example, the SACC IBM 360's support five different FORTRAN compilers, FORTRAN's G, G1, H, H extended and Code and Go. The G1 and H extended compilers represent upward-compatible supersets of the G and H compilers, respectively. However, the latter two compilers are no longer supported as "Program Products" from IBM. In any further discussion of IBM FORTRAN, the G1 version will be considered as an "IBM standard." For completeness, the differences between the five IBM compilers will be noted.

The FORTRAN G1 compiler is the IBM Program Product designed for the development phase of FORTRAN programs. It compiles programs relatively fast while producing moderately inefficient (i.e., unoptimized) code. Since much of the effort in developing new FORTRAN programs often occurs in debugging, whose costs tend to be concentrated in the compilation phase rather than in short test execution of the programs, the G1 compiler fills this need for both foreground and background applications. The version of FORTRAN acceptable to the G1 compiler also possesses a debugging facility that can be invoked through special statements, and the compiler can produce code that can be executed through the interactive debugger, TESTFORT. It should be noted that the Code and Go FORTRAN compiler accepts a G1 type of FORTRAN but executes the code after compilation instead of producing an object module. It can provide additional savings in program development cost over the G1 compiler because it can streamline the procedure to compile and test execute new code. Of course, once a program is debugged it should be compiled using an optimizing compiler for run-time efficiency.

The G1 compiler, for the most part, represents a small subset of the FORTRAN 77 standard. The following is a summary of the important G1 facilities not found in or different than the standard:

1. The PUNCH statement.
2. The debugging statements: DEBUG, AT, TRACE ON, TRACE OFF and DISPLAY.

3. Alternate returns in subroutine calls marked with an ampersand [e.g., CALL X (&10)].
4. Length specification in 'type' statements. (G1 allows INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*4, REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 statements in addition to what the standard allows, INTEGER, REAL, DOUBLE PRECISION, LOGICAL and COMPLEX statements.)
5. Direct access I/O specified in a different manner. [e.g., DEFINE FILE and FIND statements, READ or WRITE (9'IPTR)].
6. Use of hexadecimal constants and Z formats.
7. NAMELIST I/O [e.g., NAMELIST statement, READ or WRITE (device, namelist)].
8. Some SERVICE routines (e.g., ERRSET, DUMP, SLITE).
9. COMPLEX*16 intrinsic functions.
10. The following intrinsic functions (slight spelling differences between the same routines are not listed):
    (a) COTAN/DCOTAN – cotangent
    (b) ERF/DERF/ERFC/DERFC – error function
    (c) GAMMA/DGAMMA/ALGAMMA/DLGAMMA – gamma function.

The FORTRAN G compiler accepts a large subset of the language processed by the G1 compiler. FORTRAN G does not permit list-directed I/O, does not generate code for TESTFORT, does not round infinite binary expansions (e.g., G1 rounds 3.9999 . . . to 4.000 and G truncates 3.9999 . . . to 3.99999), etc.

The FORTRAN H extended compiler is designed to generate optimized object code for production-type execution of FORTRAN programs. It is, therefore, expensive to use but the resultant code is relatively cheap to execute. FORTRAN H extended does not provide the debugging facilities of FORTRAN G1 but does have several other extensions to the G1 language, none of which are within the FORTRAN 77 standard:

1. Asynchronous I/O [e.g., READ or WRITE (device, ID = identifier) list, and the WAIT statement].

2. Extended precision arithmetic (i.e., REAL*16 and COMPLEX*32) and the intrinsic functions and subroutine library to support it.
3. External statement extension [e.g., EXTERNAL &subprogram-name].
4. Automatic function selection [e.g., use of the GENERIC statement].

The FORTRAN H compiler is also designed to generate optimized object code for the production-type execution of FORTRAN programs. FORTRAN H represents a subset of FORTRAN H extended by providing less powerful optimization, allowing no list-directed I/O and permitting none of its language extensions over FORTRAN G1. The H compiler accepts and should be consistent with all FORTRAN G code except for the latter's debugging facilities. However, there is no guarantee, even from IBM, that code written for the G or G1 compilers will execute in the same manner if it is compiled by the H or H extended compiler.

XEROX FORTRAN

The Xerox Sigma 9 computer supports three FORTRAN compilers, Xerox Extended FORTRAN IV (FORT4), FORTRAN Load and Go (FLAG) and an improved, new FORTRAN (ANSF). All three compilers are available on the AE (Atmosphere Explorer) Sigma 9 but at GSFC only FORT4 is supported and used by the AE community. Thus, FORT4 will be considered the standard Xerox compiler for this discussion.

ANSF represents a fairly complete subset of the FORTRAN 77 standard. Although the ANSF compiler is better than the FORT4 compiler, the languages acceptable to each compiler have limited overlap, and hence, are not very compatible.

The FLAG compiler operates in one pass by compiling and executing FORTRAN code in a single step. In many respects FLAG is similar to the IBM Code and Go compiler. Hence, FLAG can be used to reduce the cost of compilation during the development phase of new software. However, FLAG accepts only a subset of the FORTRAN allowed by FORT4 and thus, the two compilers are not compatible.

The FORT4 compiler is very incompatible with the FORTRAN 77 standard. It can produce code to permit program execution through an on-line debugger. The following is a summary of the important FORT4 facilities not found in or different than the standard:

1. Conditional compilation [i.e., X in column one].
2. In-line assembly language.
3. Compound statements [e.g., B = C; A = B].
4. Hexadecimal constants.
5. DOUBLE COMPLEX data.
6. Extended and optional relational expressions [e.g., A.EOR.B, I < J < 10].
7. Multiple assignment statements [e.g., A = B = C].
8. The END LABELS statement.

9. Global statement labels [e.g., 10$].

10. The REPEAT statement.

11. NAMELIST I/O [e.g., the NAMELIST statement, READ or WRITE (device, namelist), INPUT or OUTPUT (device) list, *].

12. Direct access and list-directed I/O specified in a different manner.

13. R, Z, M, backward (i.e., negative) X, widthless and adjustable format specifications.

14. Formatted data conversions (i.e., ENCODE/DECODE) specified in a different manner.

15. Asynchronous I/O [e.g., calls to BUFFER IN, BUFFER OUT and ICHECK].

16. FORTRAN II I/O (some forms like READ or WRITE DISK or TAPE, etc.).

17. Carriage control with a '+' in column one not supported on the Sigma 9, and '-' in column one not supported in the standard.

18. GLOBAL data.

19. Alternate returns in subroutine calls marked with an ampersand, dollar sign or letter S [e.g., CALL X(&10), CALL X(10$), CALL X(10S)].

20. Some service routines [e.g., EOFSET].

21. DOUBLE COMPLEX intrinsic functions.

22. The following intrinsic functions (slight differences in spelling are not listed):

    (a) CASIN/CATAN/CACOS/CCOSH/CSINH/CSNGL/CTANH/CTAN/CINT - functions of a complex variable.

    (b) ISL/IAND/IEOR/IF/INOT/IOR/ISA/ISC - boolean functions.

DEC FORTRAN

The DEC PDP-11 computer supports two FORTRAN compilers, FORTRAN IV (FOR) and FORTRAN-IV PLUS (F4P). Since the DCAC (Pioneer Venus/Stratosphere Data Communication and Analysis Center) PDP 11/70 only supports the latter, only F4P will be discussed. F4P represents a small subset of the FORTRAN 77 standard with a few incompatibilities. It conforms to the standard better than any of the aforementioned compilers except for the Xerox ANSF compiler. F4P can generate code to permit program execution through the on-line debugger, ODT. The following is a summary of the important F4P facilities not found in or different than the standard:

1. Length specification in 'type' statements. (F4P allows INTEGER*2, INTEGER*4, LOGICAL*2, LOGICAL*4, BYTE, REAL*4 and REAL*8 statements in addition to the standard, INTEGER, REAL, DOUBLE PRECISION, LOGICAL and COMPLEX statements.)
2. Comments after statements.
3. Direct access I/O specified in a different manner [e.g., DEFINE FILE and FIND statements, READ or WRITE (9'IPTR)].
4. Formatted data conversion specified in a different manner [i.e., ENCODE/DECODE].
5. Somewhat different format for the OPEN and CLOSE statements.
6. The INCLUDE statement.
7. Conditional compilation (i.e., D in column one).
8. The ACCEPT/TYPE statements.
9. Use of octal constants.
10. The VIRTUAL statement.
11. Octal, Q and adjustable formats.
12. Some service routines (e.g., ERRSNS, USEREX).
13. The following intrinsic functions (slight differences in spelling are not listed):
    (a) IAND/IOR/IEOR/NOT/ISHFT – boolean functions.
    (b) RAN – random number generator.

CONVERTING IBM FORTRAN TO XEROX FORTRAN

In converting a FORTRAN program from the SACC IBM 360 computers for use on the AE Sigma 9 one must remove any of the FORTRAN H extended special language features (possibly excepting asynchronous I/O) that the other IBM FORTRAN's do not allow, and any of the Debug Facility statements accepted by FORTRAN G, FORTRAN G1 or the Code and Go compilers. For the most part, Sigma 9 FORT4 represents a superset of IBM FORTRAN G1. In addition, the following types of statements will require some modifications:

1. LOGICAL*1 change to LOGICAL (only the four byte version is allowed).
2. INTEGER*2 change to INTEGER (only the four byte version is allowed).
3. READ (device, *) list change to INPUT (device) list.
4. WRITE (device, *) list change to OUTPUT (device) list.
5. References to cotangent, error or gamma functions.
6. IBM FORTRAN H extended asynchronous I/O statements [e.g., the WAIT statement, READ or WRITE (device, ID= identifier) list] perhaps can be simulated by Xerox FORT4 asynchronous I/O statements [e.g., calls to BUFFER IN, BUFFER OUT and ICHECK].

Xerox FORT4 is designed to be upward-compatible with the IBM FORTRAN's. The Sigma 9 is a 32-bit EBCDIC machine using hexadecimal notation like the 360 and data are stored in a similar manner on both computers. The use of the FTIO or DAIO packages in programs written for the SACC 360's can be simulated using the GETPUT package on the Sigma 9. The acceptable argument values for intrinsic mathematical functions may not be compatible between the Sigma 9 and the 360. The ability of many of the character and bit manipulation routines, and service routines (e.g., INCORE, KTIME, LAND, SLITE) on the SACC computers are available in various forms on the Sigma 9. Sigma 9 FORT4 only has full word data types so that IBM programs that use smaller types must be changed. The only problem that users of the 360 may encounter in attempting to convert IBM programs to Xerox FORTRAN is with the size of the Sigma 9. A single user on the AE Sigma 9 can only request 288K bytes of memory. This may prevent the use of some large IBM FORTRAN programs on the Sigma 9. An overlay structure that is set up

through linkage editing may help to alleviate storage difficulties. It should be noted, however, that a load module generated on the Sigma 9 for a particular program will, in general, be much smaller than its equivalent on the 360. Hence, a large IBM program may result in a small enough load module to run on the Sigma 9. In addition, the speed of the Sigma 9's cpu and the capability of its software will constrain the use of IBM programs. The FORT4 compiler is not particularly fast nor does it optimize its generated code. It tends to be quite slow in compiling programs that make extensive use of non-executable statements, especially DATA statements. Since the Sigma 9 is roughly an order of magnitude slower than the SACC 360/91 and Sigma 9 FORT4's generated code is not optimized, IBM FORTRAN programs may execute too slowly to be practical on the Sigma 9. It should be noted that negative real numbers are stored in a different fashion on the 360 and the Sigma 9 and that this will cause some difficulties in converting programs that access real data at the bit level. However, if the aforementioned items are kept in mind, the conversion of IBM FORTRAN to Xerox Extended FORTRAN should be relatively simple and painless.

CONVERTING XEROX FORTRAN TO IBM FORTRAN

The conversion of Xerox FORT4 programs to IBM FORTRAN is an extensive and messy proposition. The code will have to be rewritten to eliminate many Xerox-acceptable constructs, including the following:

1. Conditional compilation [i.e., "X" in column one].
2. In-line assembly language.
3. Compound statements [e.g., A=B; C=A].
4. Multiple assignments [e.g., A=B=C].
5. Global references [e.g., GLOBAL data, global labels like 10$].
6. REPEAT loops.
7. R, M, backward (i.e., negative) X, widthless and adjustable format specifications.
8. Extended and optional relational expressions [e.g., A.EOR.B, I < J < 10].
9. Complex inverse trigonmetric, hyperbolic and "type" conversion functions.
10. Hollerith constants in other than FORMAT, "type" or DATA statements [e.g., A=4HABCD].
11. Expressions in I/O lists [e.g., WRITE (6, 10) X**SIN(X)+3.0].
12. The END LABELS statement.
13. Negative, zero, real or complex DO indices.
14. Backward DO loops.
15. Some Fortran II I/O [e.g., READ TAPE].

The following items in a Xerox FORT4 program are acceptable to the SACC 360's but their syntax must be modified:

1. DOUBLE COMPLEX change to COMPLEX*16.
2. NAMELIST I/O (if Xerox format is used).
3. List directed I/O:

   OUTPUT (device) list change to WRITE (device, *) list

   INPUT (device) list change to READ (device, *) list.

4. ENCODE/DECODE statements can be simulated by the INCORE routine.
5. Asynchronous I/O (calls to BUFFER IN, BUFFER OUT and ICHECK) may be simulated by FORTRAN H extended code (READ or WRITE with ID and the WAIT statement).
6. Alternate returns in subroutine calls (if Xerox format is used).
7. Boolean functions may be simulated by routines available on the SACC 360's.
8. Variable names shortened to six characters.
9. Some service routines may have similar purposes (e.g., ERRSET).
10. Arguments to mathematical routines may have different acceptable values.

Since Xerox FORT4 represents a superset, for the most part, of IBM FORTRAN and both the Sigma 9 and the 360 are thirty-two-bit EBCDIC machines using hexadecimal notation, the Sigma 9 FORT4 and the 360 FORTRAN's are reasonably compatible. However, in practice, the modifications on a Xerox FORT4 program required to convert it for use on the 360 are so numerous that a design consideration of Xerox software should be the question of portability. If new software is to be possibly copied from the Sigma 9 to another computer then the use of the "Xerox-peculiar" constructs should be avoided. It should be noted that negative real numbers are stored in different fashions on the 360 and the Sigma 9 and that this will cause some difficulties in converting programs that access real data at the bit level.

CONVERTING IBM FORTRAN TO DEC FORTRAN

In converting a FORTRAN program from the SACC IBM 360 computers for use on the DCAC PDP 11/70 one must remove any of the FORTRAN H extended special language features that the other IBM FORTRAN's do not allow or any of the Debug Facility statements accepted by FORTRAN G, FORTRAN G1 or the Code and Go compilers. However, the language acceptable to the IBM FORTRAN's and DEC F4P are reasonably compatible. The following type of code will have to be removed or changed from IBM FORTRAN:

1. Optional returns from subroutines [e.g., CALL X(&10)].
2. Data initialization in "type" statements [e.g., REAL IX/3.45/].
3. COMPLEX*16 data and their intrinsic functions.
4. NAMELIST I/O [e.g., NAMELIST statement, READ or WRITE (device, namelist)].
5. The PUNCH statement.
6. References to cotangent, error or gamma functions.
7. "Type" specification should be explicit.
8. Use of hexadecimal constants and Z formats.

The PDP 11/70 is a sixteen-bit ASCII machine using octal notation while the 360 is a thirty-two-bit EBCDIC machine using hexadecimal notation. Any explicit references to EBCDIC characters or hexadecimal constants will have to be translated to ASCII or octal notation, respectively. The default size of INTEGER and LOGICAL type variables is two bytes on the 11/70 while it is four bytes on the 360. The use of explicit type declarations or the /I4 switch when DEC F4P is invoked will solve this problem. Generally, the use of two-byte variables is preferred, where possible, for the sake of speed of operation on the 11/70. Since the 360 and the 11/70 do not store their data in the same manner, the conversion of some FORTRAN programs may be severely constrained. Appendices A and B discuss this problem in detail. Some of the capabilities of the FTIO and DAIO packages on the SACC computers can be simulated by the OPEN/CLOSE statements in F4P. IBM FORTRAN and DEC F4P interpret the ENTRY statement in

different fashions (see Appendix C for one example) and its use should be avoided. The acceptable argument values for intrinsic mathematical functions may not be compatible between the PDP and the 360. The ability of many of the character and bit manipulation routines, and service routines (e.g., INCORE, KTIME, LAND, SLITE) on the SACC computers are available in various forms on the DCAC 11/70. The only other problem that users of the 360 may encounter in attempting to convert IBM programs to DEC F4P is with the size of PDP 11/70. A single user on the DCAC 11/70 can only request 64K bytes of memory. This may prevent the use of some large IBM FORTRAN programs on the 11/70. An overlay structure that is set up through task building or the use of VIRTUAL arrays may help to alleviate storage difficulties. It should be noted, however, that a task image generated on the 11/70 for a particular program will, in general, be much smaller than its equivalent load module on the 360. Hence, a large IBM program may result in a small enough task image to run on the 11/70. In addition, the fact that 11/70 is much slower than the 360/91, and despite F4P's ability to optimize code, IBM FORTRAN programs may not execute quickly enough to be practical on the 11/70.

CONVERTING DEC FORTRAN TO IBM FORTRAN

The conversion of DEC F4P programs to IBM FORTRAN will require more work than the converse. The following F4P constructs will have to be removed or changed:

1. The PARAMETER statement.
2. Expand INCLUDE statements.
3. ENCODE/DECODE statements change to use INCORE on the SACC 360's.
4. Comments after statements.
5. OPEN/CLOSE statements may be simulated by FTIO/DAIO.
6. Conditional compilation (i.e., "D" in column one).
7. ACCEPT/TYPE statements.
8. Octal constants.
9. The VIRTUAL statement.
10. Boolean functions may be substituted with equivalent SACC routines.
11. The BYTE statement change to LOGICAL*1.
12. O, adjustable, $, : and Q format specifications.
13. LOGICAL*2 data.
14. Backward DO loops.
15. Negative, zero, REAL or COMPLEX DO indices.
16. Hollerith constants in other than FORMAT or DATA statements (e.g., A = "ABCD").
17. Shorten variable names to six characters.
18. Some service routines have similar purposes (e.g., ERRSET).
19. "Type" specifications should be explicit.
20. Arguments to mathematical routines may have different acceptable values.

The PDP 11/70 is a sixteen-bit ASCII machine using octal notation, while the 360 is a thirty-two-bit EBCDIC machine using hexadecimal notation. Any explicit references to ASCII characters or octal constants will have to be translated to EBCDIC or hexadecimal notation,

respectively. The default size of INTEGER and LOGICAL type variables is two bytes on the 11/70 while it is four bytes on the 360. "Type" statements may have to be incorporated to enable the program to run equivalently on the 360's. Since the 360 and the 11/70 do not store their data in the same manner, the conversion of some FORTRAN programs may be severely constrained. Appendices A and B discuss this problem in detail. Some of the capabilities of the OPEN/CLOSE statements on the 11/70 computer can be simulated by the FTIO and DAIO packages on the 360. IBM FORTRAN and DEC F4P interpret the ENTRY statement in different fashions (see Appendix C for one example) and its use should be avoided. The ability of many of the character and bit manipulation routines, and service routines (e.g., INCORE, KTIME, LAND, SLITE) on the SACC computers are available in various forms on the DCAC 11/70. Since the 360's have so much core storage available, any program overlays and virtual arrays used on the PDP could be eliminated on the 360. The use of the VIRTUAL statement in a F4P program could be changed to the use of a DIMENSION statement in an IBM program.

CONVERTING XEROX FORTRAN TO DEC FORTRAN

To convert a Sigma 9 FORT4 program to PDP F4P the code will have to be rewritten to eliminate many Xerox-acceptable constructs including the following:

1. In-line assembly language.
2. Compound statements [e.g., A=B; B=C].
3. Multiple assignments [e.g., A=B=C].
4. Global references [e.g., GLOBAL data, global labels like 10$].
5. REPEAT loops.
6. R, M, Z, backward (i.e., negative) X, and widthless format specifications.
7. Extended and optional relational expressions [e.g., A.EOR.B, I < J < 10].
8. Complex inverse trigonometric, hyperbolic and type conversion functions.
9. DOUBLE COMPLEX data and their intrinsic functions.
10. Asynchronous I/O (calls to BUFFER IN, BUFFER OUT, and ICHECK).
11. NAMELIST I/O (in either Xerox or IBM format).
12. Alternate returns in subroutine calls (in either Xerox or IBM format).
13. The END LABELS statement.
14. Hexadecimal constants.
15. Some FORTRAN II I/O [e.g., READ TAPE].

The following items in a Xerox FORT4 program are acceptable to the PDP 11/70 but their syntax must be modified:

1. Boolean functions are available under different names.
2. The GETPUT package can be partially simulated through the OPEN/CLOSE statements.
3. Variable names of any length are acceptable but the first eight characters are significant to the Sigma 9 while only the first six are significant to the 11/70.
4. DEC F4P uses quoted Hollerith strings outside of DATA and FORMAT statements (e.g., A='ABCD' not A=4HABCD).
5. "Type" specifications should be explicit.

6. Conditional compilation in F4P uses a "D" in column one instead of "X".
7. Adjustable formats have a different syntax (i.e., use of "<" and ">" instead of "N").
8. Some service routines have similar purposes (e.g., ERRSET).
9. Arguments to mathematical routines may have different acceptable values.

The PDP 11/70 is a sixteen-bit ASCII machine using octal notation while the Sigma 9 is a thirty-two-bit EBCDIC machine using hexadecimal notation. Any explicit references to EBCDIC characters or hexadecimal constants will have to be translated to ASCII or octal notation, respectively. The default size of INTEGER and LOGICAL type variables is two bytes on the 11/70 while the only version available on the Sigma 9 is four bytes. The use of the /I4 switch when DEC F4P is invoked will solve this problem. Generally, the use of two-byte variables is preferred, where possible for the sake of speed of operation on the 11/70. Since the Sigma 9 and the 11/70 do not store their data in the same manner, the conversion of some FORTRAN programs may be severely constrained. Appendices A and B discuss this problem in detail. Xerox FORT4 and DEC F4P interpret the ENTRY statement in different fashions (see Appendix C for one example) and its use should be avoided. Large Sigma 9 programs may not fit in the 64K bytes of memory a single user can request on the DCAC 11/70. However, the use of the optimizing F4P compiler, VIRTUAL arrays or an overlay structure may alleviate this difficulty.

CONVERTING DEC FORTRAN TO XEROX FORTRAN

To convert a PDP F4P program to Xerox FORT4, the following F4P constructs will have to be removed or changed to make the code acceptable to FORT4:

1. The PARAMETER statement.
2. Expand INCLUDE statements.
3. Comments after statements.
4. OPEN/CLOSE statements may be simulated by GETPUT.
5. Octal constants.
6. The VIRTUAL statement.
7. Boolean functions may be substituted with FORT4 equivalents.
8. Any "type" specification of less than four bytes.
9. O, $, : and Q format specifications.
10. FORT4 does not use quoted Hollerith strings outside of DATA and FORMAT statements.
11. Conditional compilation in FORT4 uses an "X" in column one instead of "D".
12. Adjustable formats have a different syntax (i.e., use of "N" instead of "<" and ">").
13. Some service routines have similar purposes (e.g., ERRSET).
14. Arguments to mathematical routines may have different acceptable values.

The PDP 11/70 is a sixteen-bit ASCII machine using octal notation while the Sigma 9 is a thirty-two-bit EBCDIC machine using hexadecimal notation. Any explicit references to ASCII characters or octal constants will have to be translated to EBCDIC or hexadecimal notation, respectively. The default size of INTEGER and LOGICAL type variables is two bytes on the 11/70 while the only version available on the Sigma 9 is four bytes. "Type" statements may have to be incorporated to enable the program to run equivalently on the Sigma 9. Since the Sigma 9 and the 11/70 do not store their data in the same manner, the conversion of some FORTRAN programs may be severely constrained. Appendices A and B discuss this problem in detail. Xerox FORT4 and DEC F4P interpret the ENTRY statement in different fashions (see

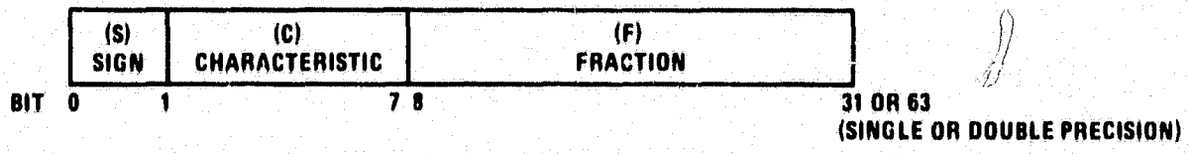
Appendix B for one example) and its use should be avoided. Since the Sigma 9 has more core storage available, any program overlays and virtual arrays used on the PDP could be eliminated on the Sigma 9. The use of the VIRTUAL statement in F4P program could be changed to the use of a DIMENSION statement in a FORT4 program.

APPENDIX A

COMPARISON OF DATA STRUCTURES ON THREE COMPUTERS

The three computers and their FORTRAN compilers discussed in this document implement various data structures (e.g., INTEGER, REAL) in different fashions. These differences become important if FORTRAN programs that are being transferred from one machine to another access binary data (e.g., reading a binary tape, manipulating the bits and bytes of a datum), and can influence the accuracy of calculations (see Appendix D).

Except for negative floating point data the Xerox Sigma 9 and the IBM 360 store their data in the same manner. Both computers store their floating point data in the following fashion:



N_{IBM} = floating point number = $SF \cdot 16^{C-64}$ where $S = \pm 1$

$N_{\Sigma 9} = F \cdot 16^{C-64} \geq 0$ $\quad 0 \leq C \leq 127$

Note: $16^{-6} = 2^{-24} \approx 5.96 \cdot 10^{-8}$

$16^{-14} = 2^{-56} \approx 1.39 \cdot 10^{-17}$

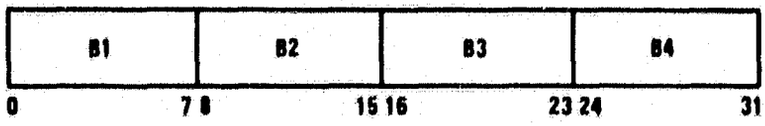
$$F = \begin{cases} 0 \\ 16^{-6} \leq |F| \leq 1 \text{ (single precision)} \\ 16^{-14} \leq |F| \leq 1 \text{ (double precision)} \end{cases}$$

F has six hexadecimal (single) or fourteen hexadecimal (double) digits.

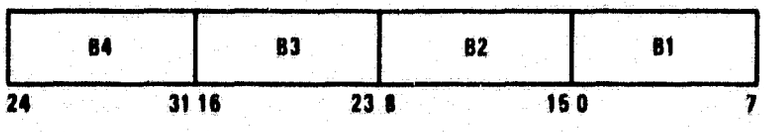
The IBM 360 always stores its fraction, F, as a true fraction. However, this is only the case for positive floating point numbers on the Sigma 9. The Sigma 9 stores a negative floating point number as the two's complement of its positive representation.

The storage of data on the PDP 11 computer differs with the 360 and the Sigma 9 by having the positions of a datum's bytes reversed with respect to how they would be stored on the

360 or the Sigma 9. If the following represents a full word integer stored on the Sigma 9 or the 360:

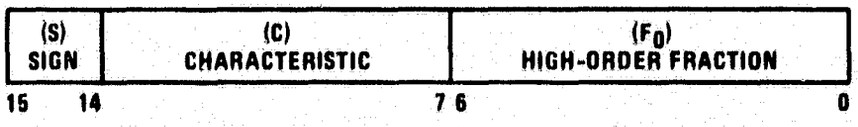


Then it would be stored in the following manner on the PDP 11:

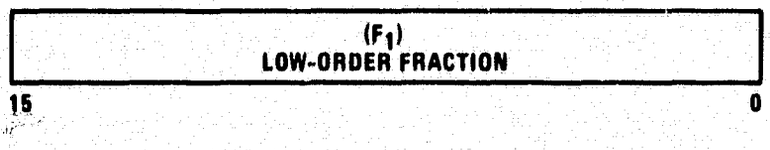


which implies a swap of half-words followed by a swap of bytes within each halfword. To change a floating point number from the 360 or the Sigma 9 to the PDP 11 the bytes within each half word are swapped and then a conversion algorithm must be applied. The conversion is necessary because the PDP 11 uses an eight bit characteristic while the Sigma 9 and 360 use a seven bit characteristic. Some of the implications of this are illustrated in Appendix D. The following will illustrate the form in which floating point numbers are stored on the PDP 11:

word 1:



word 2:



Words 3 and 4 for double precision data would contain lower order fractions (F_2 and F_3) in the same format as word 2.

Thus, $N_{PDP} = SF2^{C-128}$ where $S = \pm 1$, $0 \leq C \leq 255$

$$F = \begin{cases} 2^{-24} \leq |F| \leq 1 & \text{single precision} \\ 2^{-56} \leq |F| \leq 1 & \text{double precision} \end{cases}$$

Fractions are represented in "sign-magnitude" notation with the binary radix point on the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored because it is redundant (i.e., "hidden bit normalization"). The bit is assumed to be 1 unless the exponent is zero (corresponding to 2^{-128}), in which case it is assumed to be zero. As a result of the different floating point architectures on the IBM 360, Xerox Sigma 9 and PDP-11, the conversion of FORTRAN programs among these computers may be constrained. For example:

$$8.636 \cdot 10^{-78} \lesssim |N_{IBM}| = |N_{\Sigma 9}| \lesssim 7.237 \cdot 10^{75}$$

while

$$2.939 \cdot 10^{-39} \lesssim |N_{PDP}| \lesssim 1.701 \cdot 10^{38}.$$

Therefore, programs that use numbers with very large or very small magnitudes on the 360 or Sigma 9 may not be compatible with the PDP 11.

APPENDIX B

IBM 360/PDP 11/70 TAPE COMPATIBILITY

The following material is from Chapter 8 of the Laboratory of High Energy Astrophysics Computer User's Guide (reference 16) and illustrates the byte-swapping of the PDP 11. The routines described below are available on the DCAC PDP 11/70 for the conversion of 360 data to 11/70 data and vice-versa. The routines could be used for the Sigma 9 with an additional conversion being required for negative floating point data.

IBM 360 - PDP-11/70 Tape Compatibility

1. SOURCE Programs

Source tapes generated on IBM machines use the Extended Binary Coded Decimal Interchange Code (EBCDIC) for the representation of characters. DEC machines, however, use American Standard Code for Information Interchange (ASCII) for characters codes. Utility programs are available for easy interchange of information of these 8 bit codes on the 11/70.

2. Transfer of Data Files

2.1 Introduction

There has developed a need for algorithms for converting data on an IBM 360-generated magnetic tape to recognizable PDP-11 format and algorithms for generating IBM-360 magnetic tapes on the PDP-11. This need originates from the difference in byte addressing between the two computers. The problem applies to any INTEGER*2, INTEGER*4, REAL*4, REAL*8 or COMPLEX*8 variable. For a detailed description of the differences, refer to AOIPS Technical Note #75-001, "DEC PDP-11/IBM 360 Magnetic Tape Formats and Information Exchange Considerations."

2.2 Subroutines for PDP-11 and IBM-360 Conversion

(A) TPDPFS - converts an IBM single-precision floating-point quantity to a PDP single-precision floating-point quantity. TPDPFS requires one or two arguments:

Call TPDPFS(INQ[,OUTQ])

(1) INQ – specifies the quantity to be converted.

(2) [,OUTQ] – specifies the destination of the quantity. If omitted, the quantity is returned as a function value.

(B) TPDPFD – converts an IBM double-precision floating-point quantity to a PDP double-precision floating-point quantity. TPDPFD requires one or two arguments:

Call TPDPFD(INQ[,OUTQ])

(1) INQ – specifies the quantity to be converted.

(2) [,OUTQ] – specifies the destination of the converted quantity. If omitted, the quantity is returned as a function value.

(C) TIBMFS – converts a PDP single-precision floating-point quantity to an IBM single-precision floating-point quantity. TIBMFS requires two arguments:

Call TIBMFS(INQ,OUTQ)

(1) INQ – specifies the quantity to be converted.

(2) OUTQ – specifies the destination of the converted quantity.

(D) TIBMFD – converts a PDP double-precision floating-point quantity to an IBM double-precision floating-point quantity. TIBMFD requires two arguments:

Call TIBMFD (INQ,OUTQ)

(1) INQ – specifies the quantity to be converted.

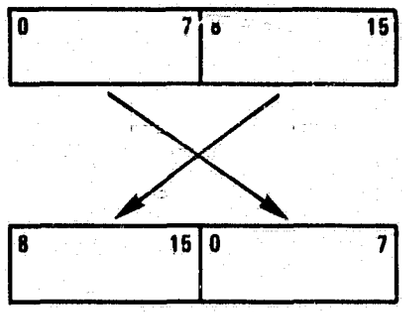
(2) OUTQ – specifies the destination of the converted quantity.

Some of the variables used in the examples which follow are:

(A) BUFF – Address of Data Area
(B) LEN – Length of Block to be Read from Tape
(C) TDAT – Halfword for INTEGER*4 Value
(D) SDAT – REAL*4 Parts of C8DAT

2.3 Algorithms for Conversion - 360 Tape to PDP Format

(A) INTEGER*2



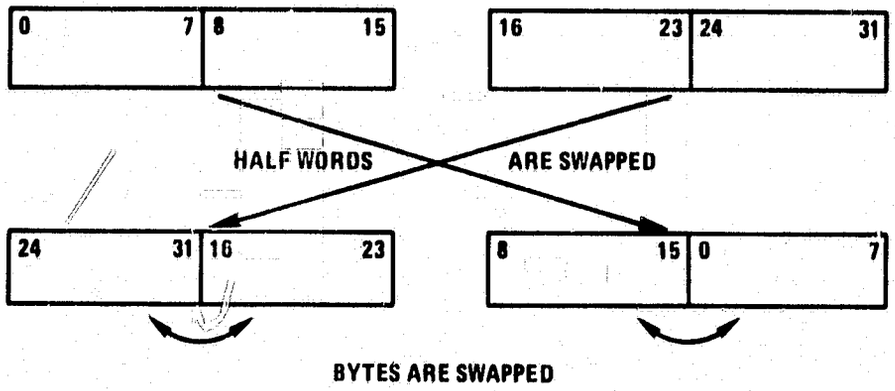
THE BYTES ARE SWAPPED

To retrieve the correct INTEGER*2 data value from a 360-generated mag tape, SWABI is called.

EXAMPLE 1: Assume the first two bytes of BUFF are an IBM 360 INTEGER*2 variable. We wish to convert these bytes to a PDP recognizable INTEGER*2 variable. The result appears in the variable I2DAT as follows:

```
LOGICAL*1  BUFF(100)
INTEGER*2  I2DAT
EQUIVALENCE (BUFF(1),I2DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
CALL FREAD(BUFF,N,LEN,IOST,LR)
CALL SWABI(I2DAT,2)
          .
          .
          .
          .
[CONTINUE PROGRAM]
          .
          .
          .
          .
CALL DISMNT(N,IVSN)
STOP
END
```

(B) INTEGER*4



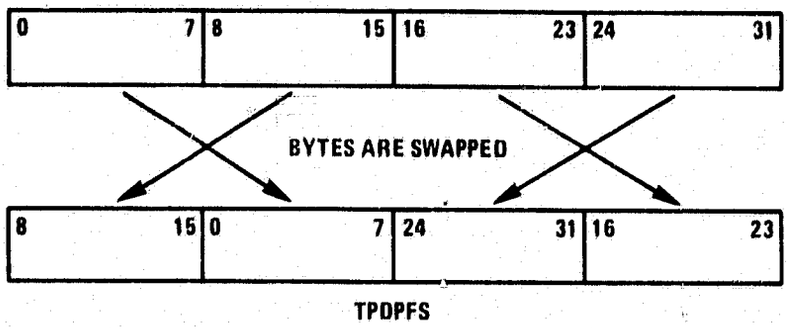
To retrieve the correct INTEGER*4 data value from a 360-generated mag tape, halfwords must be swapped following the call to SWABI.

EXAMPLE 2: Assume the first four bytes of BUFF are an IBM 360 INTEGER*4 variable. We wish to convert these bytes to a PDP recognizable INTEGER*4 variable. The result appears in the variable I4DAT as follows:

```
LOGICAL*1  BUFF(100)
INTEGER*2  TDAT(2),K
INTEGER*4  I4DAT
EQUIVALENCE (BUFF(1),I4DAT),(TDAT(1),I4DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
CALL FREAD(BUFF,N,LEN,IOST,LR)
K=TDAT(1)
TDAT(1)=TDAT(2)
TDAT(2)=K

CALL SWABI(I4DAT,4)
          .
          .
          .
          .
[CONTINUE PROGRAM]
          .
          .
          .
          .
CALL DISMNT(N,IVSN)
STOP
END
```

(C) REAL*4



TPDPFS

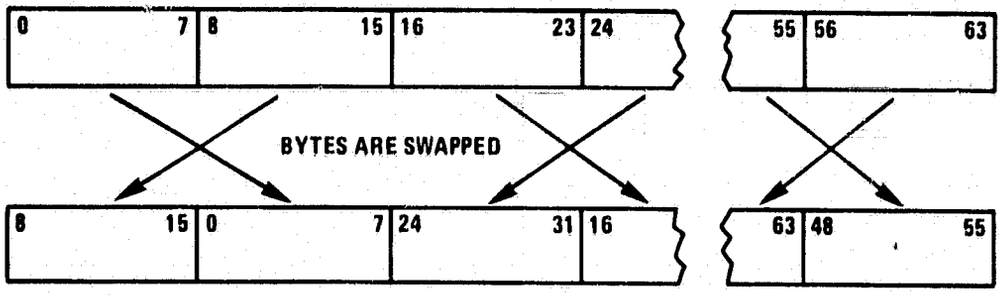
PERFORMS FLOATING POINT CONVERSION

To retrieve the correct REAL*4 data value from a 360-generated mag tape, SWABI is first called, followed by the calling of TPDPFS.

EXAMPLE 3: Assume the first four bytes of BUFF are an IBM 360 REAL*4 variable. We wish to convert these bytes to a PDP recognizable REAL*4 variable. The result appears in the variable R4DAT in the following program:

```
LOGICAL*1  BUFF(100)
REAL*4  R4DAT
EQUIVALENCE (BUFF(1),R4DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
CALL FREAD(BUFF,N,LEN,IOST,LR)
CALL SWABI(R4DAT,4)
CALL TPDPFS(R4DAT,R4DAT)
                .
                .
                .
                .
[CONTINUE PROGRAM]
                .
                .
                .
                .
CALL DISMNT(N,IVSN)
STOP
END
```

(D) REAL*8



TPDPFD

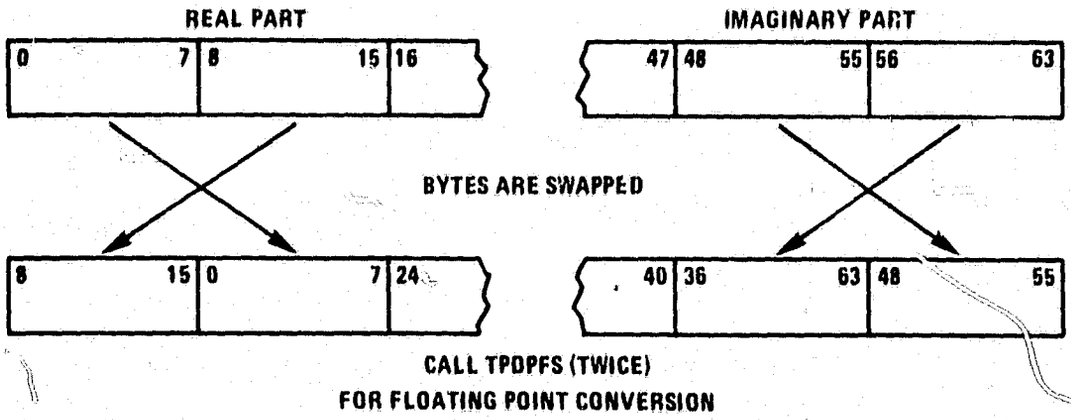
PERFORMS FLOATING POINT CONVERSION

To retrieve the correct REAL*8 data value from a 360-generated mag tape, SWABI is first called, followed by the calling of TPDPFD.

EXAMPLE 4: Assume the first eight bytes of BUFF are an IBM 360 REAL*8 variable. We wish to convert these bytes to a PDP recognizable REAL*8 variable. The result appears in the variable R8DAT in the following program:

```
LOGICAL*1  BUF(100)
REAL*8  R8DAT
EQUIVALENCE (BUFF(1),R8DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
CALL FREAD(BUFF,N,LEN,IOST,LR)
CALL SWABI(R8DAT,8)
CALL TPDPFD(R8DAT,R8DAT)
                .
                .
                .
                .
[CONTINUE PROGRAM]
                .
                .
                .
                .
CALL DISMNT(N,IVSN)
STOP
END
```

(E) COMPLEX*8



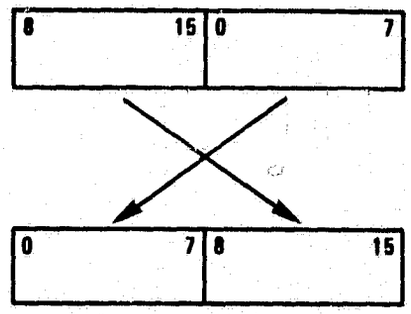
To retrieve the correct COMPLEX*8 data value from a 360-generated mag tape, the whole value is treated as two (2) REAL*4 values. Again, SWABI is first called, followed by the calling of TPDPFS.

EXAMPLE 5: Assume the first eight bytes of BUFF are an IBM 360 COMPLEX*8 variable. We wish to convert these bytes to a PDP recognizable COMPLEX*8 variable. The result appears in the variable C8DAT in the following program:

```
      LOGICAL*1  BUFF(100)
      REAL*4  SDAT(2)
      COMPLEX*8  C8DAT
      EQUIVALENCE  (BUFF(1),C8DAT),(BUFF(1),SDAT(1))
      CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
      CALL FREAD(BUFF,N,LEN,IOST,LR)
      CALL SWABI(SDAT,8)
      CALL TPDPFS(SDAT(1),SDAT(1))
      CALL TPDPFS(SDAT(2),SDAT(2))
C
C     THE COMPLEX*8 VALUE IS NOW CONVERTED AND
C     CAN BE REFERRED TO AS C8DAT
                .
                .
                .
      [CONTINUE PROGRAM]
                .
                .
                .
      CALL DISMNT(N,IVSN)
      STOP
      END
```

2.4 Algorithms for Conversion - PDP-11 to IBM Tape

(A) INTEGER*2



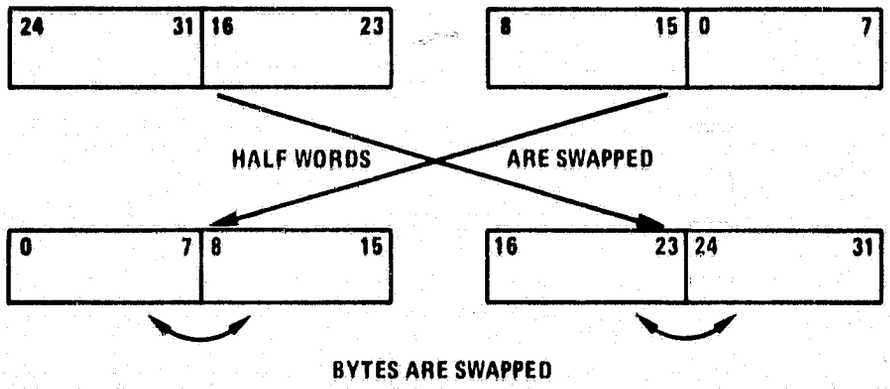
THE BYTES ARE SWAPPED

To generate the correct INTEGER*2 value onto a 360 mag tape, SWABI is called.

EXAMPLE 1: Assume the variable I2DAT is a PDP-11 INTEGER*2 variable which is to be converted to an IBM recognizable INTEGER*2 variable. After converting, the result appears in the variable I2DAT and is then written to a mag tape.

```
LOGICAL*1  BUFF(100)
INTEGER*2  I2DAT
EQUIVALENCE  (BUFF(1),I2DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
          .
          .
          .
          .
[CONTINUE PROGRAM]
          .
          .
          .
          .
CALL SWABI(I2DAT,2)
CALL FWRITE(BUFF,N,LEN,IOST)
CALL DISMNT(N,IVSN)
STOP
END
```

(B) INTEGER*4

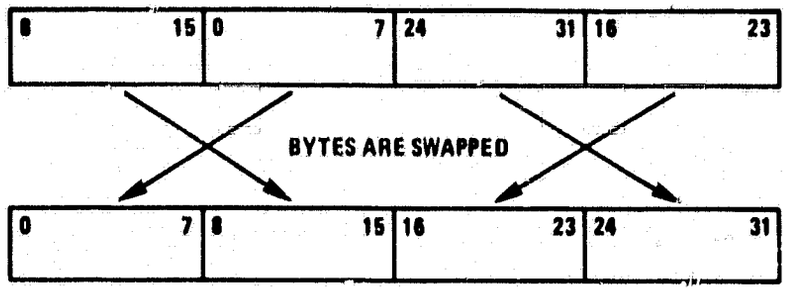


To generate the correct INTEGER*4 value onto a 360 mag tape, halfwords must be swapped following the calling of SWABI.

EXAMPLE 2: Assume the variable I4DAT is a PDP-11 INTEGER*4 variable which is to be converted to an IBM recognizable INTEGER*4 variable. After converting, the result appears in the variable I4DAT and is written to a mag tape.

```
LOGICAL*1  BUFF(100)
INTEGER*2  TDAT(2),K
INTEGER*4  I4DAT
EQUIVALENCE  (BUFF(1),I4DAT),(TDAT(1),I4DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
      .
      .
      .
[CONTINUE PROGRAM]
      .
      .
      .
K=TDAT(1)
TDAT(1)=TDAT(2)
TDAT(2)=K
CALL SWABI(I4DAT,4)
CALL FWRITE(BUFF,N,LEN,IOST)
CALL DISMNT(N,IVSN)
STOP
END
```

(C) REAL*4



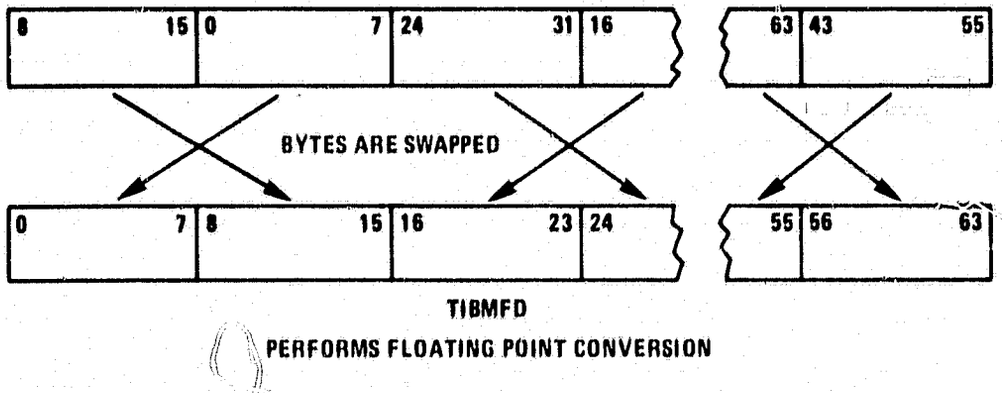
TIBMFS
PERFORMS FLOATING POINT CONVERSION

To generate the correct REAL*4 value onto a 360 mag tape, TIBMFS is first called, followed by a call to SWABI.

EXAMPLE 3: Assume the variable R4DAT is a PDP-11 REAL*4 variable which is to be converted to an IBM recognizable REAL*4 variable. After converting, the result appears in the variable R4DAT and is written to a mag tape.

```
LOGICAL*1  BUFF(100)
REAL*4  R4DAT
EQUIVALENCE  (BUFF(1),R4DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
          .
          .
          .
[CONTINUE PROGRAM]
          .
          .
          .
CALL TIBMFS(R4DAT,R4DAT)
CALL SWABI(R4DAT,4)
CALL FWRITE(BUFF,N,LEN,IOST)
CALL DISMNT(N,IVSN)
STOP
END
```

(D) REAL*8



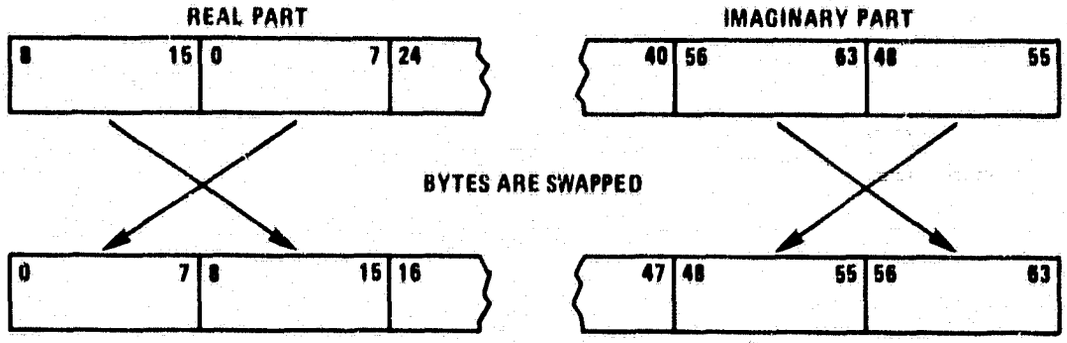
TIBMFD
PERFORMS FLOATING POINT CONVERSION

To generate the correct REAL*8 value onto a 360 mag tape, TIBMFD is first called, followed by a call to SWABI.

EXAMPLE 4: Assume the variable R8DAT is a PDP-11 REAL*8 variable which is to be converted to an IBM recognizable REAL*8 variable. After converting, the result appears in the variable R8DAT and is written to a mag tape.

```
LOGICAL*1  BUFF(100)
REAL*8  R8DAT
EQUIVALENCE  (BUFF(1),R8DAT)
CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
                .
                .
                .
                .
[CONTINUE PROGRAM]
                .
                .
                .
                .
CALL TIBMFD(R8DAT,R8DAT)
CALL SWABI(R8DAT,8)
CALL DISMNT(N,IVSN)
STOP
END
```

(E) COMPLEX*8



CALL TIBMFS (TWICE)
FOR FLOATING POINT CONVERSION

To generate the correct COMPLEX*8 value onto a 360 mag tape, the whole value is treated as two (2) REAL*4 values. Again, TIBMFS is first called followed by a call to SWABI.

EXAMPLE 5: Assume the variable C8DAT is a PDP-11 COMPLEX*8 variable which is to be converted to an IBM recognizable COMPLEX*8 variable. After converting, the result appears in the variable C8DAT and is written to a mag tape.

```
      LOGICAL*1 BUFF(100)
      REAL*4 SDAT(2)
      COMPLEX*8 C8DAT
      EQUIVALENCE (BUFF(1),C8DAT),(BUFF(1),SDAT(1))
      CALL MOUNT(N,IVSN,NF,LABEL,IDEN)
                .
                .
                .
      [CONTINUE PROGRAM]
                .
                .
                .
      CALL SWABI(SDAT,8)
      CALL TIBMFS(SDAT(1),SDAT(1))
      CALL TIBMFS(SDAT(2),SDAT(2))
      CALL FWRITE(BUFF,N,LEN,IOST)
C
C     THE CORRECT COMPLEX*8 VALUE IS
C     NOW WRITTEN ONTO TAPE AS C8DAT
C
      CALL DISMNT(N,IVSN)
      STOP
      END
```

APPENDIX C

FORTRAN FOR THE 11/70 AND THE 360's

Paragraph IV (page 40) of the document in this appendix illustrates one way in which the implementation of the ENTRY statement by the PDP 11/70 F4P compiler differs from the implementation by the Xerox Sigma 9 and IBM 360 FORTRAN compilers. Many DCAC users are familiar with FORTRAN for the 11/70 and the 360's and it is included here for the sake of completeness.

FORTRAN for the 11/70 and the 360's

A guide to the writing of FORTRAN programs compatible with both the DCAC PDP 11/70 and the SACC S/360's

or

Ways to save man-hours in transferring programs between machines

Jim Hamill

John Campbell

Laboratory for Planetary Atmospheres

Data Analysis Branch

June 1978

These guidelines outline FORTRAN programming practices that provide compatibility between two particular Goddard computers: the DCAC PDP 11/70 and either of the SACC S/360's. We emphasize that these practices are easy to follow in the generation of new code.

This memorandum mentions, additionally, a few of the peculiar differences between the two machines. Attention is directed, in particular, to Paragraphs II and IV.

The two compilers compared and contrasted are:

- IBM's FORTRAN-IV level "H"
- DEC's FORTRAN-IV-PLUS

I. We recommend that most programming be in American National Standard FORTRAN (X3.9-1966). There are two important exceptions to this recommendation.

1. We actively encourage the use, where it is easiest for the programmer, of certain commonplace extensions to "ANSI FORTRAN". Table I lists some non-standard features that are common to both vendors' languages. We feel that these non-standard features are of sufficient usefulness and are sufficiently widespread among vendors of software that their use should not be discouraged.

2. In some cases, essential language features are machine-dependent. (A good example is the means by which a given data set is opened for program use. In IBM programming, the Job Control Language (JCL) provides a simple way of doing it, whereas PDP-11 programmers will want to use the OPEN statement.) In other cases, use of a non-standard feature can provide one of the following advantages with respect to the best "standard FORTRAN" alternative:

- appreciable ease of coding;
- significant reduction in I/O (for IBM applications, where "time is big money");
- significant reduction in core requirements (more important on the smaller machine, the 11/70);
- significant reduction in execution time.

If any of these advantages can be realized, or if no sensible alternative is available, we of course encourage the use of the non-standard features.

II. Programmers of the 11/70 must be aware that the default sizes of INTEGER and LOGICAL type variables have been set to two bytes on the DCAC machine, whereas the default

sizes are each four bytes in the IBM FORTRAN languages. We encourage explicit or implicit type declarations. We recommend the following practices in particular.

1. Declare as INTEGER*4 those variables that may at some time exceed 32,767 in absolute value.

2. For character type data, use the LOGICAL*1 declaration wherever possible.

3. Use IMPLICIT statements liberally.

III. Structured programs often consist of numerous subprograms tied together with labeled and/or unlabeled COMMON blocks. It is good practice to use identical COMMON declarations for a given COMMON block wherever it occurs.

This can be a painful and tedious duty for programmers of the IBM machines. The practice is facilitated on the DCAC machine, because the PDP FORTRAN IV-plus language supports the INCLUDE statement. In that language, identical common blocks can be guaranteed by the following simple device: each subroutine has, near its beginning, a statement referring to a file that contains all the relevant COMMON declarations, for instance,

INCLUDE PGM02.INC

The especial advantage of such an approach is that adjustments of COMMON variables (for example, a change in the dimensions of an array) are taken care of by modifications to a single file, in this case the file PGM02.INC.

If any interest is expressed, we can write a utility program to convert PDP FORTRAN (provided that it is sufficiently "standard"!) into a form that should be acceptable to the IBM compilers. In particular, such a utility can replace INCLUDE statements with the fully expanded FORTRAN code.

IV. The ENTRY statement is supported on both machines, but it does not work the same way on each. Example 1 shows the difference, and Example 2 accomplishes the desired effect on either machine.

Example 1. In order to calculate $k = j^{(i)}$, the following subroutine is written:

```
SUBROUTINE SUB1 (I)
RETURN
ENTRY SUB2 (J, K)
K = J**I
RETURN
END
```

(The variable "i" initialized with a call to SUB1, and subsequently entry point SUB2 is used.)

If this subroutine is compiled under the IBM FORTRAN-H compiler and executed, it will give the expected result. The same code run on the PDP will invariably give the result $k = j^{(i)}$. Evidently, it is not sufficient to name a variable in the list of pseudo-parameters.

Example 2. The following code will calculate $k = j^{(i)}$ correctly on either machine:

```
SUBROUTINE SUB1 (II)
I = II
RETURN
ENTRY SUB2 (J, K)
K = J**I
RETURN
END
```

(There are other ways.)

V. The following features are available in IBM FORTRAN-H but not in PDP FORTRAN IV-plus:

- optional returns from a subroutine;
- data initialization in type declaration statements (example:

  INTEGER ZERO / 0 /;)
- COMPLEX*16 type variables.

The first and second deficiencies may be remedied by the programmer. Programs which rely on double-precision complex numbers probably should not be considered for the 11/70. (If double-precision complex arithmetic is restricted to a small portion of the code, a fixup is feasible. See an experienced scientific programmer.)

VI. The PDP FORTRAN-plus language supports the following unusual features. We strongly discourage their use in any GSFC programming!

- expressions as DO parameter;
- expressions in subroutine DIMENSION statements;
- run-time format specifications.

To replace these handy features during initial coding requires only a little extra thought. Experienced programmers are aware of how difficult it can be to replace them once they have been incorporated into a program.

VII. Two features available on the 11/70 are handy and not particularly dangerous in conversion. At the worst, a failure to convert them properly will result in a compiler syntax error. These features are:

- the BYTE statement;
- the same-line commenting feature.

We do not actively encourage the use of these features, but we recognize that they can be convenient.

VIII. Table 2 lists some hierarchies of [IMPLICIT, explicit, COMMON, DIMENSION, DATA] declarations that are acceptable to both the IBM FORTRAN-H and the PDP FORTRAN IV-plus compilers.

Table 1. The below-listed language features are common to both IBM FORTRAN-IV level H and PDP FORTRAN IV-PLUS.

1. Array subscripts may be integer valued expressions. If a floating-point expression has been coded, it is implicitly "fixed".

2. Mixed-mode expressions are fully supported, with implied "floats" as appropriate.

3. The following I/O statements are useful:

   - DEFINE FILE
   - direct access READ (formatted and unformatted)
   - direct access WRITE (formatted and unformatted)
   - FIND

4. The "END =" and "ERR =" options are permitted in I/O statements.

5. The LOGICAL*1 and INTEGER*2 variable types are useful.

6. The IMPLICIT statement is useful.

7. The lengths of variables and of function values may be easily defined, for instance,

```
FUNCTION JFUNC*2 (I)
INTEGER*2 IRAY (10)
```

8. The ENTRY STATEMENT IS SUPPORTED. Please note the caveat in Paragraph IV.

Table 2. The standard declaration statements may not be coded in an arbitrary order. Here we list them in an hierarchical order that is acceptable to both compilers of interest.

1. IMPLICIT declarations.

2. Any of the following declarations in any order:

   - COMMON
   - explicit type declaration
   - DIMENSION
   - EXTERNAL

One may declare the type or the dimension of variables in COMMON, either before or after the corresponding COMMON statement.

3. Data statements.

(Type 1 must precede Type 2 which must precede Type 3.)

APPENDIX D

EFFECT OF FLOATING POINT ARCHITECTURE ON COMPUTATION ACCURACY

This material appeared in the Goddard Weekly Report for May 25, 1979 to May 31, 1979 from Code 570.

Effect of Floating Point Architecture on Computation Accuracy

Recent experience with scientific computations on several computers raised suspicion as to the accuracy of the floating point arithmetic units in the respective machines. Upon investigation it was learned that some of the machines in question truncate the results of a floating point operation, while the others round the results. The overall effects of truncation and rounding on a lengthy computation can have a dramatic impact on the final result. To assess the magnitude of this effect, an analysis was performed and the results are presented here in the hope that others involved in large scientific computations may gain insight into the numerical errors which can be attributed to floating point arithmetic. These results are not to be construed as criticism or praise of any machine, since many more factors must be considered in fully evaluating the numerical accuracy of a particular machine. Instead, the results can and should be used in understanding the differences between truncation and round-off with respect to the accuracy of long sequences of operations.

The analysis reported here addresses, in particular, the IBM S/360 and the DEC PDP 11 computers. With appropriate care, however, the results can be generalized to other computers.

The DEC VAX 11/780 and PDP 11/70 represent double precision floating point numbers in a binary format. Fifty-five bits comprise the fraction, eight bits the exponent, and one bit is used for the sign. Binary normalization is used and arithmetic results are rounded. Numerical precision to sixteen decimal digits is provided in double precision.

The IBM S/360 represents double precision floating point numbers in a hexadecimal format. Fourteen hexadecimal digits (fifty-six bits) comprise the fraction, seven bits the exponent (base 16), and one bit is used for the sign. Hexadecimal normalization is used, and arithmetic results are truncated. Numerical precision to sixteen decimal digits is provided in double precision.

The following graph displays the results of the analysis for these two floating point architectures. On the ordinate is the number of double precision floating point operations performed in sequence on the same set of operands, such that the numerical errors will be compounded. The abscissa then shows on two scales, binary and decimal, the number of digits which will be accurate after a number of operations.

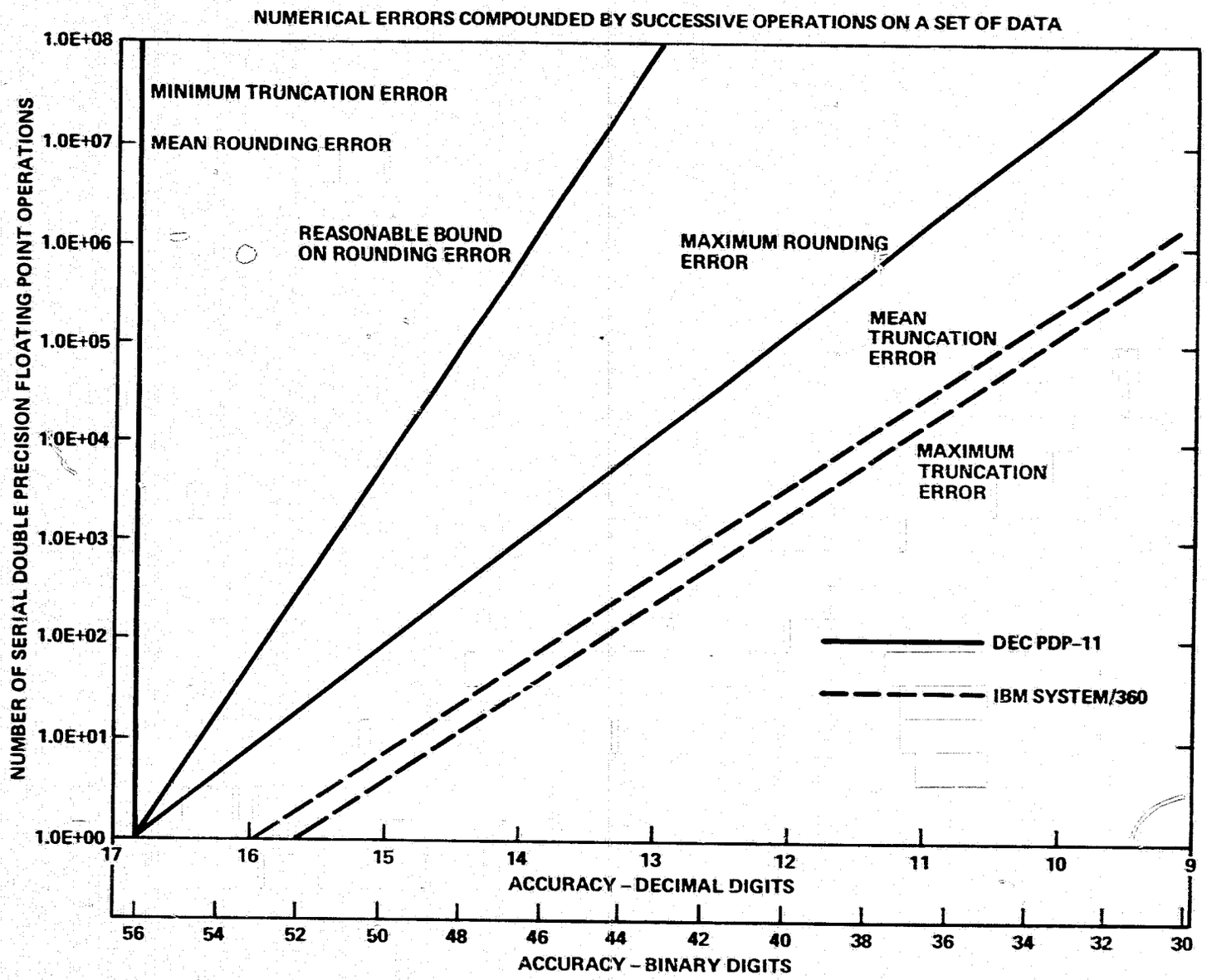
Considering the case of truncation first, it is important to understand that truncation will either leave a result unchanged, or make it smaller; hence, truncation imposes a bias on the result. In the ideal case, where all operations result in leaving the result unchanged after truncation, the accuracy of the result will be the accuracy of the machine (56 bits in the case of the S/360). This is displayed as the minimum truncated error on the graph, the vertical line next to the ordinate. The maximum truncation error line assumes every operation results in truncation of the result, and the mean truncation error is computed assuming a uniform mix of truncation errors throughout the sequence of operations. Note that the mean truncation error is half of the maximum, a point which is easy to miss on a log-log graph such as this.

In the case of round-off, a result can either be left alone, made larger, or made smaller. There exists no bias as is encountered in truncation. Over a long sequence of operations, some results will be rounded up, others rounded down, and still others will remain unchanged. The mean rounding error, therefore, corresponds to the minimum truncation error, as shown on the graph. The maximum rounding error theoretically corresponds precisely to the maximum truncation error. The difference shown on the graph is a result of the binary fraction on the PDP 11 versus the hexadecimal fraction on the S/360. Perhaps the most significant line on the graph is

the one labelled "Reasonable Bound on Rounding Error". This line corresponds to the square root of the maximum rounding error and is derived from the Central Limit Theorem.

Most numerical results on a rounding machine will fall within the envelope between the mean rounding error and the reasonable bound on the rounding error. In contrast to this, most results on a truncating machine will tend to lie closely clustered about the mean, introducing significant errors after long sequences of operations. For example, it is clear from this graph that after a million serial operations on a truncating machine, only about nine digits can confidently be expected to be correct, whereby the same sequence of operations on a rounding machine can be expected to be correct to fourteen decimal places. The conclusion of this analysis is that the numerical architecture of some current computers may limit the achievable accuracy of lengthy scientific computations; in other words, caveat emptor.

For further information on clarification on this concept, please contact Ron Larsen, 344-7777. Anyone engaged in precision computing tasks on the 360 computers should be aware of this effect.



Effect of Floating Point Architecture on Computation Accuracy

REFERENCES

1. IBM System/360 and System/370 FORTRAN IV Language, (IBM: GC28-6515-10), 1974.

2. IBM System/360 Operating System FORTRAN IV (G and H) Programmer's Guide, (IBM: GC28-6817-4), 1973.

3. IBM System/360 OS FORTRAN IV (H Extended Plus) Compiler and Library PRPQ Users Supplement (IBM: SC28-6868-0), 1971.

4. IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide, (IBM: SC28-6852-1), 1972.

5. Xerox Extended FORTRAN IV Language Reference Manual, (Xerox: 90 09 56F), 1975.

6. Xerox Extended FORTRAN IV Operations Reference Manual, (Xerox: 90 11 43E), 1975.

7. Xerox Control Program-Five Reference Manual, (Xerox: 90/7 64 H-1), 1978.

8. FORTRAN IV-Plus User's Guide, (DEC-11-LFPUA-B-D), 1975.

9. FORTRAN Language Reference Manual, (DEC-11-LFLRA-C-D), 1977.

10. "FORTRAN for the 11/70 and the 360's," J. Hamill and J. Campbell, (GSFC Code 626), 1978.

11. ANS Programming Language FORTRAN (ANSI X3.9-1977), 1978 (i.e., FORTRAN 77).

12. "FORTRAN 77", Brainerd (ed.), CACM, V. 21, N. 10, 1978, pp. 806-820.

13. "Effect of Floating Point Architecture on Computation Accuracy", R. Larsen (GSFC-Code 570), Goddard Weekly Report, May 25, 1979-May 31, 1979.

14. Xerox Sigma 9 Computers Reference Manual (Xerox: 90-17-33-1), 1979.

15. IBM System/360 Principles of Operation (IBM: GA22-6821-7), 1968.

16. Laboratory for High Energy Astrophysics (LHEA) PDP 11/70 User's Guide (GSFC T.M. 79584), 1978.

ACKNOWLEDGMENTS

The author wishes to thank Christine Gloeckler, Robert Turkelson, and Narindra Bewtra for their suggestions on the content of and helpful review of this text.

BIBLIOGRAPHIC DATA SHEET

1. Report No. TM 81995	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Comparison of and Conversion Between Different Implementations of the FORTRAN Programming Language		5. Report Date August 1980 6. Performing Organization Code 932
7. Author(s) Lloyd Treinish		8. Performing Organization Report No.
9. Performing Organization Name and Address Interpretive Techniques Branch Information Extraction Division Goddard Space Flight Center Greenbelt, MD 20771		10. Work Unit No. 11. Contract or Grant No. 13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This document is intended to be used as a guideline for computer programmers who may need to exchange FORTRAN programs between several computers. The characteristics of the FORTRAN language available on three different types of computers are outlined, and procedures and other considerations for the transfer of programs from one type of FORTRAN to another are discussed. In addition, the variance of these different FORTRAN's from the FORTRAN 77 standard are discussed.

17. Key Words (Selected by Author(s)) Programming Languages – FORTRAN, Software Portability, Software Conversion	18. Distribution Statement Unclassified – Unlimited		
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages	22. Price*

*For sale by the National Technical Information Service, Springfield, Virginia 22151.

GSFC 25-44 (10/77)