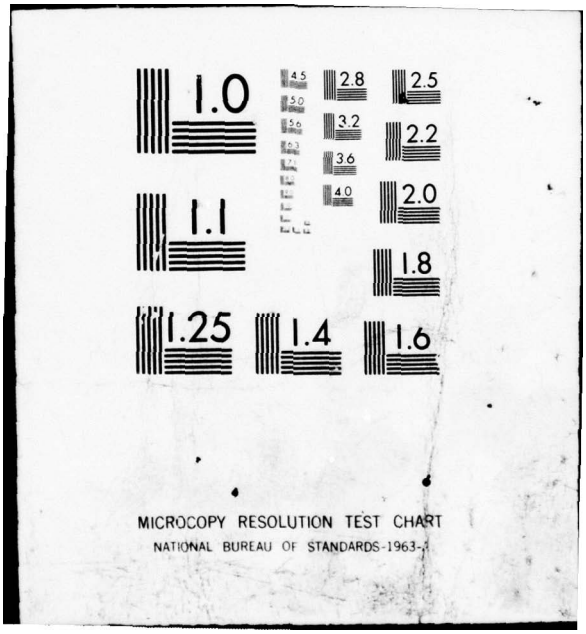
AD-A070 959 STRATEGIC AIR COMMAND OFFUTT AFB NE F/G 9/2
THE INTERFACE OF COBOL AND FORTRAN ON THE HONEYWELL 6080 AND IB--ETC(U)
APR 79 D W LIND, R D GEER, J W WHITE

UNCLASSIFIED NL

1 OF 3
AD A070959


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL

STRATEGIC AIR COMMAND

AD A070959

APTAW ASA-ØØ8

ADWA TECHNICAL REPORT

THE INTERFACE OF COBOL AND FORTRAN ON THE HONEYWELL 6080 AND THE IBM 370/3033 COMPUTER SYSTEMS

1 APRIL 1979

DDC
RECEIVED
JUL 10 1979
C

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

DDC FILE COPY

MAJ LIND
CAPT GEER
LT WHITE
LT McCANLESS
HQ SAC/ADWA
3156

This document has been approved for public release and sale; its distribution is unlimited.

HEADQUARTERS STRATEGIC AIR COMMAND

Offutt Air Force Base, Nebraska

79 07 10 046

LEGAL NOTICE

This final report was prepared by the Technical Analysis Branch, Analysis Division, Directorate of War Plans Programming, Deputy Chief of Staff/Data Systems, Headquarters Strategic Air Command, Offutt Air Force Base, Nebraska under Air Launched Cruise Missile (ALCM) Project task number ASA-ØØ8. Lt James W. White was the project officer.

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation or conveying any rights or permission to manufacture use, or sell any patented invention that may in any way be related thereto.

The report has been authored by employees of the US Government. Accordingly, the US Government retains a nonexclusive, royalty-free license to publish or reproduce the material contained herein, or allow others to do so, for the US Government purposes.

This report has been reviewed by the Information Office (OI) and is releasable to the general public. Distribution is unlimited.

This report has been reviewed and is approved for publication.

Jas W White

JAMES W. WHITE, 1Lt, USAF
Project Officer

Albert H. Jones Jr.

ALBERT H. JONES, JR., Major, USAF
Chief, Technical Analysis Branch

D.L. EVANS, Brigadier General, USAF
Deputy Chief of Staff, Data Systems

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: APTAW ASA-ØØ8	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Interface of COBOL and FORTRAN on the Honeywell 6080 and IBM 370/3033 Computer Systems	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Major David W. Lind; Capt Roger D. Geer; 1Lt James W. White; 2Lt Donald P. McCanless	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: HQ SAC/ADWA, Offutt AFB, NE 68113	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: DAR #A77Ø68
11. CONTROLLING OFFICE NAME AND ADDRESS: HQ SAC/ADWA, Offutt AFB, NE 68113	12. REPORT DATE: 1 April 1979
	13. NUMBER OF PAGES: 113
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: Distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer language, COBOL, FORTRAN, interface, bit, byte, call subroutine, IBM, Honeywell, linkage, data

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the research performed in the interface of the COBOL and FORTRAN programming languages on the IBM 370/3033 and the Honeywell 6080 computers. It identifies specific problem areas in the linkage of the two languages and the transference of programs between machines. The report also includes solutions and alternatives to most difficulties encountered and recommendations for successful language interface and program transfer from one computer to the other.

335 250 JOB

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACKNOWLEDGEMENT

"Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention 'COBOL' in acknowledgement of the source, but need not quote this entire section.

"COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

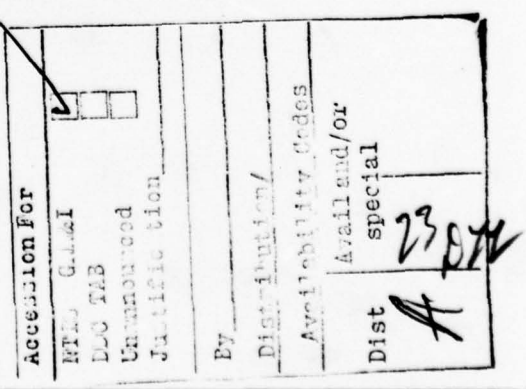
"No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

"Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

"The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications."



PREFACE

The authors wish to thank the following individuals for their support:

For technical assistance

Major Bruce Bauer/ADWA, Major Albert Jones/ADWA, Capt James Jack/ADDP, Capt Marjorie Torsiello/ADWA, Lt Richard Magnan/ADWN, MSgt Dan Myers/ADDP, Mr Robert Chaney, GS-12/ADDP, and Mr Harold Tornquist, GS-12/ADXR

whose comments, assistance, and guidance contributed significantly not only to the debugging of the programs herein, but also to the range and overall quality of this technical report.

For typing support

SSgt Jerry A. Byrd/ADWA and Ms Elizabeth Richards

whose patience allowed the authors to continue making changes in wording and phraseology, and whose speed and accuracy helped to produce a quality product in a timely manner.

Reference to a company or product name does not imply approval or recommendation of the product by the US government to the exclusion of others that may be suitable.

No warranties expressed or implied are made by the government that any program or methodology in this report is free of error. If any such program or methodology is used as the sole basis to solve a problem whose incorrect solution could result in destruction of property or injury of personnel, the user assumes all risk and the government disclaims all liability for such misuse.

ABSTRACT

This report describes the research performed in the interface of the COBOL and FORTRAN programming languages on the IBM 370/3033 and the Honeywell 6080 computers. It identifies specific problem areas in the linkage of the two languages and the transference of programs between machines. The report also includes solutions and alternatives to most difficulties encountered and recommendations for successful language interface and program transfer from one computer to the other.

TABLE OF CONTENTS

	Page
Title Page	i
Abstract	ii
Table of Contents	iii
List of Figures	v
Section 1. Introduction	
1.1 Background	1-1
1.2 Purpose	1-1
1.3 Motivation	1-1
1.4 Programming Languages	1-1
1.5 Computer Hardware	1-2
1.6 Assembler Language	1-2
1.7 Documentation	1-2
1.8 The Linkage or Interface Problem	1-3
1.9 The Report Contents	1-3
Section 2. Conclusions and Recommendations	
2.1 Purpose and Scope	2-1
2.2 Conclusions	2-1
2.3 Recommendations	2-2
Section 3. COBOL and FORTRAN	
3.1 Introduction	3-1
3.2 COBOL	3-1
3.3 FORTRAN	3-3
Section 4. The Computers	
4.1 Introduction	4-1
4.2 The Bit	4-1
4.3 The Byte	4-1
4.4 The Word	4-6
4.5 Addressing	4-7
4.6 Subprogram Addressing and Linkage	4-7
4.7 Types of Data and Data Addressing	4-8
4.8 Implications of Data Structure	4-13
4.9 Implications of Data Alignment and Addressing	4-13
4.10 File Access Methods	4-16
Section 5. Data Representation	
5.1 Introduction	5-1
5.2 The IBM Computer	5-1
5.3 The Honeywell Computer	5-13
5.4 Honeywell FORTRAN/COBOL	5-25
5.5 IBM FORTRAN/COBOL	5-25
5.6 IBM/Honeywell Equivalents	5-25

		Page
Section 6. Linkage		
6.1	Introduction	6-1
6.2	Honeywell COBOL Program Linkage	6-1
6.3	IBM COBOL Program Linkage	6-4
6.4	FORTRAN Linkage	6-8
6.5	Honeywell COBOL/FORTRAN	6-12
6.6	IBM COBOL/FORTRAN Linkage	6-17
Section 7. Job Control Language		
7.1	Introduction	7-1
7.2	IBM JCL	7-1
7.3	Honeywell JCL	7-3
Section 8. Additional Techniques		
8.1	Introduction	8-1
8.2	Passing a Table from COBOL to FORTRAN	8-1
8.3	Double Word Integers	8-2
8.4	Reversing the Double Word Process	8-2
8.5	Packed Decimals	8-3
8.6	Double Precision Floating Point Numbers	8-3
8.7	Passing Integers Through Real Variables	8-3
8.8	Comparisons	8-3
8.9	An Illustration	8-4
8.10	COBOL PROGRAM-ID Name Selection	8-11
8.11	Passing COBOL Group Items	8-13
8.12	The STUB Program	8-14
ATTACHMENTS		
Attachment 1		A-1
Attachment 2		A-15
Attachment 3		A-29
Attachment 4		A-35
Attachment 5		A-42
Attachment 6		A-49
Attachment 7		A-58
Attachment 8		A-65
Attachment 9		A-73
Attachment 10		A-85
Attachment 11		A-90
Glossary		G-1
Bibliography		B-1

List of Figures

Figure		Page
3-1	Basic Definitions of COBOL Words (Verbs)	3-8
3-2	Basic Definitions of FORTRAN Statements	3-10
4-1	Natural Binary Coded Decimal System	4-2
4-2	Character Sets	4-3
4-3	Ranges of Fractional Fixed Point Representations	4-10
4-4	Ranges of Integral Fixed Point Representations	4-10
4-5	Ranges of Floating Point Numbers	4-11
4-6	Honeywell 6000 Series Data Alignment	4-14
4-7	IBM 360/370 Series Data Alignment	4-15
4-8	Honeywell Series 60 Level 66 File Structure	4-18
4-9	IBM 370 File Structure	4-18
4-10	File Access Method Relationships	4-19
4-11	File Access Methods and Language Relationships	4-20
5-1	Honeywell COBOL to FORTRAN Data Types	5-27
5-2	Honeywell FORTRAN to COBOL Data Types	5-28
5-3	IBM COBOL to FORTRAN Data Types	5-29
5-4	IBM FORTRAN to COBOL Data Types	5-30
5-5	Honeywell to IBM COBOL	5-31
5-6	IBM to Honeywell COBOL	5-32
5-7	Honeywell to IBM FORTRAN	5-33
5-8	IBM to Honeywell FORTRAN	5-34
6-1	ENTRY/EXIT Statements	6-5
6-2	Compiler View of a Floating Point Number	6-14
6-3	Actual View of Any Word	6-14
6-4	Output Without DECODE Statement	6-18

SECTION 1. INTRODUCTION

1.1 Background. The computer is a very useful machine. It can be used to process a large amount of data or perform an enormous number of computations in a very short period of time. Unfortunately, the utility of a computer is limited by one's inability to easily and correctly instruct the machine in the task to be performed and efficiently transfer data to and from the machine. To help resolve this limitation, a number of computer "languages" have been devised. Some of these languages work well for data processing requirements, but have severe limitations when used to perform complex computations; hence, computer languages have also been developed for these complex computations, but these languages have serious limitations when used for data processing. Attempts have been made to write languages that strike a balance between data processing and computational usages; however, computer programmers are most familiar with language limitations in either data processing or computational applications. Therefore, it often becomes necessary to use more than one language in the same computer program. This application generates problems involving interfacing or "linking" the modules of one language to the modules of another.

1.2 Purpose. The purpose of this report is to examine these problems as they occur on the Honeywell 6080 and IBM 370/3033 computers and propose solutions. The following paragraphs describe some of the problems in detail.

1.3 Motivation. The Air Launched Cruise Missile (ALCM) strategic planning will be accomplished at Headquarters, Strategic Air Command (HQ SAC) on the TRIAD COMputer System (TRICOMS) using the IBM 370/3033 computer. Some contractor-produced code will be written in the FORmula TRANslation (FORTRAN) language and will require interfacing with Common Business Oriented Language (COBOL) programs. Many of these programs will be written by HQ SAC computer programmers. In addition, some programs will be transferred between the Honeywell and IBM machines. Information from various documents and the experience of HQ SAC programmers was gathered and documented in this report to facilitate the writing of these interfaces and the transference of programs for the ALCM project.

1.4 Programming Languages. Two higher level languages will be described in this report: COBOL and FORTRAN. These two languages have been approved by the Joint Cruise Missile Project Office (JCMPO) for use in the ALCM Mission Planning System. COBOL is useful for manipulating the data but has serious shortcomings when used for sophisticated computations. Hence, the computer programs that will perform sophisticated computations are normally written in FORTRAN. Unfortunately, FORTRAN does not manipualte data records and files very well; therefore, these two languages are used together to take advantage of their

unique capabilities. The problem of passing information from one language to another can be difficult.

1.5 Computer Hardware. It is possible to give instructions and data to a computer (i.e., program it) without any knowledge of the electronic systems that do the computations or processing. Of course, the more one knows about the hardware (electro/mechanical systems) of a computer, the more one is able to do with the computer. The higher level languages usually require no knowledge of the computer hardware. This attribute of a higher level language makes possible the direct transfer of a program from one computer to another although the hardware may be quite different. Theoretically, this attribute makes programming for different machines easy if standard higher level languages are used. In practice, however, several problems arise. Different machines represent data differently. For example, the IBM 370/3033 represents a real "floating point" number with 32 bits where the Honeywell 6080 uses 36 bits for the same number. Hence, a direct transfer of data from an IBM produced data record to a Honeywell machine may not be possible. This type of situation exists on the same machine between different languages. For example, COBOL on the IBM machine may produce data that cannot be interpreted by a FORTRAN program on the same machine. In addition to this problem, a higher level language may be able to link together its own subroutines, but not link up with the subroutines produced by another higher level language on the same machine. For example, a COBOL subroutine may not be able to use a FORTRAN subroutine in the same way it uses another COBOL subprogram. Finally, data sets generated by one higher level language may not be compatible with another higher level language. For example, a FORTRAN program may not be able to read a COBOL produced file. All these problems are discussed in this report.

1.6 Assembler Language. A word should be said about assembler languages. These special languages are peculiar to the specific machines for which they are designed. For example, the Honeywell 6080 assembler is called GMAP and the IBM 370/3033 assembler is BAL. Neither will work on the other machine because they are hardware oriented; hence, the use of these languages prohibits the direct transfer of a program from one machine to the other and presents further linkage problems when used with COBOL or FORTRAN. Normally, the use of assembler languages can and should be avoided; hence, this report will not discuss assembler linkage to higher level languages in detail.

1.7 Documentation. The typical computer reference library has numerous volumes that discuss almost every aspect of computer software and hardware. Unfortunately, most documentation is confined to the discussion of one higher level language or the requirements of a "loader" with respect to linking software produced by several higher level languages at the machine

language level (lowest level). However, little is said about the interfacing of higher level languages at the higher level. No document is known to the authors that compares the Honeywell 6080 to the IBM 370/3033 in sufficient detail to permit a transference of mixed higher order languages between the machines. This report is intended to fill many of these voids.

1.8 The Linkage or Interface Problem. In this report, subroutine linkage will mean all the factors that need to be examined to properly pass information from one subroutine to another. Each higher level language has its own way of accomplishing interfaces between its subprograms. Unfortunately, these techniques are not always common to different higher level languages, even on the same machine. For example, the COBOL program may call a subprogram with the code:

```
CALL BLEEP USING ZONK, BLOOP, UP-DATE.
```

The equivalent FORTRAN code is

```
CALL BLEEP (ZONK, BLOOP, UPDATE)
```

Note that "UP-DATE" is not permitted in FORTRAN. Also FORTRAN assumes that "UPDATE" is a real number of one computer word length (32 bits in the IBM computer and 36 bits in the Honeywell computer). In the COBOL program UP-DATE could be any kind of data, even a type that FORTRAN cannot read. However, since the variables "ZONK, BLOOP and UP-DATE or UPDATE" represent memory locations, the computer will attempt an interface. Hence, if "BLEEP" is a FORTRAN subroutine being called by a COBOL program, UP-DATE will be interpreted by the FORTRAN routine as a real number (unless told to do otherwise) and will be used in this way even though the COBOL program considers it a completely different type of datum. Since the computer will seldom detect a linkage problem of this type, the results of the data manipulation or computation may be grossly in error. Even worse, the FORTRAN program could expect an array or table at UPDATE and read more data than the COBOL program intended. Another interesting problem arises when interfacing COBOL and FORTRAN arrays. The COBOL and FORTRAN subscripting conventions are different. COBOL stores arrays in row order; i.e., the right most subscript varies most rapidly. FORTRAN stores arrays in column order; i.e., the left most subscript varies most rapidly.

1.9 The Report Contents. Section 2 of this report summarizes the conclusions and recommendations. The two higher level languages, COBOL and FORTRAN are described and compared in Section 3. The Honeywell 6080 and IBM 370/3033 are described in Section 4. Data representation on both machines and in the two languages is described in Section 5. Section 6 goes into the detail of

linking subprograms to one another. The Job Control Language requirements pertaining to the use of more than one higher level language are discussed in Section 7. Finally, Section 8 describes some "tricks of the trade" to get around some of the stickier problems of subprogram linkage.

SECTION 2. CONCLUSIONS AND RECOMMENDATIONS

2.1 Purpose and Scope. This section summarizes the conclusions of the research accomplished concerning the linkage of COBOL and FORTRAN computer programs. No research of this character can exhaust the possible problems associated with this linkage nor cover all the solutions to these problems. The research was performed in sufficient depth to make recommendations that will insure that problems with linkage will be minimized or at least identified. These recommendations are included in this section after a discussion of the conclusions.

2.2 Conclusions. Serious problems may occur when attempting to link a COBOL produced program and a FORTRAN produced program. The computer will seldom detect these problems, and the program may not process the data as the programmer intended. The greatest problem occurs in the transference of data from a program created by one compiler to a program created by another compiler. Conclusions and report references are listed below:

a. COBOL programs and FORTRAN programs can be linked together. Difficulties may arise when passing data between them. Insuring that data descriptions are the same in both languages is imperative. (1.8, 4.6, 6.5, 6.6.1)

b. Most types of data can be passed between COBOL and FORTRAN with little or no trouble. Other data types require special handling either in the driver or in the subroutine. (4.7, Fig 5-1 thru Fig 5-4)

c. Group items may be passed from a COBOL program to a FORTRAN subroutine on a limited basis, as they must be read into a FORTRAN array and are subject to the requirements thereof. (5.2.3.4.4, 5.3.3.4.4)

d. (Honeywell) When more than one program is compiled for execution, the $ ENTRY card in the Job Control Language insures control enters the correct program as the driver or main program. (7.3.3)

e. When testing and debugging programs, using selected options in the Job Control Language can be of great help in locating errors. (7.3)

f. (Honeywell) No way was found to pass a file code between a COBOL program and a FORTRAN program using the CALL statement. The file must be created or identified in the COBOL program and accessed by the FORTRAN program using a $ FILE card for each language and the same LUD (Logical Unit Designator). (7.3.8)

g. When interfacing COBOL and FORTRAN programs, output files must be closed when transferring control to a subroutine or back to the driver. (Attachment 8)

h. COBOL PROGRAM-IDs which contain more than 6 characters may cause problems when called from FORTRAN. (Section 8)

i. IBM COBOL requires the use of a LINKAGE SECTION in a called COBOL program. (6.3.2.1)

2.3 Recommendations. The linking of programs written in different languages for arbitrary reasons is to be avoided. However, if linking programs of different languages cannot be avoided, the following practices are recommended to insure successful linkage:

a. All COBOL computational items should be synchronized. (4.7g, 5.2.1.2)

b. Programmers should carefully study the linkage descriptions in applicable technical manuals before linking programs produced by different compilers. (Section 6)

c. In COBOL data descriptions, FILLER strings should be used to align items even when the compiler would insert them. (5.2.3.4.1, 5.3.3.4.1)

d. In COBOL, Ø1 level items should be used in WORKING-STORAGE in place of 77 level items. (5.2.3.3.1, 5.3.3.3.1)

e. Use the "QUOTE" parameter in IBM COBOL JCL to maintain ANSI standards.

f. Avoid the use of double word integers in COBOL. (Section 8)

g. Never pass packed decimals to a FORTRAN routine. (4.7b, Figures 5-1, 5-3)

h. Avoid passing double word integers to FORTRAN routines. (Section 4 and 8)

i. When passing data items from COBOL to FORTRAN, always pass them to items of equivalent FORTRAN descriptions. (Fig 5-1 thru Fig 5-4)

j. Start separate display or character items on word boundaries. (5.2.3.4.2, 5.3.3.4.2)

k. COBOL tables should be passed to FORTRAN using the table name only if the items are homogeneous and synchronized. Otherwise pass the data item by item. (5.2.3.4.4, 5.3.3.4.4)

l. Programmers should insure that items to be compared are justified properly within the computer words. (4.7, 5.2.3.3, 8.8)

m. Always use a PROGRAM-ID which contains 6 or fewer characters. (Section 8)

n. (Honeywell) Always use the $ ENTRY card when executing more than one program at a time. (7.3.3)

o. Never use DISPLAY for items used in arithmetic computations. (5.2.1.7, 5.3.1.6)

p. Never pass COBOL numeric data in DISPLAY format to a FORTRAN routine when the FORTRAN routine assumes the data is not in character format. (Figures 5-1, 5-3)

q. Always use available JCL features to assist testing and debugging. (Section 7)

r. (IBM ONLY) Use FORTRAN JCL DDNAME in COBOL to refer to the same file. (Attachment 4)

s. (Honeywell) Do all writing in COBOL driver to avoid carriage control problems in FORTRAN. (6.5.4)

t. Reverse subscripts of COBOL table elements when using the table in a FORTRAN subroutine. (8.2)

u. Write a "stub" program with simple data to test and insure proper linkage before attempting to link more sophisticated code and data (8.12)

SECTION 3. COBOL and FORTRAN

3.1 Introduction. The ALCM planning system will use the COBOL and FORTRAN computer languages. This section describes and compares the basic features of these languages to establish the relationships between the features of different languages that are intended to perform similar tasks. The formats of these languages and the words the computers recognize as instructions are discussed. Data representation is discussed in Section 5.

3.2 COBOL. The COmmon Business Oriented Language (COBOL) programming language was introduced in 1960 for business data processing purposes. It has become one of the most widely used languages resisting attempts to replace it with more general languages. (19.1173 - 1174)

3.2.1 The COBOL Divisions. A COBOL program consists of four divisions: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE.

a. The IDENTIFICATION DIVISION gives a name to the program. The program name is the primary symbolic reference for the program; that is, it is the symbol for the starting address for the program instructions. The IDENTIFICATION DIVISION may contain other information, but only the program name is recognized by the machine and this name is vital to linking the COBOL program to any other program.

b. The ENVIRONMENT DIVISION consists of two sections: the CONFIGURATION SECTION and INPUT-OUTPUT SECTION. The CONFIGURATION SECTION describes the computer hardware configuration and special names assigned to hardware by the programmer. The paragraphs of the CONFIGURATION SECTION called SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES should not affect linkage since they normally affect only the COBOL compilation. The FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION identifies files with peripheral devices. This information must correspond to file codes (Honeywell) or data names (IBM) in the Job Control Language (JCL). For example, the Honeywell COBOL statement

SELECT FILE1 ASSIGN TO AB, ACCESS IS RANDOM.

assigns a random access file called FILE1 to a device identified in the JCL as AB. The equivalent IBM statement is

SELECT FILE1 ASSIGN TO DA-D-AB, ACCESS IS RANDOM.

Note that the IBM sentence, contains the extra characters DA-D-AB for the class of device, which in this case is mass storage. The single character D is the organization of the file. D implies direct organization as opposed to sequential. The Honeywell class and organization is determined by the JCL. As will become apparent in Section 7, this information is important for linkage purposes. The I-O-CONTROL paragraph describes special file techniques that will normally not affect other subprograms, but may contain information about the location of files on tapes, and information about file label and record conventions that may be useful in establishing linkage with other subprograms.

c. The DATA DIVISION of a COBOL program establishes the form of the data that each variable represents. It consists of three sections: FILE, WORKING-STORAGE, and (OPTIONAL) REPORT SECTION. Also when data must be passed from one program to another program a LINKAGE-SECTION (IBM) must be coded in the program receiving the data. The FILE SECTION contains file descriptions. The WORKING-STORAGE SECTION contains data descriptions of the variables used internally to the program. Hence, the FILE SECTION links the computer memory to external media and the WORKING-STORAGE SECTION describes the representation of the data within memory. This configuration permits rapid transfer of data from the external media to memory. The conversion of the data to workable form, as defined in the WORKING-STORAGE SECTION, may be accomplished when required. The knowledge of data format is essential to correct subprogram linkage. As will become more apparent with the discussion of FORTRAN, the COBOL means of transferring data to and from external devices is superior to the FORTRAN method. The REPORT SECTION is a COBOL peculiar feature which should never involve linkage. Hence, it will not be discussed. Data representation is discussed in detail in Section 5.

d. The PROCEDURE DIVISION contains all the algorithms and steps to do the task required of the program. It is divided into sections called paragraphs which consist of sentences. These sentences are terminated by a period and a space. The PROCEDURE DIVISION is translated by the compiler into assembler and machine language instructions.

3.2.2 COBOL Verbs. COBOL sentences consist of combinations of words and symbols and begin with a verb. COBOL verbs are words that direct an action; e.g., READ or COMPUTE. They are from a group of words called "reserved words". These reserved words cannot be used for any other purpose than their intended use. FORTRAN has no reserved words. For example, WRITE can be a verb and variable in the same program in FORTRAN. The statement

```
WRITE (Ø6,1Ø) WRITE
```

is perfectly legitimate in FORTRAN, but

```
WRITE RECORD-ONE FROM WRITE
```

would not be an acceptable statement in COBOL. COBOL verbs are normally associated with the PROCEDURE DIVISION, but the COPY verb may be used in the ENVIRONMENT DIVISION and the DATA DIVISION. A list of COBOL verbs and a brief description of their use may be found in Figure 3-1. If an * precedes a verb, the verb is peculiar to Honeywell COBOL; a # indicates that the verb is peculiar to IBM COBOL.

3.2.3 Advantages/Disadvantages of COBOL. COBOL has several noteworthy advantages over most other languages. Data may be rapidly transferred from files to core. The COBOL sentence structure makes the code easy to read and understand, and it is somewhat self documenting. A COBOL program can be written and tested rapidly. Finally, the structure of the language and the compiler prevents many programming errors. Unfortunately, it is not easily adapted for scientific or mathematical uses and its structure is somewhat lengthy and cumbersome. (19: 1116-1119)

3.3 FORTRAN. The FORmula TRANslation (FORTRAN) computer language was introduced in 1956 for scientific and engineering purposes. Unlike COBOL, FORTRAN programs are not divided into divisions and FORTRAN does not have reserved words. Each line of code represents one FORTRAN statement. The Honeywell 6080 computer and some other compilers permit more than one statement per line of code. The IBM FORTRAN-G compiler does not. (19: 1173)

3.3.1 Program Identification. FORTRAN subprograms are identified by subroutine name, function name, or default to "MAIN" program. For example, the statement

```
SUBROUTINE BLEEP (ZONK, BLOOP, UPDATE)
```

identifies a subprogram called "BLEEP".

```
FUNCTION BLEEP (ZONK, BLOOP, UPDATE)
```

also identifies a subprogram called "BLEEP". If a subprogram

begins with any other statement, the subprogram is identified as a "MAIN" program. These program identifiers represent the starting address of the subprograms and, therefore, are important for linkage. All ANSI FORTRAN program identifiers must be six or less characters and must begin with an alphabetic character.

3.3.2 Data Definition. FORTRAN identifies or 'types' data elements as integers, real numbers, complex numbers or logical variables. Honeywell FORTRAN and some other compilers also identify character data. The equivalences between these FORTRAN data elements and COBOL data elements are described in Section 5. All FORTRAN compilers assume that any variable beginning with the characters I, J, K, L, M or N is an integer item. Any other variable is assumed to be a real number. Complex, logical, and character variables must be defined by a special statement. For example, the statement

```
LOGICAL CHECK
```

defines "CHECK" to be a logical variable, that is "CHECK" has the value TRUE or FALSE. Note that FORTRAN does not require data items to be typed as integer or real, but in the absence of explicit definitions, FORTRAN will default to the assumptions described above for a variable.

3.3.3 Data Transfer from Files. The transfer of data from a file and its conversion to a workable form is accomplished at the same time by a FORTRAN program. COBOL normally reads a file into core as is and makes data conversions only when necessary in the course of doing computations or making comparisons. Hence, FORTRAN data transfer tends to be slower than COBOL transfer. There are exceptions to this assertion. For example, the general FORTRAN statement

```
READ Ø7 FILE
```

is very fast, but usually requires "decoding" which is a very slow operation. The standard FORTRAN read statement is of the form

```
READ (Ø5, 1ØØ) DATA1, DATA2, DATA3,
```

where Ø5 is the file code; DATA1, DATA2, and DATA3 are data variables and 1ØØ is the statement number of a "FORMAT" statement. The FORTRAN format statement contains the information that would be found in the picture clauses of the COBOL DATA DIVISION. For example,

```
1ØØ FORMAT (A3, 1X, I4, 2X, F8.2)
```

describes a file record consisting of a datum of three alphanumeric characters (A3), one space (1X), a datum of four

integers (I4), two spaces (2X) and a real floating point number occupying 8 bytes, of which the last 2 bytes are decimals.

If the file were a card deck a typical card would be

```
NUBØØ123ØØ-1234567
```

which would assign the following values in the read statement above

```
DATA1 = "NUB"
DATA2 = 123
DATA3 = -12345.67
```

Unless these variables were specified to be specific lengths, they would each default to one word in length. Hence, all the data would probably require conversion when the READ statement is executed. Unlike the COBOL READ verb, the FORTRAN READ verb will open the file, read it and make conversions without the need of any statement other than a FORMAT statement.

3.3.4 Data Transfer Between Subprograms. Data may be transferred from one subprogram to another through the CALL or "COMMON BLOCKs". A typical call is:

```
CALL BLEEP (X,Y,Z)
```

The data are transferred through the variables X, Y, and Z. The order of these variables is important. The subroutine definition:

```
SUBROUTINE BLEEP (ZONK, BLOOP, UPDATE)
```

equates X to ZONK, Y to BLOOP, and Z to UPDATE. In other words, the address of X becomes the address of ZONK, etc. Hence, data may be transferred in and out of a subprogram through a call variable as with COBOL. The COMMON BLOCK can be "labeled common" and is established by a statement like:

```
COMMON/LABEL/DATA1, DATA2
```

Unlabeled common may be established by a statement like:

```
COMMON DATA1, DATA2
```

Any FORTRAN subprogram containing common statements can transfer data through these common areas to any other subprogram containing these statements. This feature of FORTRAN is very powerful. Unfortunately, COBOL cannot access common. Hence, labeled or unlabeled common are not good candidates for transferring data to non-FORTRAN programs.

3.3.5 Executable and Compiler Instruction Statements. There are basically two types of FORTRAN statements: executable and instructions to the compiler. Instructions to the compiler are statements like COMMON, LOGICAL, and SUBROUTINE. These compiler instructions normally must come before any executable statements in the subprogram. FORMAT statements are an exception in that they may appear anywhere in the subprogram. Executable statements like CALL, READ and WRITE appear in the order in which they are to be executed. Computations are written as assignment statements in equation form. For example,

```
Y = X*(Z**2)
```

where Z is squared, then multiplied by X, and the result stored in Y. This statement is similar to that which follows the COBOL COMPUTE verb. FORTRAN also permits the use of "functions". FORTRAN functions are subprograms that perform a special operation and may appear as part of a FORTRAN assignment statement. For example,

```
Y = A*(SQRT(Z))
```

takes the square root of Z, multiplies it by A and stores the result in Y. There are many predefined FORTRAN functions such as LOG, SIN, COS, EXP, INT, and FRAC. The programmer can define many more if he desires. FORTRAN functions can be passed through a call. For example, let SAT and ART be FORTRAN functions and

```
SUBROUTINE NEAT (X,Y, FUNCT) be a FORTRAN subroutine
```

that requires a specific equation identified by "FUNCT" that may change from time to time. The following uses of NEAT each use different functions; that is, FUNCT is replaced in each case by the function identified:

```
CALL NEAT (X,Y, SAT)
CALL NEAT (X,Y, ART)
```

These two calls may coexist quite comfortably in the same subprogram. Figure 3-2 is a list of FORTRAN statements and their meanings.

3.3.6 Advantages and Disadvantages of FORTRAN. FORTRAN has several advantages. All kinds of complex mathematical operations can be performed with relative ease. The statements are short and very close to being equations. Many library subroutines are available and the programmer can define still others.

The subprograms are short and easily modularized. Unfortunately, since FORTRAN variables cannot exceed six characters in length, the subprograms are not self documenting. A programmer can make data handling errors that the compiler will not catch. Data manipulation may not be very efficient. Therefore, FORTRAN has some serious drawbacks. (19: 1113-1116) (General References for this section - 6, 10, 12, 17).

FIGURE 3-1

BASIC DEFINITIONS OF COBOL WORDS (VERBS) (6, 7, 12, 14, 16)

	ACCEPT	- Causes low volume data to be made available from Special Names paragraph.
	ADD	- Add one or more items to a resultant field.
	ALTER	- Changes the destination of a go-to from one procedure name to another.
	CALL	- Transfers control to a separately compiled program.
#	CANCEL	- Releases storage occupied by a called program.
	COPY	- Places prewritten text into a COBOL program.
	CLOSE	- Terminates processing of reels, units, files.
	COMPUTE	- Sets one or more items equal to the value of an arithmetic expression.
#	DISABLE	- Notifies Message Control System (MCS) to transfer data between specified sources and input/output queues.
	DISPLAY	- Transmits low volume data to a specified output device.
	DIVIDE	- Divides one item into another.
#	ENABLE	- Allows MCS to transfer data between specified sources and input/output queues.
*	ENTER	- Includes specific statements in source programs not defined in COBOL.
	ENTRY	- Establishes an alternate entry point into a COBOL called subroutine.
*	EXAMINE	- Changed to INSPECT in ANS 74 COBOL.
	EXIT	- Defines exit point for a series of procedures.
	GENERATE	- Presents a report entry based on PROCEDURE DIVISION control.
#	GO BACK	- Specifies the logical end to a called program.
	GO TO	- Transfers control to another part of the PROCEDURE DIVISION.
	IF	- Evaluates a condition; action taken based on whether condition is T or F.
	INITIATE	- Begins processing of a report.
#	INSPECT	- Specifies that characters in a data item are to be counted, replaced, or both.
	MERGE	- Combines two or more identically sequenced files.
	MOVE	- Transfers data to one or more receiving fields.
	MULTIPLY	- The product of two items.
	OPEN	- Initiates the processing of files.

	PERFORM	- Allows departure from normal sequence to execute other procedures a specified number of times, or until a condition is met, returning to sequence.
	READ	- Makes available the next logical record; allows action when EOF detected.
#	RECEIVE	- Makes available a message & pertinent data about it from MCS input queue.
	RELEASE	- Transfers records to the initial phase of a sort operation.
*	RETURN	- Obtains the sorted records from the final phase of a sort operation.
#	REWRITE	- Logically replaces an existing record in a mass storage file.
	SEARCH	- Search a table for an element that satisfies a condition.
*	SEEK	- Initiate accessing a mass storage record for subsequent reading/writing.
#	SEND	- Causes a message to be released to the MCS (Message Control System).
	SET	- Associates index names with table elements for table handling reference points.
	SORT	- Puts the records of a file into a predefined order.
#	START	- Provides a means of positioning within a file for later sequential retrieval.
	STOP	- Halts execution temporarily or permanently.
#	STRING	- Puts together the contents of two or more items into a single data item.
	SUBTRACT	- Subtract one item from two or more items.
#	TRANSFORM	- Alters characters according to a transformation rule.
	TERMINATE	- Terminates processing of a report.
#	UNSTRING	- Separates contiguous data into multiple receiving fields.
	WRITE	- Releases a logical record for an output file.

* Honeywell Peculiar

IBM Peculiar

FIGURE 3-2

Basic Definitions of FORTRAN Statements (10, 15, 17)

*	ABNORMAL	- Reverses default interpretation, as side effects of function may alter output.
	ASSIGN	- Gives status of item as numeric, character, logical, or label.
	BACKSPACE	- Backup one record.
	BLOCK DATA	- One way to enter data into a labeled common block during compile.
	CALL	- Transfers control to a subprogram.
*	CHARACTER	- A string of characters composed of any combination of characters of the ASCII or BCD character set.
	COMMON	- Assigns two elements in different programs to the same storage location.
	COMPLEX	- A number composed of a real part and an imaginary part, i.e., (2,i).
	CONTINUE	- Dummy statement used to end the range of a DO.
	DATA	- Allows programmer to enter data at compile time.
#	DEBUG	- Sets conditions for operation and designates debugging options for entire program.
*	DECODE	- Causes a character string to be converted to data items.
#	DEFINE FILE	- Describes characteristics of data sets to be used; direct access statement.
	DIMENSION	- Provides information to allocate storage for arrays and their size.
#	DISPLAY	- Eliminates need for format or namelist and write in debug packet.
	DO	- Allows departure from normal sequence to execute a block of code n times.
	DOUBLE PRECISION	- A two word number in floating point format.
*	ENCODE	- Causes data items to be converted to a character string.
	END	- Specifies the physical end of the source program.
	ENDFILE	- Causes the indicated (sequential) file to be closed with an end of file signal.
	ENTRY	- Specifies a point of entry into another program or subroutine.
	EQUIVALENCE	- Assigns variables in the same program to the same storage location.
	EXPLICIT	- Declares type of variable by name instead of initial character.
	EXTERNAL	- Permits the use of a subprogram as an argument in another subprogram call.

#	FIND	- Overlaps record retrieval from direct access device; direct access statement.
	FORMAT	- Provides conversion and editing for input/output statements.
	FUNCTION	- First statement of a function subprogram; states relationship between variables.
#	GENERIC	- Declares a set of names to be generic.
	GO TO	- Indicates the next statement to be executed.
	IF	- Checks for a condition, respond accordingly: logical, arithmetic, implicit.
	IMPLICIT	- Used to declare the type of variables in a program.
	INTEGER	- A fixed point number.
	LOGICAL	- Item with a logical value of true or false.
*	PARAMETER	- Defines program constants as the result of an expression at compile time.
	PAUSE	- Causes a temporary halt in execution until resumed by operator.
	PRINT	- Used for list directed, formatted output to the standard system output device.
	PUNCH	- Causes converted data, in punchable form, to be transmitted to the standard output device.
	READ	- Transfers data from a direct access device to internal storage.
	REAL	- A floating point number.
	RETURN	- Logical end of any subprogram, returning control to the calling program.
	REWIND	- Positions specified file at its' initial point.
	STOP	- Halts object program and returns control to operating system.
	SUBROUTINE	- Must be first statement of a called subroutine.
	TYPE	- Defines data items as Integer, Real, Double-Precision, Logical, Complex, or Character.
#	WAIT	- Completes data transmission begun by asynchronous Read/Write.
	WRITE	- Used for formatted output to a designated output device.

* Honeywell Peculiar

IBM Peculiar

SECTION 4. THE COMPUTERS

4.1 Introduction. Some knowledge of the computer hardware operation is desirable to understand the data and subroutine linkage or interface problems associated with the use of more than one computer language in the same program. This section discusses some of the hardware operation of the IBM and Honeywell computers.

4.2 The Bit. BIT is an acronym for BInary digiT (BIT). (2:5) There are only two binary digits: 0 and 1. Simple electrical circuits are either on or off and therefore may be used to represent the binary digits. These simple circuits may be combined using the principles of Boolean algebra to perform comparisons and computations with groups of binary digits. Decimal numbers may be represented by groups of bits in basically two ways:

a. The binary number system equivalent to the decimal number can be used to represent the decimal number. This configuration is called a binary form. (3, 5)

b. The decimal number can be represented by a somewhat arbitrary bit code called Binary Coded Decimal (BCD). Sometimes the term BCD is used to refer to a specific code called the Natural Binary Coded Decimal (NBCD), see Figure 4-1. In this report the usage of BCD is more general. (19:164) Computers may use either or both representations to perform computations. Most third generation computers use both representations, but normally perform computations using only the binary representation. (3, 5)

4.3 The Byte. A group of binary digits that are considered or operated on as a unit is called a byte. (2:6) Normally, a byte is a specified number of bits that represent numbers and characters according to some code. For example, the Honeywell 6000 standard character set uses a six bit byte. The letter "A" is represented by the Honeywell system as

010001

The numbers 0 to 9 in the Honeywell system are represented by the NBCD system with leading zeros. Figure 4-2 lists the Honeywell, IBM Extended Binary Coded Decimal Interchange Code (EBCDIC) and American Standard Code for Information Interchange eight bit code (ASCII-8). EBCDIC and ASCII-8 are 8 bit byte codes. (2, 3, 5, 19)

Natural Binary Coded Decimal System

Decimal Number	Binary Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

"Forbidden" Binary Combinations

1010
1011
1100
1101
1110
1111

Reference (19:164)

FIGURE 4-1

CHARACTER SETS

These Symbols are not used for normal computation

ALPHABETIC CHARACTERS

Character	EBCDIC Binary	EBCDIC HEX	Honeywell Binary	Honeywell Octal	ASCII-8 Binary
A	1100 0001	C1	010001	21	1010 0001
B	1100 0010	C2	010010	22	1010 0010
C	1100 0011	C3	010011	23	1010 0011
D	1100 0100	C4	010100	24	1010 0100
E	1100 0101	C5	010101	25	1010 0101
F	1100 0110	C6	010110	26	1010 0110
G	1100 0111	C7	010111	27	1000 0111
H	1100 1000	C8	011000	30	1010 1000
I	1100 1001	C9	011001	31	1010 1001
J	1101 0001	D1	100001	41	1010 1010
K	1101 0010	D2	100010	42	1010 1011
L	1101 0011	D3	100011	43	1010 1100
M	1101 0100	D4	100100	44	1010 1101
N	1101 0101	D5	100101	45	1010 1110
O	1101 0110	D6	100110	46	1010 1111
P	1101 0111	D7	100111	47	1011 0000
Q	1101 1000	D8	101000	50	1011 0001
R	1101 1001	D9	101010	51	1011 0010
S	1110 0010	E2	110010	62	1011 0011
T	1110 0011	E3	110011	63	1011 0100
U	1110 0100	E4	110100	64	1011 0101
V	1110 0101	E5	110101	65	1011 0110
W	1110 0110	E6	110110	66	1011 0111
X	1110 0111	E7	110111	67	1011 1000
Y	1110 1000	E8	111000	70	1011 1001
Z	1110 1001	E9	111001	71	1011 1010

FIGURE 4-2

Character	EDCDIC Binary	EBCDIC HEX	Honeywell Binary	Honeywell Octal	ASCII-8 Binary
a	1000 0001	81			1110 0001
b	1000 0010	82			1110 0010
c	1000 0011	83			1110 0011
d	1000 0100	84			1110 0100
e	1000 0101	85			1110 0101
f	1000 0110	86			1110 0110
g	1000 0111	87			1110 0111
h	1000 1000	88			1110 1000
i	1000 1001	89			1110 1001
j	1001 0001	91			1110 1010
k	1001 0010	92			1110 1011
l	1001 0011	93			1110 1100
m	1001 0100	94			1110 1101
n	1001 0101	95			1110 1110
o	1001 0110	96			1110 1111
p	1001 0111	97			1111 0000
q	1001 1000	98			1111 0001
r	1001 1001	99			1111 0010
s	1010 0010	A2			1111 0011
t	1010 0011	A3			1111 0100
u	1010 0100	A4			1111 0101
v	1010 0101	A5			1111 0110
w	1010 0110	A6			1111 0111
x	1010 0111	A7			1111 1000
y	1010 1000	A8			1111 1001
z	1010 1001	A9			1111 1010
			NUMBER CHARACTERS		
0	1111 0000	F0	000000	00	0101 0000
1	1111 0001	F1	000001	01	0101 0001
2	1111 0010	F2	000010	02	0101 0010
3	1111 0011	F3	000011	03	0101 0011
4	1111 0100	F4	000100	04	0101 0100
5	1111 0101	F5	000101	05	0101 0101
6	1111 0110	F6	000110	06	0101 0110
7	1111 0111	F7	000111	07	0101 0111
8	1111 1000	F8	001001	10	0101 1000
9	1111 1001	F9	001001	11	0101 1001

Character	EBCDIC Binary	EBCDIC HEX	Honeywell Binary	Honeywell Octal	ASCII-8 Binary
			SPECIAL CHARACTERS		
.	0100 1011	4B	011011	33	0100 1110
<	0100 1100	4C	011110	36	0101 1100
(	0100 1101	4D	011101	35	1011 1011
+	0100 1110	4E	110000	60	0100 1011
&	0101 0000	5D	011010	32	0100 0110
!	0101 1010	5A	111111	77	0100 0001
$	0101 1011	5B	101011	53	0100 0100
*	0101 1100	5C	101100	54	0100 1010
)	0101 1101	5D	101101	55	1011 1101
;	0101 1110	5E	101110	56	0101 1011
-	0110 0000	60	101010	52	0100 1101
/	0110 0001	61	011111	37	0100 1111

Character	EBCDIC Binary	EBCDIC HEX	Honeywell Binary	Honeywell Octal	ASCII-8 Binary
,	0110 1011	6B	111011	73	0100 1101
%	0110 1100	6C	111100	74	0100 0101
(Underscore) _	0110 1101	6D			1011 1111
>	0110 1110	6E	001110	16	0101 1110
?	0110 1111	6F	001111	17	0101 1111
:	0111 1010	7A	001101	15	0101 1010
#	0111 1011	7B	001011	13	0100 0011
@	0111 1100	7C	001100	14	1010 0000
'	0111 1101	7D	101111	57	0100 0111
=	0111 1110	7E	111101	75	0101 1101
"	0111 1111	7F	111110	76	0100 0010

NOTE: Control characters and some machine peculiar special characters have been omitted from this table. Reference (3, 5, 19)

It is convenient to print a byte in a number system other than the binary system. The octal and hexidecimal system are used for this purpose. The Honeywell 6000 series computers use two octal numbers to describe one six bit byte. The IBM 360/370 series of computers uses a two digit hexidecimal number. Hence, an "A" in the Honeywell System would be printed in a binary dump as "21". (2, 3)

4.4 The Word. A computer word is normally the group of bytes that the computer manipulates with one instruction. (2:62) The standard Honeywell 6000 series word is 36 bits or six bytes in length. The IBM 360/370 word length is 32 bits or four bytes. These lengths are the sizes of the primary working registers in the computer. Hence, a computer normally operates on one word at a time. A word can be a string of BCD data, a machine instruction or a binary system number. The specific use of a word depends upon the context in which it appears in the sequence of operations in the machine. For example, the 36 bit Honeywell word

001010010010010011101001101000000000

which in octal form is

122223515000

represents

[BCRQØ

when used as a BCD word, or when used as an instruction,

"compare the floating point number at address 122223 to bits 0-27 of the AQ register".

If this word is used as a binary number, it can be either an integer (fixed point) or floating point number.

As an integer it is

11,044,559,360

and as a floating point number it is

$6.34153597 \times 10^{11}$.

These varied meanings for the same word create a severe linkage problem. If one subprogram considers a specific word a fixed point number and another subprogram considers that word a floating point number, a gross error will result. (3, 5)

4.5 Addressing. The computer locates a word through a location number called an address. Some computers, e.g., the Honeywell 6000 series, assign address numbers to words. Other computers, e.g., the IBM 360/370 series, assign address numbers to bytes. In either case the computer can operate on both words and bytes. The mechanism by which a computer addresses words or bytes is characteristic of the machine level language and the same procedure will generally be used regardless of the higher level language used. (3, 5)

4.6 Subprogram Addressing and Linkage. Subprograms occupy a distinct section of computer core. There are two types of subprograms that are of importance to this report: Subroutines and functions. Subroutines are of two types: opened and closed. An open subroutine is a set of code that is inserted in the main program or other subprogram where it is needed. This type of subroutine makes inefficient use of core and is not normally used in higher level languages. The closed subroutine is a set of code that occurs only once in core, but is used by the main program or other subprogram through a transfer of program control to that area of core when the subroutine is used. Hereafter, subroutine shall mean closed subroutine. The function subprogram is very similar to the closed subroutine. The difference between a function subprogram and a closed subroutine is in the way in which these subprograms are used in the source code. A function is inserted into a line of code. For example, the function SQRT may be used as follows: Y = A*SQRT(X). A closed subroutine must be called. For example, a subroutine called ORBIT may be used as follows:

```
CALL ORBIT (X,Y,Z,NAMES) or CALL ORBIT USING X,Y,Z,NAMES.
```

In both examples the computer sets up a table with addresses associated with SQRT and ORBIT. These addresses are the entry or beginning point of the subprogram. When a function or subroutine is used, program control is passed to the instruction at the address of the function or subroutine. The address of the next instruction in the calling program is stored. When the function or subroutine task is completed, control of the program returns to the calling program by returning to the instruction whose address was stored. Before execution the computer has inserted subprogram addresses in the program. However, the subprogram table exists during the compilation and loading/linking activities. Hence, the computer can link subprograms written in different higher level languages to the main program and other subprograms. Library routines are also linked during the loading/linking activity. Since linking and address resolution take place after compilation, simple subprogram linkage is usually not a problem. (19:1370-1373)

4.7 Types of Data and Data Addressing. The linkage problem becomes more difficult when data is transferred from one subprogram to another. If only one higher level language is used, the compiler will normally resolve data transference difficulties. However, when more than one language is used serious problems can result. These problems are discussed in detail in Section 5. This paragraph describes the types of data and how they are addressed.

a. Display or Hollerith Data. Numeric and character representation was described in paragraph 4-3. In the COBOL language terminology these data are called "display items". In the FORTRAN language the terms "character" or "Hollerith" data are used. Each individual character is stored in one byte. A string of characters is normally left justified in the first and last computer words that the string uses. The address of the string is the address of the first byte. Of great importance is the fact that the end of one string of character bytes can occupy the same word as the beginning of another string. For some computer languages, e.g., FORTRAN, this characteristic can be problematic. (3, 5, 12)

b. Packed Decimal Data. Packed decimal data are special numeric data only and use the NBCD system. The Honeywell 6000 series computer can pack eight NBCD characters into one 36 bit word. The IBM 360/370 series computers can also pack eight NBCD characters into its 32 bit word. Four bits in the Honeywell word are unused in packed decimal representation. Unfortunately, some computer languages, e.g., FORTRAN, cannot handle packed decimals very well. Hence, packed decimals create linkage problems. (3, 5)

c. Zoned Decimal Numbers. Zoned decimal numbers are NBCD characters separated by a bit string called a zone. The IBM 360/370 series uses a four bit zone and the Honeywell 6000 series uses a five bit zone (the leading bit is zero). These numbers are used for input/output where the zone bits represent the presence or lack of zone punches on cards and therefore the difference between sets of ASCII/EBCDIC characters. Zoned decimal numbers and packed decimal numbers are treated for address purposes like character data. (3, 5)

d. Fixed Point Data. Fixed point data is a straight binary representation of decimal data usually used for integer arithmetic. The binary point is assumed to be at the extreme right of the number. If the binary point is assumed to be at the extreme left, the number can be

treated as a fraction. The programmer or compiler must normally keep track of the decimal point. Any fixed point number must be aligned on a word boundary for computations. Therefore, in the IBM 360/370 series machine the address of the word should be divisible by 4 before computations are performed. Double fixed point numbers are possible in the Honeywell 6000 series machine. Any such number must begin at an even address. There are two types of fixed point numbers: algebraic and logical. Algebraic numbers use the extreme left bit as a sign bit. Logical numbers have no sign bit. Arithmetic may be performed with either type. Both the IBM 360/370 and Honeywell 6000 series machines have the capability of performing arithmetic on half words; i.e., 16 bits in the IBM and 18 bits in the Honeywell. The ranges of these fixed point numbers are listed in Figure 4-3 and Figure 4-4. (3, 5)

e. Single Precision Floating Point Data. Floating point data is a binary representation of a number using an exponent and mantissa. In both the Honeywell 6000 series and IBM 360/370 series machines, the first eight bits of a floating point representation is a binary exponent. The remaining bits of the word represent a binary mantissa. This mantissa is usually a "normalized binary fraction", that is, the binary or hexidecimal point is assumed to be to the left of the first bit in the binary form of the mantissa and this bit must be non-zero (on) so that all the binary digits are significant. The process of normalization requires an adjustment of the exponent. Hence, any number X is represented mathematically by

$$X = A\ (2^E)$$

where A is the normalized mantissa and E is the binary exponent. In some IBM literature the exponent is called the "characteristic" and the mantissa is called the "fraction". As an example, refer to the 36 bit Honeywell word in paragraph 4-4. The exponent is

00101001

The leading bit is 0 which means that the exponent is positive. The value of this exponent is 41. the mantissa is

001001001110100110100000000

The leading bit is the sign bit which in this case indicates that the number is positive. The mantissa is not

Ranges of Fractional Fixed Point Representations

	Honeywell	IBM
Algebraic Single Word	-1 to $(1-2^{-35})$	-1 to $(1-2^{-31})$
Logical Single Word	0 to $(1-2^{-36})$	0 to $(1-2^{-32})$
Algebraic Double Word	-1 to $(1-2^{-71})$	-1 to $(1-2^{-63})$
Logical Double Word	0 to $(1-2^{-72})$	0 to $(1-2^{-64})$
Algebraic Half Word	-1 to $(1-2^{-17})$	-1 to $(1-2^{-15})$
Logical Half Word	0 to $(1-2^{-18})$	0 to $(1-2^{-16})$

References (3, 5)

Algebraic Fixed Point Format

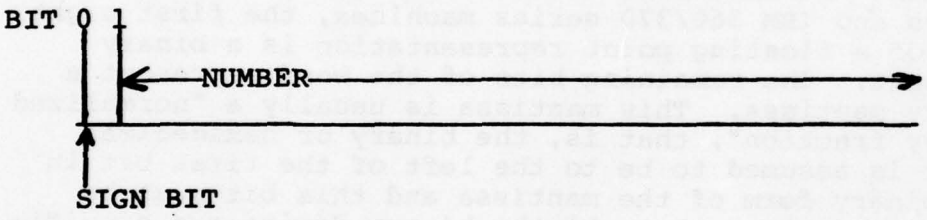


FIGURE 4-3

Ranges of Integral Fixed Point Representations

	Honeywell	IBM
Algebraic Single Word	-34359738368 to 34359738367	-2147483648 to 2147483647
Logical Single Word	0 to 68719476735 -2^{71} to $(2^{71}-1)$	0 to 4294967295 -2^{63} to $(2^{63}-1)$
Logical Double Word	0 to $(2^{72}-1)$	0 to $(2^{63}-1)$
Algebraic Half Word	-131072 to 131071	-32768 to 32767
Logical Half Word	0 to 262143	0 to 65535

References (3, 5)

FIGURE 4-4

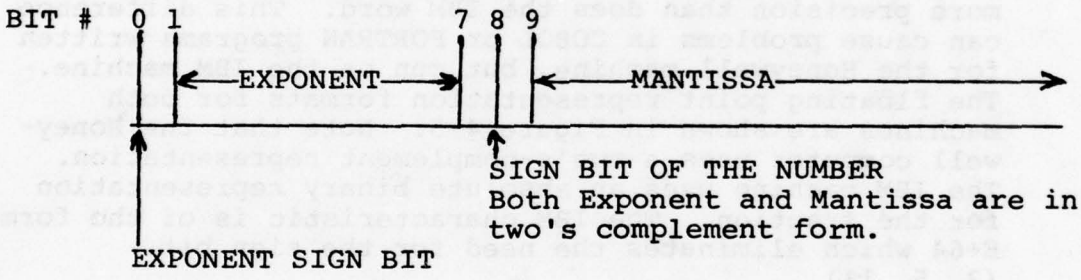
Ranges of Floating-Point Numbers

Single Precision Honeywell	-2^{127} to $(1-2^{-27})\ 2^{127}$
Single Precision IBM	$-(1-2^{-25})\ 16^{63}$ to $(1-2^{-25})\ 16^{63}$
Double Precision Honeywell	-2^{127} to $(1-2^{-63})\ 2^{127}$
Double Precision IBM	$-(1-2^{-57})\ 16^{63}$ to $(1-2^{-57})\ 16^{63}$

NOTE: Numbers of very small magnitude (less than about 2^{-128} on the Honeywell or 16^{-64} on the IBM) are considered to be zero.

(References 3, 5)

Honeywell Floating-Point Format



IBM Floating-Point Format

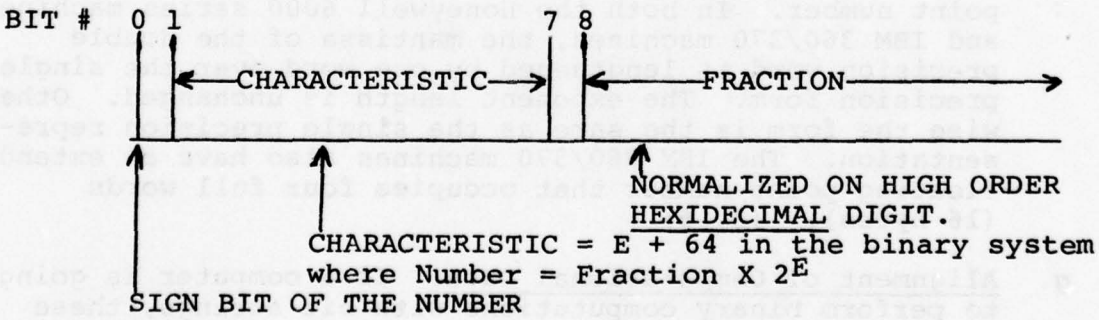


FIGURE 4-5

normalized. The machine would probably normalize this number prior to performing arithmetic with it. In its present form this binary fraction is

$$1/4 + 1/32 + 1/256 + 1/512 + 1/1024 + 1/4096 + 1/32768 + 1/65536 + 1/262144$$

or

$$0.288379669.$$

Hence, the number is

$$0.288379669\ (2^{41})$$

or

$$6.34153597 \times 10^{11}$$

Note that the Honeywell word, being larger, allows for more precision than does the IBM word. This difference can cause problems in COBOL or FORTRAN programs written for the Honeywell machine, but run on the IBM machine. The floating point representation formats for both machines are shown in Figure 4-5. Note that the Honeywell computer uses a two's-complement representation. The IBM machine uses an absolute binary representation for the fraction. The IBM characteristic is of the form E+64 which eliminates the need for the sign bit. (3, 5, 11)

f. Double Precision Floating Point Data. If the floating point representation occupies two words instead of one, it is referred to as a "Double Precision" floating point number. In both the Honeywell 6000 series machines and IBM 360/370 machines, the mantissa of the double precision word is lengthened by one word over the single precision form. The exponent length is unchanged. Otherwise the form is the same as the single precision representation. The IBM 360/370 machines also have an extended floating point number that occupies four full words (16 bytes). (3, 5)

g. Alignment of Computational Data. If a computer is going to perform binary computations with bit strings, these strings must be correctly aligned in the processor registers. To insure correct alignment, the strings should be stored in core on specific word or byte boundaries. As was mentioned in paragraph d, the single precision fixed point number must occupy one word in

both the IBM and Honeywell machine and should begin at a word address (word boundary) in the Honeywell 6000 series or at an address divisible by four (single word boundary) on the IBM machine. The double precision fixed point number must occupy two words and should begin at an even address (double word boundary) in the Honeywell machine or at an address divisible by eight (double word boundary) in the IBM machine. The same rules apply to single and double precision floating point representations. Figures 4-6 and 4-7 show these requirements. In some machines, e.g., the IBM 370/3033, it is possible to store computational binary items beginning on any byte boundary. Although this practice may save space initially, there are several disadvantages:

i) Passing the data to another program may cause alignment or program boundary problems.

ii) Before computations can be performed, the machine must align the data properly somewhere in core. This procedure is an internal machine function and is not controlled by the programmer. Therefore storage space may not be saved.

iii) Computations are less efficient.

iv) Programs may not be machine independent.

Therefore aligning computational binary items on proper boundaries is recommended. (3, 5)

4.8 Implications of Data Structure. From the preceeding discussion one can see that it is possible to misuse data transferred to other subprograms or read from data files. For example, one subprogram could treat a datum as a fixed point number whereas another subprogram could treat it as a floating point number. This situation would normally produce gross errors that would not be detected by the computer. Similar problems are possible while reading/writing files. The programmer must insure that data is used in a consistent manner.

4.9 Implications of Data Alignment and Addressing. Character data and packed decimal data can be aligned on any byte boundary. Computational data, i.e., fixed or floating point data, should be aligned on word boundaries. Usually this arrangement does not pose address problems in that like data in the same machine is addressed the same way irrespective of the higher level language used. However, there are situations when data may need to be handled in an unorthodox way causing addressing problems.

HONEYWELL 6000 SERIES DATA ALIGNMENT

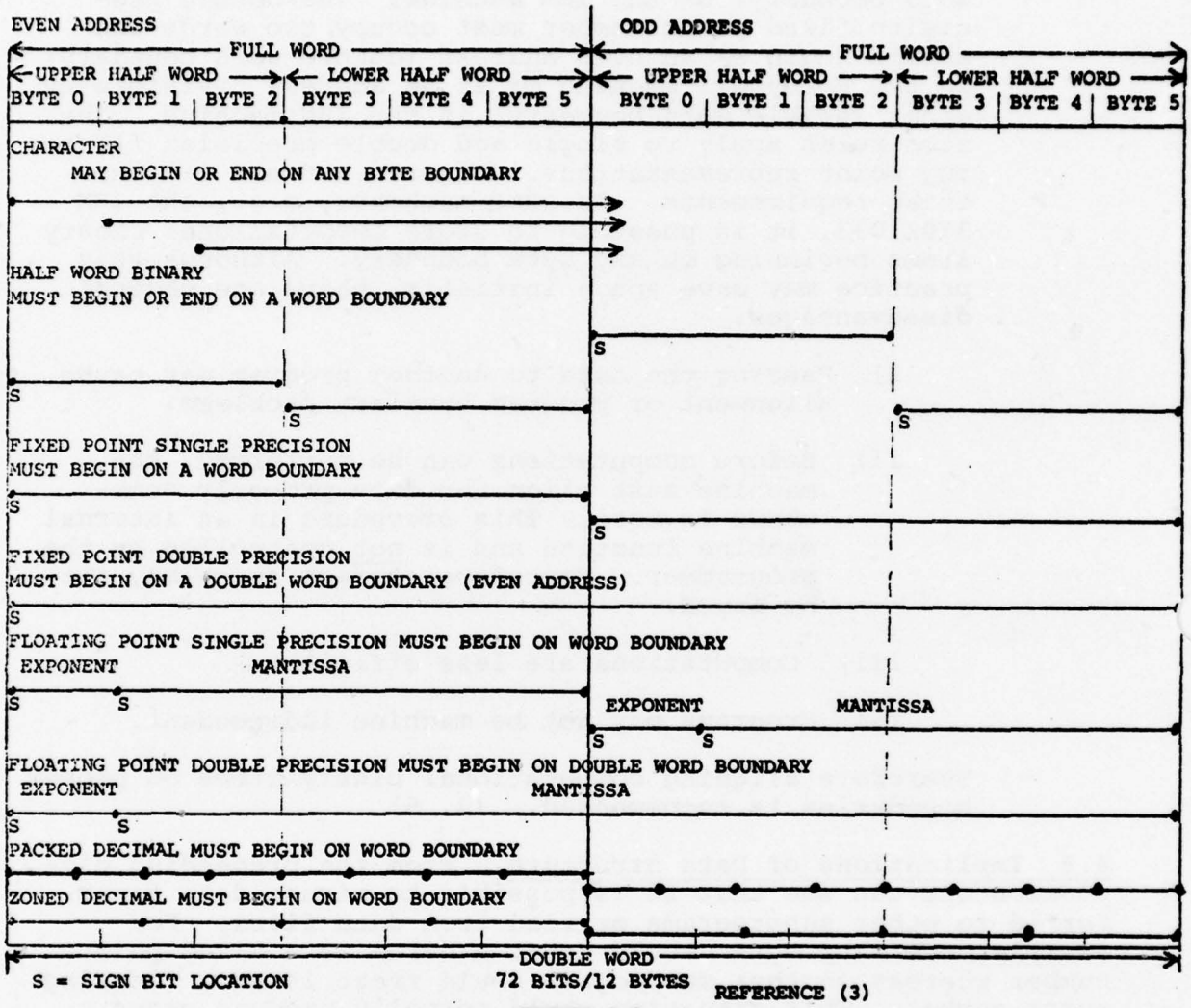
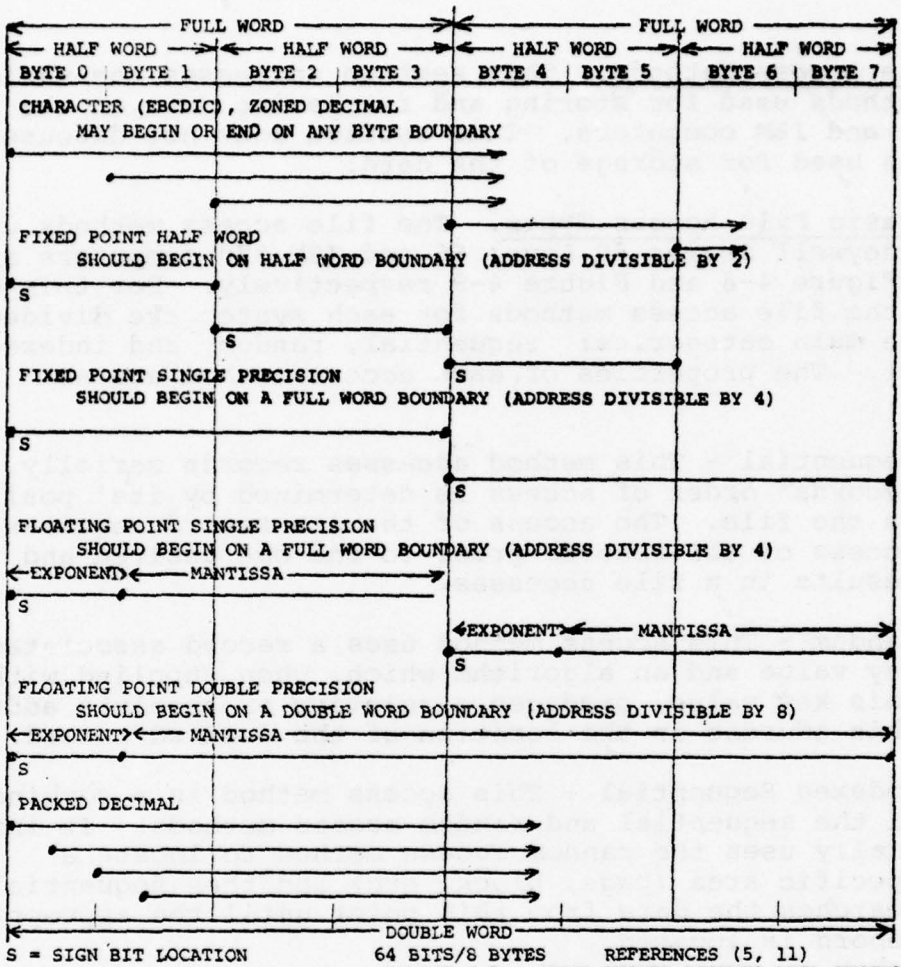


FIGURE 4-6

IBM 360/370 SERIES DATA ALIGNMENT



S = SIGN BIT LOCATION 64 BITS/8 BYTES REFERENCES (5, 11)

FIGURE 4-7 (THIS FIGURE ASSUMES DATA SYNCHRONIZATION)

For example, ANSI FORTRAN does not identify variables as character or display data. Hence character data is treated as computational data until outputed. If the data were established by a COBOL program as a display item, the data may not begin on a word boundary. Therefore part of the data could be lost if subsequently accessed by a FORTRAN subroutine. This problem is discussed in detail in Sections 5 and 8.

4.10 File Access Methods. This section addresses the various access methods used for storing and retrieving data on the Honeywell and IBM computers. This section does not discuss the structures used for storage of the data.

4.10.1 Basic File Access Types. The file access methods used by the Honeywell Series 60 Level 66 and IBM 370 computers are shown in Figure 4-8 and Figure 4-9 respectively. For this discussion, the file access methods for each system are divided into three main categories: sequential, random, and indexed sequential. The properties of each access method are as follows:

a. Sequential - This method accesses records serially. The records' order of access is determined by its' position in the file. The access of the nth record requires the access of all records prior to the one desired and results in n file accesses.

b. Random - This access method uses a record associated key value and an algorithm which, when supplied with this key value, produces a relative or absolute address. This address is the location of the desired record.

c. Indexed Sequential - This access method is a combination of the sequential and random access methods. It initially uses the random access method to locate a specific area (page, block, etc) and then sequentially searches the data from this point until the appropriate record is located.

The sequential, indexed sequential, and random file access methods are performed by calling special system I/O routines. The specifics for calling the I/O routines can be found by referencing the appropriate systems's manuals. (7, 11, 14)

4.10.2 Honeywell Files. The Honeywell file access methods are discussed below:

a. The Honeywell computer's random access method is called Random. Another file system available on the Series 60 Level 66 computer is the Integrated Data Store (IDS) system. This system utilizes a random access method designed to store fields of information and link those fields in a number of ways depending upon the desired application.

b. The Honeywell indexed sequential access method is the Indexed Sequential Processor (ISP).

c. The sequential access method is called Sequential.

4.10.3 IBM File Access Methods. The file access methods of the IBM 370 system are as follows:

a. The random access method is called the Direct Access Method (DAM). Another IBM file system which uses a random access method is the Integrated Data Base Management System (IDMS). This system is comparable to the IDS system described for the Honeywell machine. In addition, it allows the accessing of a single field of information using various field descriptions.

b. The indexed sequential access method is called Indexed Sequential Access Method (ISAM).

c. The sequential file access method is the Sequential Access Method (SAM).

d. The last access method to be discussed is the Virtual Storage Access Method (VSAM). This method actually encompasses all three types of file access methods: sequential, random, and indexed sequential. The VSAM file organization was developed to provide optimized usage of virtual storage and is used with direct access storage devices.

4.10.4 Honeywell/IBM Language/File Relationships. The Honeywell Series 60 Level 66 and IBM 370 computers have some file access methods which appear similar in concept. The relationships between these systems are shown in Figure 4-10.

a. Figure 4-10 provides some guidance for selecting the appropriate file access method when programs are transferred from one machine to another. Although the specific language verbs may not be precisely the same, the overall concepts behind the file organizations are similar.

b. Another consideration when working with any of the file access methods on either the Honeywell or IBM computer systems is the languages' ability to read or write using a particular file access method. Figure 4-11 illustrates some COBOL/FORTRAN language restrictions for reading and writing to either Honeywell or IBM files. It should be noted that the IDS file organization on the Honeywell computer and the IDMS, ISAM, and VSAM file access methods on the IBM computer can only be used by COBOL programs.

BASIC FILE ACCESS TYPE	HONEYWELL FILE ACCESS METHOD
SEQUENTIAL	SEQUENTIAL
INDEXED SEQUENTIAL	INDEXED SEQUENTIAL PROCESSOR (ISP)
RANDOM	RANDOM INTEGRATED DATA STORE (IDS)

FIGURE 4-8

HONEYWELL SERIES 60 LEVEL 66 FILE STRUCTURE

BASIC FILE ACCESS TYPE	IBM FILE ACCESS METHOD
SEQUENTIAL	SEQUENTIAL ACCESS METHOD (SAM)
INDEXED SEQUENTIAL	INDEXED SEQUENTIAL ACCESS METHOD (ISAM)
RANDOM	DIRECT ACCESS METHOD (DAM) INTEGRATED DATA BASE MANAGEMENT SYSTEM (IDMS)

FIGURE 4-9

IBM 370 FILE STRUCTURE

BASIC FILE ACCESS TYPE	IBM FILE ACCESS METHOD	HONEYWELL FILE ACCESS METHOD
SEQUENTIAL	SAM	SEQUENTIAL
INDEXED SEQUENTIAL	ISAM	ISP
RANDOM (DIRECT)	DAM IDMS**	RANDOM* IDS**
SEQUENTIAL INDEXED SEQUENTIAL RANDOM	VSAM	-

*NOTE: User requirements for operation of the random access methods for the Honeywell and IBM machines differ.

**NOTE: Reference paragraph 4.10.3(a).

FIGURE 4-10

FILE ACCESS METHOD RELATIONSHIPS

HONEYWELL

SEQUENTIAL	WRITTEN IN COBOL	WRITTEN IN FORTRAN
READ IN COBOL	YES	YES
READ IN FORTRAN	YES	YES
ISP		
READ IN COBOL	YES	YES
READ IN FORTRAN	YES	YES
RANDOM		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	YES
IDS		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	NO

IBM

SAM	WRITTEN IN COBOL	WRITTEN IN FORTRAN
READ IN COBOL	YES	YES
READ IN FORTRAN	YES	YES
ISAM		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	NO
DAM		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	YES
IDMS		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	NO
VSAM		
READ IN COBOL	YES	NO
READ IN FORTRAN	NO	NO

FIGURE 4-11

FILE ACCESS METHODS AND LANGUAGE RELATIONSHIPS

SECTION 5. DATA REPRESENTATION

5.1 Introduction. Before effective interfaces between FORTRAN and COBOL can be accomplished, the programmer must be aware of the manner in which his data is being stored. The manner in which data are stored varies between data types, languages, and computers. Following is a discussion of this problem.

5.2 The IBM Computer. The IBM computer uses words composed of 4 bytes (see Section 4 for a more complete discussion). Each byte contains 8 bits. The addressing scheme on IBM computers is to use the number of the first byte in the word as the address of the entire word (recall bytes are numbered beginning with Ø). Thus, the address of the first word is Ø; the address of the second word is 4 (the first word has 4 bytes numbered Ø, 1, 2, 3; the second word contains bytes 4, 5, 6, 7; etc), the address of the third word is 8, etc. In addition to using a full-word (4 bytes) of storage, the IBM computer can also use half-words (2 bytes) or double-words (8 bytes). The word length the computer will use depends upon which data type is being used. However, when passing data from one language to another it becomes critical to know on which word boundary data are aligned. A boundary is a half-word boundary if the address is evenly divisible by 2; a full-word boundary if it is evenly divisible by 4; and a double word boundary if evenly divisible by 8.

5.2.1 IBM COBOL. Following are the data types found in IBM COBOL and the word alignment used.

5.2.1.1 SYNC vs Non-SYNC. The word alignments used by COMP, COMP-1, or COMP-2 items vary between compilers. Without the SYNC usage it is very difficult to determine the proper word alignment. Therefore, all COMP, COMP-1, and COMP-2 items should have a SYNC usage.

5.2.1.2 COMP SYNC. The COMP SYNC usage is used for fixed-point (integer) data representation. The word boundary alignment for COMP SYNC depends on the PICTURE clause being used. If the PICTURE clause contains 1-4 digits the computer will use a half-word for storage and will align it on a half-word boundary. If the PICTURE contains 5-9 ditits a full-word will be used and aligned on a full-word boundary. If the PICTURE contains 10 or more digits 2 full full-words will be used; however, the alignment is on a full-word boundary, not necessarily a double word boundary.

5.2.1.3 COMP-1 SYNC. The COMP-1 SYNC usage is used for single precision floating-point (real) data representation. COMP-1 SYNC uses a full-word of storage aligned on a full-word boundary.

5.2.1.4 COMP-2 SYNC. The COMP-2 SYNC usage is used for double precision floating-point (real) data representation. COMP-2 SYNC uses a double word of storage aligned on a double word boundary.

5.2.1.5 COMP-3. The COMP-3 usage is used for internal decimal items and stores them in packed decimal format. COMP-3 items may begin on any byte, and extend for as many bytes as necessary.

5.2.1.6 COMP-4. COMP-4 is equivalent to the COMP format (see paragraph 5.2.1.2).

5.2.1.7 DISPLAY. The use of DISPLAY results in data being represented in character format. The data may begin on any byte and continue for as may bytes as necessary. Unless COMP, COMP-1, COMP-2, COMP-3, or COMP-4 is specified the default will be DISPLAY, even if the PICTURE clause is numeric. For example, if the following is in a COBOL program:

```
77  ITEM PICTURE S99 VALUE +12.
```

the character representation of the number +12 will be stored in ITEM.

5.2.2 IBM FORTRAN. Following are the data types found in IBM FORTRAN and their corresponding word alignment.

5.2.2.1 INTEGER*2. The INTEGER*2 field is used for single precision fixed-point (integer) data representation. INTEGER*2 uses a half-word of storage and is aligned on a half-word boundary.

5.2.2.2 INTEGER. The INTEGER (or INTEGER*4) field is used for double precision fixed-point data representation. INTEGER (or INTEGER*4) uses a full-word of storage and is aligned on a full-word boundary.

5.2.2.3 REAL. The REAL (or REAL*4) field is used for single precision floating-point (real) data representation. REAL*4 (or REAL) uses a full-word of storage and is aligned on a full-word boundary.

5.2.2.4 DOUBLE PRECISION. The DOUBLE PRECISION (or REAL*8) field is used for double precision floating-point (real) data representation. DOUBLE PRECISION (REAL*8) uses a double-word of storage and is aligned on a double-word boundary.

5.2.3 IBM COBOL/FORTRAN Interfacing. Programs written in one language calling routines in another language, be it COBOL calling FORTRAN or FORTRAN calling COBOL, require special care. Great attention must be taken to insure data items are compatible, and that word boundaries are correct. Below are some considerations to use when interfacing IBM COBOL and IBM FORTRAN.

5.2.3.1 Data Type Compatibility. In order for the computer to interpret the data correctly in both the calling program and the called subroutine, the data being passed must be consistently defined in both programs. Figure 5-3 shows the COBOL data types and their equivalent FORTRAN types. Figure 5-4 shows the FORTRAN data types and their equivalent COBOL types. For example, if the FORTRAN program is to pass an INTEGER value to the COBOL program, the COBOL data item must be defined as COMP. In addition, if the FORTRAN data item is INTEGER*2 the PICTURE clause in COBOL must contain 4 or less digits. If the item is INTEGER*4 the PICTURE clause must conain 5-9 digits.

5.2.3.2 Incompatible Data Types. As can be seen from Figures 5-3 and 5-4, data types exist in one language for which no corresponding data types exist in the other language. For example, a COMP usage with 10 or more digits in COBOL represents an integer number stored in a double word (see paragraph 5.2.1.1). FORTRAN, however, only has the ability of storing integer numbers in a half-word or a full-word, not in a double word (paragraph 5.2.2.2). Therefore, if a COBOL program passes a COMP value with a PICTURE containing 10 or more digits to FORTRAN, the FORTRAN program will be unable to handle the passed data correctly. However, it should be noted that when inconsistent data items are passed from language to language the computer will not issue an error statement. The computer will continue to operate on this incorrectly defined data, assuming this is exactly what the programmer had originally desired. It is, therefore, entirely up to the programmer to insure data items are defined the same in both the calling and called programs.

5.2.3.3 Word Boundary Alignment. The programmer is also responsible for insuring correct word boundary alignment on data items being passed from one language to another. Word boundary alignment may be the single most critical, and possibly the most difficult consideration when passing data between two languages. It is extremely important to insure that the word boundary of a particular data item matches the word boundary expected in the called routine. Following is a discussion of word boundary alignment.

5.2.3.3.1 Word Alignment. Paragraph 5.2.1 describes the IBM COBOL data types and the word boundaries on which these data types align. Paragraph 5.2.2 describes the FORTRAN data types

and alignment. One must keep these facts in mind when passing data from one language to another. Also, it is important to realize that COBOL begins Ø1 levels on a double word boundary. Using this information, it is possible to determine where word boundaries of the data items will fall.

5.2.3.3.2 Data Passing. Data is not actually passed from one routine to another. In reality, the item being passed is the address of the beginning of the data item. In FORTRAN this address would be the beginning of the data items represented by a variable name. In COBOL this address could be the beginning of an item, table, group item, etc. For example, if FORTRAN is passing a 3 x 4 array named ARRAY, the address of the beginning of ARRAY is passed. It is up to the receiving program to realize this address represents the beginning of a 3 x 4 array. No information stating the size, dimension, etc. of the data item is passed.

5.2.3.3.3 Internal Storage of Data. Data is stored in the computer's memory on a sequential basis. For example, assume the following appear in a COBOL program:

```
77  VALUE-1 PIC 99 VALUE 1.
77  VALUE-2 PIC 99 VALUE 2.
```

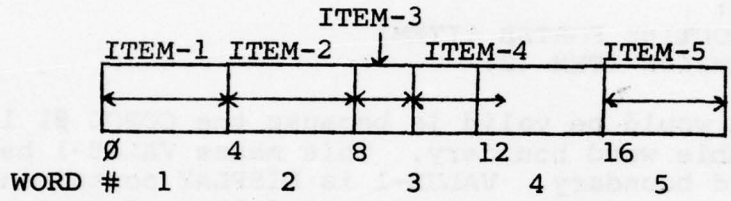
As the COBOL program is compiled internal storage will be allocated for both VALUE-1 and VALUE-2. Since these two items are defined sequentially in the program, the two items will be stored sequentially in the computer. Therefore, VALUE-1 will be immediately followed by VALUE-2. This is true of each data item described in a COBOL program, and is also true of each data item in FORTRAN.

5.2.3.3.4 COBOL Alignment. The easiest way to explain the manner in which COBOL aligns data may be to use the following example:

```
Ø1  TABLE.
    Ø3  ITEM-1 PIC XXXX.
    Ø3  ITEM-2 PIC S9(5) COMP.
    Ø3  ITEM-3 PIC S9(4) COMP.
    Ø3  ITEM-4 PIC XXX.
    Ø3  ITEM-5 COMP-1 SYNC.
```

In this example TABLE begins on a double word boundary because TABLE is a Ø1 level (paragraph 5.2.3.3.1). Since TABLE represents the collection of data items grouped beneath it, and not an individual data item, ITEM-1 begins at the same location as TABLE, i.e., a double word boundary. ITEM-1 is DISPLAY (paragraph 5.2.1.7), therefore each digit represents 1 byte. ITEM-1 contains 4 bytes (1 word). ITEM-2 begins immediately following ITEM-1; therefore ITEM-2 begins on a full-word boundary. ITEM-2 is COMP (paragraph 5.2.1.1) with a PIC of S9(5) which means

ITEM-2 occupies 1 word of storage. ITEM-3 begins following ITEM-2 and is on a full-word boundary. ITEM-3 is PIC S9(4) COMP and uses a half-word of storage. ITEM-4 begins following ITEM-3. Since ITEM-3 ended on a half-word boundary ITEM-4 begins on this half-word boundary. ITEM-4 is DISPLAY with a length of 3 bytes. This means ITEM-4 ends in the middle of a half-word. ITEM-5 follows ITEM-4, but ITEM-5 is COMP-1 SYNC (paragraph 5.2.1.3). COMP-1 SYNC items must begin on a full-word boundary. This compiler will skip the appropriate number of bytes necessary so that ITEM-5 can begin on the next full-word boundary. See the following diagram for a visual description of TABLE (note that the first full-word is numbered Ø. When determining word alignment, numbering the first full-word Ø makes the process simpler):



The unused bytes in word 4 are called "slack bytes." The compiler provided these bytes to force ITEM-5 to begin in a full-word boundary. Unless the programmer is familiar with the manner in which the compiler works, the programmer will be unaware of these slack bytes. The compiler does not inform the programmer when slack bytes are being inserted.

5.2.3.3.5 IBM FORTRAN Alignment. FORTRAN data items are aligned as described in paragraph 5.2.2. In an array, each element is aligned on the same type of boundary. FORTRAN does not allow group items, as COBOL does. Because of this fact slack bytes do not present the problem in FORTRAN as they do in COBOL.

5.2.3.4 Language Interfacing. Interfacing FORTRAN and COBOL presents some problems. Some of the more common problems, and their solutions are discussed below.

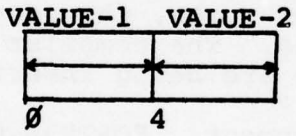
5.2.3.4.1 Padding. Just as it is necessary for the compiler to insert "slack bytes" (paragraph 5.2.3.3.4) for proper word boundary alignment in internal storage, it may be necessary for the programmer to insert "slack bytes" in the data structure for proper alignment. This becomes increasingly necessary when interfacing two languages such as FORTRAN and COBOL. Again it is very important for the programmer to realize where data alignment will occur.

For example, assume a COBOL program is to pass a group item to FORTRAN. This group item consists of VALUE-1 and VALUE-2, each PIC XXXX. If the FORTRAN routine receives the COBOL values in a two element array, with the first element of the array containing VALUE-1, and the second element containing VALUE-2, then the following would be correct:

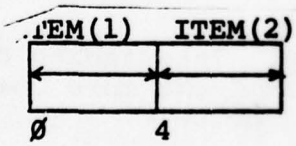
```
COBOL:
   Ø1  TABLE.
       Ø3  VALUE-1 PIC XXXX.
       Ø3  VALUE-2 PIC XXXX.
        .
        .
        .
       CALL 'FORTSB' USING TABLE.

FORTRAN:
   SUBROUTINE FORTSB (ITEM)
   DIMENSION ITEM (2)
```

The reason this would be valid is because the COBOL Ø1 level begins on a double word boundary. This means VALUE-1 begins on a double word boundary. VALUE-1 is DISPLAY containing 4 characters (bytes) and therefore is one full-word in length. VALUE-2 begins immediately following VALUE-1. This means VALUE-2 begins on a full-word boundary. Storage looks as follows:



The FORTRAN routine has ITEM dimensioned to 2. Each element of ITEM (by default) is 1 word in length, beginning on a full-word boundary. The FORTRAN storage appears as:

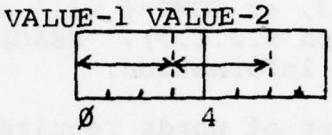


The COBOL storage begins on a double-word boundary; the FORTRAN on a full-word. Since a double word boundary is also a full-word boundary, the FORTRAN representation is compatible with the COBOL. The four characters of VALUE-1 will be in ITEM(1), the 4 in VALUE-2 will be in ITEM(2).

Now assume that the items in the COBOL program only contain 3 characters. The COBOL would appear as:

```
Ø1 TABLE.
   Ø3  VALUE-1 PIC XXX.
   Ø3  VALUE-2 PIC XXX.
    .
    .
    .
   CALL 'FORTSB' USING TABLE.
```

TABLE would still begin on a double word boundary. However, VALUE-1 and VALUE-2 would each contain 3 bytes, and be stored sequentially as:



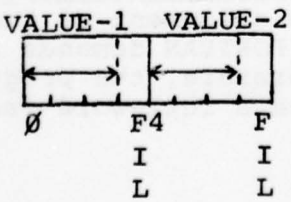
Notice the first byte of VALUE-2 appears as the last byte in word 1. When FORTRAN is called only the address of the beginning of TABLE (address Ø in this case) is passed. FORTRAN goes to this address and assumes storage is as FORTRAN expects it (as described above).

If VALUE-1 contained the characters ABC, and VALUE-2 the characters DEF, after the call to FORTRAN ITEM(1) would contain the characters ABCD and ITEM(2) would contain EFØØ, where ØØ may be blanks, or garbage characters.

In order for FORTRAN to receive the data properly (i.e., ITEM(1) containing ABC, and ITEM(2) containing DEF) padding in the form of a FILLER must be used in COBOL. One word of storage contains 4 bytes, or 4 characters. VALUE-1 and VALUE-2 contain 3 characters each. A FILLER of one character is necessary to complete the word. In addition, this filler should have a value of a blank in order to insure no garbage data is introduced. The following would be acceptable in the COBOL program:

```
Ø1 TABLE.
   Ø3  VALUE-1 PIC XXX.
   Ø3  FILLER PIC X VALUE SPACES.
   Ø3  VALUE-2 PIC XXX.
   Ø3  FILLER PIC X VALUE SPACES.
```

The COBOL storage would be:



Notice that VALUE-2 now begins in the second word. If the FORTRAN routine is now called using this TABLE, ITEM(1) would contain ABC, and ITEM(2) DEF.

5.2.3.4.2 Passing Character Strings. Passing character strings between COBOL and FORTRAN present some unique problems. Paragraph 5.2.3.4.1 used this problem as an example; however, the problem is more complex than this example.

One character requires one byte for storage. One word contains 4 bytes and can, therefore, hold a maximum of 4 characters.

Recall any COBOL data item which does not have a USAGE of COMP, COMP-1, COMP-2, COMP-3, or COMP-4 will automatically be considered DISPLAY (paragraph 5.2.1.7). USAGE DISPLAY means the data is stored as character information.

To demonstrate the number of words required assume the following:

```
Ø1  ITEM-1 PIC XX.
Ø1  ITEM-2 PIC XXXXX.
```

ITEM-1 would require a half-word of storage since PIC XX defines two characters of data. Two characters require two bytes (one half-word) of storage. ITEM-2 contains 5 characters. This would require 5 bytes of storage: one full-word plus one additional byte.

IBM FORTRAN does not contain a character data type as such. However, character data may be stored into either a fixed-point or a floating-point variable. If the variable is defined as single-precision (1 word of storage) 4 characters will be stored per variable element. If the variable is defined as double-precision (two words of storage) each variable element will contain 8 characters. The programmer must realize that FORTRAN requires the use of a full-word for each element regardless of how many characters may actually be contained in the word. The programmer must also realize that FORTRAN stores the character representation of character data in the variable. However, as far as FORTRAN is concerned the data is actually fixed or floating-point numbers. If arithmetic operations, comparisons, etc, are accomplished on this character (not numeric) data unpredictable results will occur. The computer will not inform the programmer that any problem has occurred.

When passing character data from COBOL to FORTRAN the programmer must be aware of the alignment of the data and the number of words being passed. FORTRAN demands that the data begin on a full-word boundary; therefore, the programmer must insure that his COBOL data begins on a full-word (see paragraphs

5.2.3.3.3 and 5.2.3.4.1). FORTRAN also requires the use of complete words. The COBOL data should use complete words also. If the COBOL data has an incomplete word the programmer should pad (paragraph 5.2.3.4.1) the data so that a complete word is used.

For example, if the following appeared in COBOL:

```
Ø1  CHAR-ITEM.
    Ø3  CHAR PIC X(10).
    Ø3  FILLER PIC XX VALUE SPACES.
```

CHAR-ITEM could be passed to FORTRAN since it begins on a full-word boundary (insured by the Ø1 level), and contains 3 full-words (10 characters +2 characters = 12 characters).

The FORTRAN program which is to receive the character data must allow for at least as many words of storage in the FORTRAN program as is being sent by the COBOL programmer. Each single-precision variable contains 1 word per element. The variable must be dimensioned to contain as many elements as there are words being sent from COBOL. For example, if the COBOL program is passing 3 words of storage then the FORTRAN single-precision variable must be DIMENSIONed to at least 3. If double-precision (two words) variables are used the variable must be dimensioned to at least 1/2 of the number of COBOL words being passed if the number of words is even, or 1 word + 1/2 the number of words if the number is odd.

For example:

```
COBOL:
   Ø1  TABLE.
       Ø3  CHAR PIC X(21).
       Ø3  FILLER PIC X(3) VALUE SPACES.
       .
       .
       .
       CALL 'FORTSB' USING TABLE.

FORTRAN:
   SUBROUTINE FORTSB (A)
   DIMENSION A(6)
```

would be valid. TABLE begins on a full-word boundary and contains 6 full-words (when the FILLER is supplied). A is dimensioned to 6, so 6 words of storage is allocated.

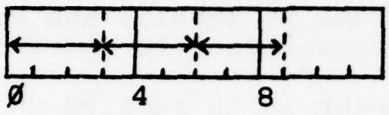
After the CALL is completed the first element of A would contain the first 4 characters of CHAR, the second element the next 4, and so on. The last element of A would contain the last character of CHAR, plus the 3 FILLER blanks.

It is important to realize the manner in which COBOL stores the character data. For example,

```
Ø1  ITEMS PIC XXX OCCURS 3 TIMES.
```

contains 9 characters in succession.

In storage this appears as:



Assuming the first 9 letters of the alphabet were stored into ITEMS, and the following FORTRAN subroutine is called

```
SUBROUTINE FORT (K)
DIMENSION K(3)
```

then K(1) contains ABCD, K(2) contains EFGH, and K(3) contains I plus three garbage characters. If the programmer actually wanted the COBOL data moved on an element-by-element basis (i.e., K(1) containing ABC, K(2) DEF, and K(3) GHI) then the COBOL data could be padded as follows:

```
Ø1  GROUP-ITEM.
    Ø3  SUB-GROUP OCCURS 3 TIMES.
        Ø5  ITEMS PIC XXX.
        Ø5  FILLER PIC X VALUE SPACES.
```

This would pad each element with a blank so that each word would contain 3 characters and 1 blank.

When passing FORTRAN character data to COBOL care must again be used. The COBOL data must be defined in the LINKAGE SECTION to accommodate the number of words being passed from FORTRAN. If the FORTRAN program contains a variable containing character data, and the variable is dimensioned to 6, then the COBOL must provide 6 words of storage. In addition, the data must begin on a full-word. For example:

```
Ø1  VARIABLE PIC X(24).
```

would begin on a full-word, and contain 6 words. VARIABLE would

be acceptable in the COBOL program to recieve the data from FORTRAN.

If the COBOL LINKAGE SECTION contained:

```
Ø1  TABLE.
    Ø3  ITEM-1 PIC XX.
    Ø3  ITEM-2 PIC X(4).
    Ø3  ITEM-3 PIC X(15)
    Ø3  ITEM-4 PIC X.
    Ø3  ITEM-5 PIC XX.
    .
    .
    .
    PROCEDURE DIVISION USING TABLE.
```

again TABLE would begin on a full-word boundary and would contain 6 words (2+4+15+1+2 = 24 characters = 6 words). TABLE would also be acceptable in the COBOL program to receive the FORTRAN data. Using would result in the first two characters passed from FORTRAN being in ITEM-1, the next 4 characters in ITEM-2, the next 15 in ITEM-3, and so on.

5.2.3.4.3 Passing Arrays. Arrays may be passed between COBOL and FORTRAN, and FORTRAN and COBOL. One dimensional arrays (e.g., arrays of length n) do not pose as many problems as multi-dimensional arrays.

As with all items being passed between languages word alignment and length must be the same in both languages. For example, assume the following double precision one dimensional array (length 5) in FORTRAN is to be passed to COBOL:

```
DOUBLE PRECISION A
DIMENSION A(5)
CALL COBSUB (A)
    .
    .
    .
```

Each element of A is two words in length, aligned on a double word boundary (paragraph 5.2.2.4). In addition A contains 5 elements. The corresponding IBM COBOL data type is COMP-2 (Figure 5-4). The COBOL could contain

```
LINKAGE SECTION.
Ø1  ITEM.
    Ø3  NUMBERS COMP-2 OCCURS 5 TIMES.
.
.
.
PROCEDURE DIVISION USING ITEM.
.
.
.
```

ITEM begins on a double word boundary (paragraph 5.2.3.3.1). Each element is two words in length (paragraph 5.2.1.4) and there are 5 elements. The FORTRAN CALL statement could correctly pass the FORTRAN array A into the COBOL table (array) ITEM. A(1) would be stored in NUMBERS (1), A(2), in NUMBERS (2), etc.

Passing a two dimensional array between FORTRAN and COBOL, or COBOL and FORTRAN has an additional factor of which the programmer must be aware. A FORTRAN column is a COBOL row, and a FORTRAN row is a COBOL column.

Attachment 3 contains an example of this problem. The FORTRAN driver defines I as a 3 x 4 array. This array is passed to the COBOL subroutine COBSUB. COBSUB contains:

```
LINKAGE SECTION.
Ø1  ARRAY-TABLE.
    Ø3  MATRIX OCCURS 4 TIMES.
        Ø5  COLUMNS PIC X(6) COMP OCCURS 3 TIMES.
```

This defines the COBOL ARRAY-TABLE as a 4 x 3 array.

In a 3 dimensional array the first and last dimensions are reversed.

5.2.3.4.4 Passing COBOL Group Items. COBOL has the capability to define an item (group item) which is in turn divided into other items. These subdivisions may have different PICs and USAGEs. For example:

```
Ø1  GROUP-ITEM.
    Ø3  ITEM-1 PIC S9(5) COMP.
    Ø3  ITEM-2 PIC XXXX.
    Ø3  ITEM-3 COMP-2.
```

GROUP-ITEM is the name of the group item. This group item is subdivided into ITEM-1 (integer, full-word), ITEM-2 (display, full-word), and ITEM-3 (floating-point double precision, double word).

FORTRAN does not have the equivalent to a group-item. Each element in a FORTRAN data item is assumed by the compiler to be of the same data type. For example, if the above COBOL program calls a FORTRAN subroutine using

```
CALL 'FORT' USING GROUP-ITEM.
```

and the FORTRAN subroutine is as follows:

```
SUBROUTINE FORT (ARRAY)
DIMENSION ARRAY(4)
```

ARRAY is a floating-point data item containing 4 elements. After the CALL, ARRAY(1) would contain ITEM-1, ARRAY(2) ITEM-2. ARRAY(3) and ARRAY(4) would contain the two words found in ITEM-3. The number in ARRAY(1) would be in fixed point representation. However, FORTRAN would assume the number represented was in floating-point. ARRAY(2) would contain the character string found in ITEM-2; FORTRAN would again assume the representation was floating. ARRAY(3) would contain the first half of a double word floating-point number. FORTRAN would assume this is a single word floating-point numer. ARRAY(4) would contain the second word of the double word ITEM-3. Again, FORTRAN would assume this is a floating-point number.

In many cases it is possible to pass data in this manner and handle the FORTRAN data correctly. This process is lengthy, and confusing. Because of this confusion, an alternate method of passing group items is encouraged.

If GROUP-ITEM is again defined as above, the following CALL could be used:

```
CALL 'FORT2' USING ITEM-1, ITEM-2, ITEM-3.
```

The FORTRAN routine would now appear as:

```
SUBROUTINE (I,A,B)
DOUBLE PRECISION B
```

ITEM-1 (integer, single word) would be passed to I (integer, single word), ITEM-2 (character, single word) to A (character, single word) (see paragraph 5.2.3.4.2), and ITEM-3 (floating-point, double word) to B (floating-point double word). (5, 11, 12, 14, 15, 16, 17)

5.3 The Honeywell Computer. The Honeywell computer uses words composed of 6 bytes (see Section 4 for a more complete discussion). Each byte contains six bits. The addressing scheme on the Honeywell computer is to sequentially number each word (recall words are numbered beginning with Ø). Thus, the address of the first word is Ø, the address of the second word is 1, the address of the third word is 2, etc. In addition to using a full-word (6 bytes) of storage, the Honeywell computer can also use double words (12 bytes). The word length the computer uses depends upon which data type is being used. However, when passing data from one language to another it becomes critical to know on which word boundary data are aligned. A boundary is a double word boundary if its address is divisible by 2.

5.3.1 Honeywell COBOL. Following are the data types found in Honeywell COBOL and the word alignment used.

5.3.1.1 COMP. The COMP usage is for floating-point data representation. If the PICTURE clause contains 1-8 digits one full-word of storage will be used. Alignment is on a full-word boundary. If the PICTURE clause contains 9-18 digits a double word will be used. Alignment is on a double word boundary.

5.3.1.2 COMP-1. The COMP-1 usage is for fixed-point data representation. If the PICTURE clause contains 1-8 digits a full-word of storage will be used. Alignment is on a full-word boundary. If the PICTURE clause contains 9-18 digits a double word will be used on a double word boundary.

5.3.1.3 COMP-2. The COMP-2 usage is for floating-point data representation. If the PICTURE clause contains 1-8 digits a single-precision floating-point number will be stored in one full-word, aligned on a full-word. If the PICTURE clause contains 9-18 digits a double-precision floating-point number will be stored in a double word, aligned on a double word.

5.3.1.4 COMP-3. The COMP-3 usage is used for fixed point data representation. One full-word of storage is used, with alignment on a full-word.

5.3.1.5 COMP-3 PACKED SYNCHRONIZED or COMP-4. The COMP-3 PACKED SYNCHRONIZED or COMP-4 usage is used for packed decimal format representation.

5.3.1.6 DISPLAY. The DISPLAY usage results in data being represented in character format. The data may begin on any byte and continue for as many bytes as necessary. Unless COMP, COMP-1, COMP-2, COMP-3, or COMP-4 is specified the default will be DISPLAY, even if the PICTURE clause is numeric. For example, if the following is in a COBOL program:

```
77  ITEM PICTURE S99 VALUE +12.
```

the character representation of the number +12 will be stored in ITEM.

5.3.2 Honeywell FORTRAN. Following are the data types found in Honeywell FORTRAN and the word alignment used.

5.3.2.1 REAL. REAL data items are used for single-precision floating-point data representation. One full-word of storage is used, aligned on a full-word boundary.

5.3.2.2 DOUBLE PRECISION. DOUBLE PRECISION (or REAL*9) data items are used for double-precision floating-point data representation. One double word of storage is used, aligned on a double word boundary.

5.3.2.3 INTEGER. INTEGER data items are used for fixed point data representation. One full-word of storage is used, aligned on a full-word of storage.

5.3.2.4 CHARACTER. CHARACTER data items are used for character data representation. The form of this statement is

```
CHARACTER item*S, item*S...
```

where item is the variable name, and *S is the number of characters in the item. If each item is of the same length, the statement may be written as

```
CHARACTER*S item, item
```

The number of bytes of storage used is equal to the number of characters defined by *S.

5.3.3 Honeywell COBOL/FORTRAN. Programs written in one language calling routines in another language, be it COBOL calling FORTRAN or FORTRAN calling COBOL, require special care. Great attention must be taken to insure data items are compatible, and that word boundaries are correct. Below are some considerations to use when interfacing Honeywell COBOL and Honeywell FORTRAN.

5.3.3.1 Data Type Compatibility. In order for the computer to interpret the data correctly in both the calling program and the called subroutine, the data being passed must be consistently defined in both programs. Figure 5-1 shows the COBOL data types and their equivalent FORTRAN types. Figure 5-2 shows the FORTRAN data types and their equivalent COBOL types. For example, if a FORTRAN program is to pass a value defined to be INTEGER in FORTRAN, the corresponding data item in the COBOL routine must be defined as COMP-1 or COMP-3.

5.3.3.2 Incompatible Data Types. As can be seen from Figures 5-1 and 5-2, data types exist in one language for which no corresponding data types exist in the other language. For example, a COMP-1 usage in COBOL defines a fixed point data representation. If the PICTURE clause contains 9-18 digits, a double word of storage will be used (see paragraph 5.3.1.2). FORTRAN, however, only can represent a fixed point number in a full-word, not a double word (see paragraph 5.3.2.3). Therefore, if a COBOL program passes a COMP-1 value with a PICTURE containing 9 or more digits to FORTRAN, the FORTRAN program will be unable to handle the passed data correctly. However, it should be noted that when inconsistent data items are passed from language to language the computer will not issue an error statement. The computer will continue to operate on

this incorrectly defined data, assuming this is exactly what the programmer had originally desired. It is, therefore, entirely up to the programmer to insure data items are defined the same in both the calling and the called programs.

5.3.3.3 Word Boundary Alignment. The programmer is also responsible for insuring correct word boundary alignment on data items being passed from one language to another. Word boundary alignment may be the single most critical, and possibly the most difficult, consideration when passing data between two languages. It is extremely important to insure that the word boundary of a particular data item matches the word boundary expected in the called routine. Following is a discussion of word boundary alignment.

5.3.3.3.1 Word Alignment. Paragraph 5.3.1 describes the Honeywell COBOL data types and the word boundaries on which these data types are aligned. Paragraph 5.3.2 describes the FORTRAN data types and alignment. One must keep these facts in mind when passing data from one language to another. Also, it is important to realize that COBOL begins Ø1 levels on a double word boundary. Using this information, it is possible to determine where word boundaries of data items will fall.

5.3.3.3.2 Data Passing. Data is not actually passed from one routine to another. In reality, the item being passed is the address of the beginning of the data item. In FORTRAN this address would be the beginning of the data items represented by a variable name. In COBOL this address could be the beginning of an item, table, group item, etc. For example if FORTRAN is passing a 3 x 4 array named ARRAY, the address of the beginning of ARRAY is passed. It is up to the receiving program to realize that this address represents the beginning of a 3 x 4 array. No information stating the size, dimension, etc, of the data item is passed.

5.3.3.3.3 Internal Storage of Data. Data is stored in the computer's memory on a sequential basis. For example, assume the following appears in a COBOL program:

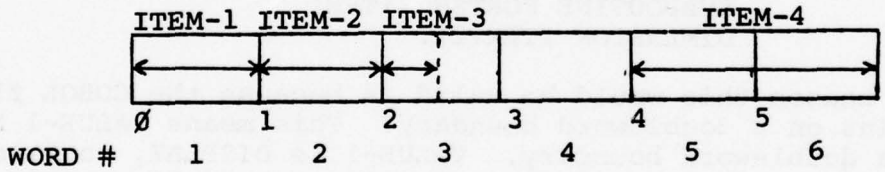
```
77  VALUE-1 PIC 99 VALUE 1.
77  VALUE-2 PIC 99 VALUE 2.
```

As the COBOL program is compiled internal storage will be allocated for both VALUE-1 and VALUE-2. Since these two items are defined sequentially in the program the two items will be stored sequentially in the computer. Therefore, VALUE-1 will be immediately followed by VALUE-2. This is true of each data item described in a COBOL program, and is also true of each data item in FORTRAN.

5.3.3.3.4 Honeywell COBOL Alignment. The easiest way to explain the manner in which COBOL aligns data may be to use the following example:

```
Ø1  TABLE.
    Ø3  ITEM-1 PIC X(6).
    Ø3  ITEM-2 PIC S9(5) COMP-1.
    Ø3  ITEM-3 PIC XXX.
    Ø3  ITEM-4 PIC S9(10) COMP-2.
```

In this example TABLE begins on a double word boundary because TABLE is a Ø1 level (paragraph 5.3.3.3.1). Since TABLE represents the collection of data items grouped beneath it, and not an individual data item, ITEM-1 begins at the same location as TABLE, i.e., a double word boundary. ITEM-1 is DISPLAY (paragraph 5.3.1.6), therefore each digit represents 1 byte. ITEM-1 contains 6 bytes (1 word). ITEM-2 begins immediately following ITEM-1; therefore ITEM-2 begins on a full-word boundary. ITEM-2 is COMP-1 (paragraph 5.3.1.2) with a PIC of S9(5) which means ITEM-2 occupies 1 word of storage. ITEM-3 is DISPLAY using 3 bytes (one half word) of storage. This means ITEM-3 ends in the middle of a word. ITEM-4 begins following ITEM-3. However, ITEM-4 is COMP-2 (paragraph 5.3.1.3) with a PIC of S9(10). This requires that ITEM-4 begin on a double word boundary. The compiler will skip the appropriate number of bytes necessary so that ITEM-4 can begin on the next double word boundary. See the following diagram for a visual description of TABLE:



The unused bytes in words 3 and 4 are fillers. The compiler provided these bytes to force ITEM-4 to begin on a double word boundary. Unless the programmer is familiar with the manner in which the compiler works, the programmer will be unaware of these extra bytes.

5.3.3.3.5 Honeywell FORTRAN Alignment. FORTRAN data are aligned as described in paragraph 5.3.2. In an array, each element is aligned on the same type of boundary. FORTRAN does not allow group items as COBOL does. Because of this fact extra bytes do not present the problem in FORTRAN as they do in COBOL.

5.3.3.4 Language Interfacing. Interfacing FORTRAN and COBOL presents some problems. Some of the more common problems, and their solutions, are discussed below.

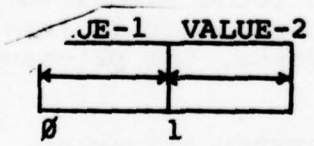
5.3.3.4.1 Padding. Just as it is necessary for the compiler to insert fillers (paragraph 5.3.3.3.4) for proper word boundary alignment in internal storage, it may be necessary for the programmer to insert fillers in the data structure for proper alignment. This becomes increasingly necessary when interfacing two languages such as FORTRAN and COBOL. Again it is very important for the programmer to realize where data alignment will occur.

For example, assume a COBOL program is to pass a group item to FORTRAN. This group item consists of VALUE-1 and VALUE-2, each PIC X(6). If the FORTRAN routine is to receive the COBOL values in a two element array, with the first element of the array containing VALUE-1, and the second element containing VALUE-2, then the following would be correct:

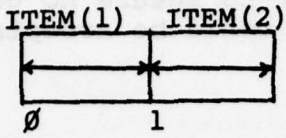
```
COBOL:
    Ø1  TABLE.
        Ø3  VALUE-1 PIC X(6).
        Ø3  VALUE-2 PIC X(6).
        .
        .
        .
        CALL FORTSB USING TABLE.

FORTRAN:
    SUBROUTINE FORTSB (ITEM)
    DIMENSION ITEM(2).
```

The reason this would be valid is because the COBOL Ø1 level begins on a doubleword boundary. This means VALUE-1 begins on a doubleword boundary. VALUE-1 is DISPLAY, containing 6 characters (bytes) and is, therefore, one full-word in length. VALUE-2 begins immediately following VALUE-1. This means VALUE-2 begins on a full-word boundary. Storage looks as follows:



The FORTRAN routine has ITEM dimensioned to 2. Each element of ITEM (by default) is 1 word in length, beginning on a full-word boundary. The FORTRAN storage appears as:

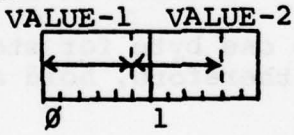


The COBOL storage begins on a double-word boundary; the FORTRAN on a full-word. Since a doubleword boundary is also a full-word boundary, the FORTRAN representation is compatible with the COBOL. The six characters of VALUE-1 will be in ITEM(1), the 6 in VALUE-2 will be in ITEM(2).

Now, assume that the items in the COBOL program only contain 5 characters. The COBOL would appear as:

```
Ø1  TABLE.
    Ø3  VALUE-1 PIC XXXXX.
    Ø3  VALUE-2 PIC XXXXX.
    .
    .
    .
    CALL FORTSB USING TABLE.
```

TABLE would still begin on a double-word boundary. However, VALUE-1 and VALUE-2 would each contain 5 bytes, and be stored sequentially as:



Notice the first byte of VALUE-2 appears as the last byte in word 1. When FORTRAN is called only the address of the beginning of TABLE (address Ø in this case) is passed. FORTRAN goes to this address and assumes storage is as FORTRAN expects it (as described above).

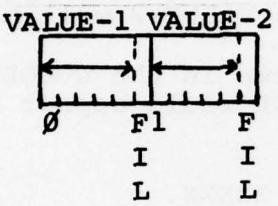
If VALUE-1 contained the characters ABCDE, and VALUE-2 the characters FGHIJ, after the call to FORTRAN ITEM(1) would contain the characters ABCDEF and ITEM-2 would contain GHIJØØ, where ØØ may be blanks, or garbage characters.

In order for FORTRAN for receive the data properly (i.e., ITEM(1) containing ABCDE and ITEM(2) containing FGHIJ) padding in the form of a FILLER must be used. One word of storage contains 6 bytes, or 6 characters. VALUE-1 and VALUE-2 contain

5 characters each. A FILLER of one character is necessary to complete the word. In addition, this FILLER should have a value of a blank in order to insure no garbage data is introduced. The following would be acceptable in the COBOL program:

```
Ø1  TABLE.
    Ø3  VALUE-1 PIC XXXXX.
    Ø3  FILLER PIC X VALUE SPACES.
    Ø3  VALUE-2 PIC XXXXX.
    Ø3  FILLER PIC X VALUE SPACES.
```

The COBOL storage would be:



Notice that VALUE-2 now begins in the second word. If the FORTRAN routine is now called using this TABLE, ITEM(1) would contain ABCDE, and ITEM(2) FGHIJ.

5.3.3.4.2 Passing Character Strings. Passing a character string between COBOL and FORTRAN presents some unique problems. Paragraph 5.3.3.4.1 used this problem as an example; however, the problem is more complex than this example.

One character requires one byte for storage. One word contains 6 bytes and can, therefore, hold a maximum of 6 characters.

Recall any COBOL data item which does not have a USAGE of COMP, COMP-1, COMP-2, COMP-3, or COMP-4 will automatically be considered DISPLAY (paragraph 5.3.1.6). USAGE DISPLAY means the data is stored as character information.

To demonstrate the number of words required assume the following:

```
Ø1  ITEM-1 PIC X(3).
Ø1  ITEM-2 PIC X(7).
```

ITEM-1 would require 3 bytes of storage since PIC X(3) defines three characters of data. ITEM-2 contains 7 characters. This would require 7 bytes of storage: one full-word plus one additional byte.

Honeywell FORTRAN contains a CHARACTER data type statement which is equivalent to the COBOL DISPLAY (see Figure 5-1). This informs the compiler that character data will be stored in the FORTRAN data element. In addition, it specifies the number of characters per element (see paragraph 5.3.2.4).

When passing character strings from Honeywell COBOL to Honeywell FORTRAN, or from FORTRAN to COBOL, the programmer must insure that word boundaries and string length are compatible in both languages.

For example, if a 9 character string in FORTRAN is to be passed to a COBOL subroutine then the following would be valid in FORTRAN:

```
CHARACTER*9 ITEM
CALL COBSUB (ITEM)
.
.
.
```

The COBOL subroutine COBSUB would have to define the receiving data item to begin on a full-word boundary, with a length of 9 characters (minimum). The following would be one way to accomplish this:

```
.
.
.
Ø1  COBOL-ITEM PIC X(9).
.
.
.
PROCEDURE DIVISION USING COBOL-ITEM.
```

Multi-dimensional character arrays (see attachment 8 for an example) present some additional problems. (See paragraph 6.5.5 for an additional discussion of multi-dimensional character arrays). Each element of the COBOL array is considered to have the number of characters defined in the PIC clause. In FORTRAN each element contains the number of characters specified on the CHARACTER statement. For example, in COBOL:

```
Ø1  ARRAY.
    Ø3  CHARS PIC XXX OCCURS 4 TIMES.
```

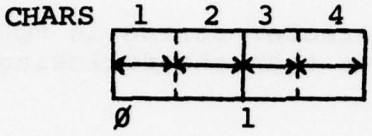
each element of ARRAY has 3 characters. In FORTRAN

```
DIMENSION LIST(4)
CHARACTER*3 LIST
```

each element of LIST contains 3 characters.

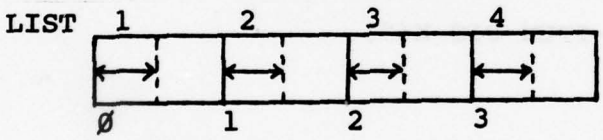
A problem arises in the different manner in which COBOL and FORTRAN interpret their size specifications. To explain this difference, assume ARRAY and LIST are still defined as above.

COBOL views ARRAY AS:



ARRAY begins at address Ø, and continues for 12 bytes (characters). CHARS (1) is in the left half of the first word, CHARS (2) is in the right half of word 1, etc. COBOL views characters as a string, disregarding intervening word boundaries, regardless of how many characters are defined in the PIC clause.

FORTRAN views the storage in a different manner. If the size defined in the CHARACTER statement is 6 or less, each element begins on a full-word boundary. However, only the number of bytes defined in the size statement are used in each word. For LIST, storage would appear as:



Notice that FORTRAN uses 4 words, COBOL uses 2 words.

If the CHARACTER statement defines more than 6 characters (bytes), FORTRAN uses this as the number of bytes per element. Again, if the character string of an element does not end on a full-word boundary the remaining bytes in the word are ignored.

In order for COBOL to pass data to FORTRAN, or FORTRAN to COBOL, the COBOL data must be defined (by use of fillers) to match the FORTRAN data.

In order to pass CHARS (1) to LIST (1), CHARS (2) to LIST (2), etc, the COBOL would need to be padded as follows:

```
Ø1  ARRAY.
    Ø5  ITEM OCCURS 4 TIMES.
    Ø5  CHARS PIC XXX.
    Ø3  FILLER PIC XXX.
```

5.3.3.4.3 Passing Arrays. Arrays may be passed between COBOL and FORTRAN, and FORTRAN and COBOL. One dimensional arrays (e.g., arrays of length n) do not pose as many problems as multi-dimensional arrays.

As with all items being passed between languages word alignment and length must be the same in both languages. For example, assume the following double precision one dimensional array (length 5) in FORTRAN is to be passed to COBOL:

```
DOUBLE PRECISION A
DIMENSION A(5)
CALL COBSUB (A)
      .
      .
      .
```

Each element of A is two words in length, aligned on a double word boundary (paragraph 5.3.2.2). In addition A contains 5 elements. The corresponding Honeywell COBOL data type is COMP-2 (Figure 5-2). The COBOL could contain

```
Ø1  ITEM.
    Ø3  NUMBERS PIC S9(10) COMP-2 OCCURS 5 TIMES.
    .
    .
    .
    PROCEDURE DIVISION USING ITEM.
    .
    .
    .
```

ITEM begins on a double word boundary (paragraph 5.3.3.3.1). Each element is two words in length (paragraph 5.3.1.3) and there are 5 elements. The FORTRAN CALL statement would correctly pass the FORTRAN array A into the COBOL table (array) ITEM. A(1) would be stored in NUMBERS (1), A(2), in NUMBERS (2), etc.

Passing a two dimensional array between FORTRAN and COBOL, or COBOL and FORTRAN has an additional factor of which the programmer must be aware. A FORTRAN column is a COBOL row, and a FORTRAN row is a COBOL column.

Attachment 3 contains an IBM example of this problem. (The problem is the same in IBM as Honeywell). The FORTRAN driver defines I as a 3 x 4 array. This array is passed to the COBOL subroutine COBSUB. COBSUB contains:

```
Ø1  ARRAY-TABLE.
    Ø3  MATRIX OCCURS 4 TIMES.
        Ø5  COLUMNS PIC 9(6) COMP OCCURS 3 TIMES.
```

This defines the COBOL ARRAY-TABLE as a 4 x 3 array.

In a 3 dimensional array the first and last dimensions are reversed.

5.3.3.4.4 Passing COBOL Group Items. COBOL has the capability to define an item (group item) which is in turn divided into other items. These subdivisions may have different PICs and USAGEs. For example:

```
Ø1  GROUP-ITEM.
    Ø3  ITEM-1 PIC S9(5) COMP-1.
    Ø3  ITEM-2 PIC XXXXXX.
    Ø3  ITEM-3 PIC S9(10) COMP-2.
```

GROUP-ITEM is the name of the group item. This group item is subdivided into ITEM-1 (integer, full-word), ITEM-2 (character, full-word), and ITEM-3 (floating point double precision, double word).

FORTRAN does not have the equivalent to a group-item. Each element in a FORTRAN data item is assumed by the compiler to be of the same data type. For example, if the above COBOL example calls a FORTRAN subroutine using

```
CALL FORT USING GROUP-ITEM.
```

and the FORTRAN subroutine is as follows:

```
SUBROUTINE FORT (ARRAY)
DIMENSION ARRAY(4)
```

ARRAY is a floating point data item containing 4 elements. After the CALL, ARRAY(1) would contain ITEM-1, ARRAY(2) ITEM-2. ARRAY(3) and ARRAY(4) would contain the two words found in ITEM-3. The number in ARRAY(1) would be in fixed point representation; however FORTRAN would assume the number represented was in floating point. ARRAY(2) would contain the character string found in ITEM-2; FORTRAN would again assume the representation was floating. ARRAY(3) would contain the first half of a double word floating point number. FORTRAN would assume this is a single word floating point number. ARRAY(4) would contain the second word of the doubleword ITEM-3. Again, FORTRAN would assume this is a floating-point number.

In many cases it is possible to pass data in this manner and handle the FORTRAN data correctly. This process is lengthy, and confusing. Because of this an alternate method of passing group items is encouraged.

If GROUP-ITEM is again defined as above the following CALL could be used:

```
CALL FORT2 USING ITEM-1, ITEM-2, ITEM-3.
```

The FORTRAN routine would now appear as:

```
SUBROUTINE (I, A, B)
CHARACTER*6 A
DOUBLE PRECISION B
```

ITEM-1 (integer, single word) would be passed to I (integer, single word), ITEM-2 (character, single word) to A (character, single word), and ITEM-3 (floating point, double-word) to B (floating point, double-word). (6, 7, 10)

5.4 Honeywell FORTRAN/COBOL. Figure 5-1 lists the Honeywell COBOL data types, and their equivalent Honeywell FORTRAN types. Figure 5-2 lists the Honeywell FORTRAN data types and their equivalent Honeywell COBOL data types.

5.5 IBM FORTRAN/COBOL. Figure 5-3 lists the IBM COBOL data types, and their equivalent IBM FORTRAN types. Figure 5-4 lists the IBM FORTRAN data types and their equivalent IBM COBOL data types.

5.6 IBM/Honeywell Equivalents. Despite the multitude of differences between languages and computers, there are elements of each language which will transfer from one machine to the other without modification. These elements (verbs and statements) hold the same meaning, structure, and use on both the IBM and Honeywell machines.

5.6.1 COBOL. The following list contains the COBOL verbs found to be equivalent between the Honeywell and IBM machines.

ACCEPT	DISPLAY	MERGE	SEARCH
ADD	DIVIDE	MOVE	SET
ALTER	EXIT	MULTIPLY	SORT
CALL	GENERATE	OPEN	STOP
COPY	GO TO	PERFORM	SUBTRACT
CLOSE	IF	READ	TERMINATE
COMPUTE	INITIATE	RELEASE	WRITE

5.6.2 FORTRAN. The following list contains the FORTRAN statements found to be equivalent between the Honeywell and IBM machines. (6, 7, 10, 12, 14, 15, 16, 17)

ASSIGN	DATA	EQUIVALENCE	INTEGER	REAL
BACKSPACE	DIMENSION	EXTERNAL	LOGICAL	RETURN
BLOCK DATA	DO	FORMAT	NAMELIST	REWIND
CALL	DOUBLE PRECISION	FUNCTION	PAUSE	STOP
COMMON	END	GO TO	PRINT	SUBROUTINE
COMPLEX	ENDFILE	IF	PUNCH	WRITE
CONTINUE	ENTRY	IMPLICIT	READ	

COBOL	ALIGNMENT	# OF WORDS	FORTRAN EQUIVALENT	ALIGNMENT	# OF WORDS
COMP					
1-8 digits	full-word	1	REAL	full-word	1
9-18 digits	double-word	2	DOUBLE PRECISION	double-word	2
COMP-1					
1-8 digits	full-word	1	INTEGER	full-word	1
9-18 digits	double-word	2	None		
COMP-2					
1-8 digits	full-word	1	REAL	full-word	1
9-18 digits	double-word	2	DOUBLE PRECISION	double-word	2
COMP-3					
1-10 digits	full-word	1	INTEGER	full-word	1
COMP-3 PACKED SYNCHRONIZED	full-word	1	None		
COMP-4	full-word	1	None		
DISPLAY	any byte	-	CHARACTER	full-word	-

HONEYWELL COBOL TO FORTRAN DATA TYPES

FIGURE 5-1

FORTRAN	ALIGNMENT	# OF WORDS	COBOL EQUIVALENT	ALIGNMENT	# OF WORDS
REAL	full-word	1	COMP 1-8 digits or COMP-2 1-8 digits	full-word full-word	1 1
REAL*9 or DOUBLE PRECISION	double-word	2	COMP 9-18 digits or COMP-2 9-18 digits	double-word double-word	2 2
INTEGER	full-word	1	COMP-1 1-8 digits	full-word	1
CHARACTER	full-word	-	DISPLAY	any-byte	-
COMPLEX	double-word	2	None		
LOGICAL	full-word	1	None		

HONEYWELL FORTRAN TO COBOL DATA TYPES

FIGURE 5-2

COBOL	ALIGNMENT	# OF WORDS	FORTRAN EQUIVALENT	ALIGNMENT	# OF WORDS
COMP SYNC					
1-4 digits	half-word	½	INTEGER*2	half-word	½
5-9 digits	full-word	1	INTEGER or	full-word	1
			INTEGER*4	full-word	1
10-18 digits	double-word	2	None		
COMP-1 SYNC	full-word	1	REAL or REAL*4	full-word	1
COMP-2 SYNC	double-word	2	REAL*8 or	double-word	2
			DOUBLE PRECISION	double-word	2
COMP-3	-	-	None		
DISPLAY	any-byte	-	None		

IBM COBOL TO FORTRAN DATA TYPES

FIGURE 5-3

FORTRAN	ALIGNMENT	# OF WORDS	COBOL EQUIVALENT	ALIGNMENT	# OF WORDS
INTEGER*2	half-word	½	COMP SYNC 1-4 digits	half-word	½
INTEGER or INTEGER*4	full-word full-word	1 1	COMP SYNC 5-9 digits	full-word	1
REAL or REAL*4	full-word full-word	1 1	COMP-1 SYNC	full-word	1
REAL*8 or DOUBLE PRECISION	double-word double-word	2 2	COMP-2 SYNC	double-word	2
COMPLEX or COMPLEX*8	double-word double-word	2 2	None		
COMPLEX*16	double-word	4	None		
LOGICAL*1	half-word	½	None		
LOGICAL or LOGICAL*4	full-word full-word	1 1	None		

IBM FORTRAN TO COBOL DATA TYPES

FIGURE 5-4

HONEYWELL	ALIGNMENT	# OF WORDS	IBM	ALIGNMENT	# OF WORDS
COMP (1-8 digits)	full-word	1	COMP-1 SYNC	full-word	1
COMP (9-18 digits)	double-word	2	COMP-2 SYNC	double-word	2
COMP-1 (1-8 digits)	full-word	1	COMP SYNC (5-9 digits)	full-word	1
COMP-1 (9-18 digits)	double-word	2	COMP SYNC (10-18 digits)	double-word	2
COMP-2 (1-8 digits)	full-word	1	COMP-1 SYNC	full-word	1
COMP-2 (9-18 digits)	double-word	2	COMP-2 SYNC	double-word	2
COMP-3 (1-10 digits)	full-word	1	COMP SYNC (5-9 digits)	full-word	1
COMP-3 PACKED SYNCHRONIZED	full-word	-	COMP-3	-	-
COMP-4	full-word	-	COMP-3	-	-
DISPLAY	any-byte	-	DISPLAY	any-byte	-

HONEYWELL TO IBM COBOL

FIGURE 5-5

IBM	ALIGNMENT	# OF WORDS	HONEYWELL	ALIGNMENT	# OF WORDS
COMP SYNC (1-4 digits)	half-word	½	None		
COMP SYNC (5-9 digits)	full-word	1	COMP-1 (1-8 digits)	full-word	1
COMP SYNC (10-18 digits)	double-word	2	COMP-1 (9-18 digits)	double-word	2
COMP-1	full-word	1	COMP (1-8 digits) COMP-2 (1-8 digits)	full-word full-word	1 1
COMP-2	double-word	2	COMP (9-18 digits) COMP-2 (9-18 digits)	double-word double-word	2 2
COMP-3	-	-	COMP-4	-	-
DISPLAY	any-byte	-	DISPLAY	any-byte	-

IBM TO HONEYWELL COBOL

FIGURE 5-6

HONEYWELL	ALIGNMENT	# OF WORDS	IBM	ALIGNMENT	# OF WORDS
REAL	full-word	1	REAL	full-word	1
DOUBLE PRECISION	double-word	2	DOUBLE PRECISION	double-word	2
INTEGER	full-word	1	INTEGER	full-word	1
CHARACTER	any-byte	-	None		
COMPLEX	double-word	2	COMPLEX	double-word	2
LOGICAL	full-word	1	LOGICAL	full-word	1

HONEYWELL TO IBM FORTRAN

FIGURE 5-7

IBM	ALIGNMENT	# OF WORDS	HONEYWELL	ALIGNMENT	# OF WORDS
INTEGER*2	half-word	½	None		
INTEGER	full-word	1	INTEGER	full-word	1
REAL	full-word	1	REAL	full-word	1
DOUBLE PRECISION	double-word	2	DOUBLE PRECISION	double-word	2
COMPLEX	double-word	2	COMPLEX	double-word	2
COMPLEX*16	double-word	4	None		
LOGICAL*1	half-word	½	None		
LOGICAL	full-word	1	LOGICAL	full-word	1

IBM TO HONEYWELL FORTRAN

FIGURE 5-8

SECTION 6. LINKAGE

6.1 Introduction. In order for two languages to execute together as a single program certain changes must be made to the subprograms. These changes vary between the IBM and Honeywell computers, and language to language. Each computer and language will be considered separately.

6.2 Honeywell COBOL Program Linkage. The linkage of a COBOL driver and COBOL subroutine on the Honeywell machine can be accomplished in two ways. The first is the overlay and is the most complex. The second is through the use of the CALL statement.

6.2.1 Link Overlay. Because of program size and complexity, it may be necessary to segment a program to make more efficient use of memory and available storage media. In this situation, a large program may be broken into smaller segments, called "LINKS". Those links used most often are organized such that they will reside in memory and the lesser used links are temporarily overlayed (in memory) when required by the driver or "main link". (4: 7-1)

Because the link overlay concept involves segmenting a single, large program and not necessarily linking separate programs, it is not within the scope of this report. Further details on the link overlay concept may be obtained from the Honeywell COBOL User's Guide.

6.2.2 The Call Statement. The CALL statement is normally used in a main or driver program to access one or more other programs, or subroutines to perform some activity and return to the calling program. While this is generally the case, a subroutine may call another subroutine, which may call still another, and so on. The general form of a COBOL CALL statement is:

```
.
.
.
CALL name.
.
.
.
```

where name is the PROGRAM-ID of the subroutine. When the CALL statement is executed, control is transferred to the called subroutine. When the subroutine has completed its processing, control is returned to the calling program by the EXIT statement. The EXIT must follow a paragraph name and would appear like this:

```
.
.
.
SUBROUTINE-EXIT.
        EXIT SUBROUTINE-NAME.
```

6.2.3 Arguments. It is often necessary for the called routine to use some data found in the main program. Through the use of the USING clause in the CALL statement, these data items, called "ARGUMENTS" may be passed to the called routine in the following manner:

```
.
.
.
CALL name USING ZAP, BIFF, POW.
.
.
.
```

The addresses of ZAP, BIFF, and POW are passed to the subroutine and not their actual values. Hence, instead of actually passing the data, the using clause tells the subroutine where to find that data and the subroutine now has access to them.

6.2.4 Receiving the Arguments. While it is important to send (pass) information to a subroutine properly, it is equally important it be received properly. In a COBOL subroutine, the "receptacle" is the PROCEDURE DIVISION header, in this form:

```
.
.
.
PROCEDURE DIVISION USING ZAP, BIFF, POW.
.
.
.
```

where ZAP, BIFF, and POW have been defined in the WORKING-STORAGE SECTION of the subroutine. These variables now contain relevant data and may be used in the subroutine in all the same ways as in the main program.

It would be useful to point out when arguments are passed to another program, the subprogram is not required to utilize the same variable names. Because the USING clause only passes the addresses of the arguments, the programmer may find it useful to use other variable names in the subroutine. For example, if the main program used ZAP, BIFF, and SOC to represent the sine, cosine, and tangent of some angle, and these values were to be used in a subroutine, the sending and receiving could appear thus:

```
(MAIN)
.
.
CALL name USING ZAP, BIFF, SOC.
.
.
(SUBROUTINE) name
.
.
.
PROCEDURE DIVISION USING SIGN, COSIGN, TANGERINE.
```

where the values of ZAP, BIFF, and SOC are called SIGN, COSIGN, and TANGERINE in the subroutine. However, this convention may be more useful when interfacing different languages, which is addressed elsewhere in this report. It is also important to insure the number of arguments recieved match the number of arguments being passed. In the example above, three arguments are passed, and three are received. (See paragraph 6.3.2.2)

6.2.5 Tables and Arrays. For all practical purposes, a table and an array are the same thing: called a table in COBOL programs, a similar arrangement of data items in a FORTRAN program is called an array.

6.2.5.1 Basic Definitions. An array is composed of elementary data items having identical data descriptions.

A table may be composed of both elementary and group items having differing data descriptions. (7; 2-6)

6.2.5.2 Passing a Table. Tables and arrays may be used as arguments and passed back and forth between COBOL and FORTRAN routines. However, when passing a table or an array to a routine of a different language (COBOL to FORTRAN, for example) the intrinsic characteristics of the two languages involved come into play. See Section 6.5.5 and Section 8 for an example.

6.2.6 Entry at Other Than PROGRAM-ID. When calling a subroutine, the implied entry point into the subroutine is its' PROGRAM-ID, and processing continues through the end of the subroutine. However, if a subroutine is designed to accomplish many activities, the programmer may not wish all of them to be processed each time the subroutine is executed. Instead, only one or two of those activities may be desired. Therefore, it would be advantageous if one could enter a subroutine at the point where processing of the desired activity begins. This may be accomplished by using the ENTRY POINT phrase. The section of code to be executed in the subroutine must be identified with an explicit beginning (entry-name) and ending (EXIT). The call statement in the main routine will call this entry-name (instead of PROGRAM-ID). (8; 15-15)

In Figure 6-1 which follows, there are three examples of the CALL/ENTRY POINT phrase. The second CALL is to JOB4Ø3. This is the simplist format. The ENTRY phrase may also be written (optional) "ENTRY POINT IS JOB4Ø3". In the third example, the using clause is added and all the rules regarding its use apply, as described above in 6.2.3 and 6.2.4. In the first example, the call is to JOB2Ø5. Notice the range of JOB2Ø5 includes JOB4Ø3 and JOB6Ø1. This illustrates the uniqueness of the EXIT statement. Control will not be returned to the calling program until the processing encounters an exit statement specifying the entry-name where processing began. Therefore, when the processing for JOB2Ø5 is begun, it will continue through JOB4Ø3 and JOB6Ø1 until EXIT-2Ø5. and EXIT JOB2Ø5 is found. More details may be found in the Honeywell COBOL Users Guide, Section XV, and the Honeywell COBOL Reference Manual, Section VII (CALL, ENTER, and EXIT).

6.2.7 Linkage Section. COBOL subroutines run on the Honeywell machine do not require the use of a "LINKAGE SECTION" as on the IBM machine. However, if one is present in the program, the Honeywell will accept it. See paragraph 6.3.2.1.

6.3 IBM COBOL Program Linkage. Following is a discussion of the IBM COBOL Program Linkage.

6.3.1 CALL Statement. COBOL can invoke the execution of a subroutine by use of the CALL statement. The program containing the call statement is the "calling" program; the program being called is the "called" program. The IBM COBOL CALL statement has the format:

```
CALL 'ident' USING var-1, var-2,...
```

where ident is the PROGRAM-ID of the subprogram or an ENTRY name in the subprogram (ENTRY is more fully discussed in paragraph 6.3.3). Var-1, var-2...are the names of the data items being passed to the subprogram. Each of these data items must be defined in the calling routine before the call. If no data items are necessary the CALL statement is reduced to

```
CALL 'ident'.
```

Upon completion of the called routine, control is returned to the line following the CALL statement in the calling program. The ident portion of the CALL statement may also be a variable name. Before the call statement is executed, the name of a valid subroutine must be stored into this variable.

```
(MAIN)
        .
        .
        .
     CALL Job2Ø5.
        .
        .
        .
     CALL Job4Ø3.
        .
        .
        .
     CALL Job6Ø1 using one, two, three.
        .
        .
        .
(SUBROUTINE)
        .
        .
        .
     ENTRY Job2Ø5.
        .
        .
        .
     ENTRY Job4Ø3.
        .
        .
        .
   EXIT-4Ø3.
     EXIT Job4Ø3.
        .
        .
        .
     ENTRY Job6Ø1 using Alpha, Delta, Fox.
        .
        .
        .
   EXIT-6Ø1.
     EXIT Job 6Ø1.
        .
        .
        .
   EXIT-2Ø5.
     EXIT Job2Ø5.
        .
        .
        .
```

ENTRY/EXIT Statements

FIGURE 6-1

For example, if the following statement appeared in the calling routine

```
MOVE 'SUBROUTN' TO VARIABLE-NAME.
```

the subroutine SUBROUTN could be called in the following manner

```
CALL VARIABLE-NAME USING...
```

(Note the variable name is not enclosed in quotes).

This would have the same effect as

```
CALL 'SUBROUTN' USING...
```

See Attachment 1 for an example of the CALL statement.

6.3.2 Subroutine (Called Program). The subroutine or called program appears as a normal IBM COBOL program, with some minor variations. These are discussed below. See Attachment 1 for a COBOL subroutine example.

6.3.2.1 Linkage Section. If data is to be passed to the subroutine through the CALL statement on the IBM computer, the called program must contain a LINKAGE SECTION. (See Attachment 1 for an example). If no data items are passed to the called program the LINKAGE SECTION may be omitted (see Attachment 4). The LINKAGE SECTION describes the data items which will be received from the calling program. The data items are described in the subroutine in the same manner as in the WORKING-STORAGE of the DRIVER, except no VALUE clause may be used. The data descriptions in the called program should match the data descriptions in the calling program. The LINKAGE SECTION must follow the WORKING-STORAGE SECTION if the WORKING-STORAGE SECTION is used. If the WORKING-STORAGE SECTION is not used the LINKAGE SECTION must follow the FILE SECTION. If no FILE SECTION is present the LINKAGE SECTION follows the DATA DIVISION statement.

6.3.2.2 PROCEDURE DIVISION Clause. The PROCEDURE DIVISION clause of the called program is modified to appear as:

```
PROCEDURE DIVISION USING var-1, var-2...
```

where var-1, var-2...are the names of the data items being passed to the called program. (See Attachment 1 for an example). If no data items are being passed to the subroutine, the USING var-1, var-2... clause is omitted. The order of the data items is important. The data items in the CALL statement are associated on a one-to-one basis with the data items in the PROCEDURE DIVISION clause, not by data item name. Since the association is by data item order, and not by name, different names may be used for the same data item in each routine. For example if the statement CALL 'SUB' USING A, B, C appeared in the calling program, and program SUB contained PROCEDURE DIVISION USING X, Y, Z, in the called program X would have the

value of A in the calling routine, Y would have the value of B, and Z the value of C. Execution of the subroutine begins with the statement following the PROCEDURE DIVISION line, and continues through the first EXIT, STOP or GOBACK. Control is then returned to the calling routine on the line following the CALL statement.

6.3.3 ENTRY Statement. An alternate entry point to a subroutine may be defined by use of an ENTRY statement. The ENTRY statement has the form

```
ENTRY 'subname' USING var-1, var-2,...
```

If no data items are being passed the form becomes

```
ENTRY 'subname'.
```

This statement appears within the actual PROCEDURE DIVISION. The 'subname' clause may be replaced by a variable name. If the following appears within the PROCEDURE DIVISION:

```
MOVE 'ENTRYNAM' TO VARIABLE-NAME.
```

the ENTRY statement

```
ENTRY VARIABLE-NAME...
```

would have the same effect as

```
ENTRY 'ENTRYNAM'...
```

(Note the variable name is not enclosed in quotes). This allows the flexibility of assigning different ENTRY names to the same section of code during program execution. The CALL statement in the calling routine is the same as that used for the PROCEDURE DIVISION call (paragraph 6.3.1). Execution of the subroutine begins on the line following the ENTRY statement. For example:

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  SUBROUTN.
      .
      .
      .
DATA DIVISION.
      .
      .
      .
```

```
WORKING-STORAGE SECTION.
      .
      .
      .
LINKAGE SECTION.
Ø1  VAR-1  PIC S9V99.
Ø1  VAR-2  PIC S9V99.
      .
      .
      .
PROCEDURE DIVISION USING VAR-1, VAR-2.
      .
      .
      .
   GOBACK.
   ENTRY 'ENTRY2' USING VAR-1, VAR-2.
      .
      .
      .
   GOBACK.
      .
      .
      .
```

In this subroutine example both the PROCEDURE DIVISION entry point and the ENTRY entry point are used. A call in the calling program of

```
CALL 'SUBROUTN' USING A, B.
```

would pass the value of A and B to SUBROUTN. Execution would begin on the line following PROCEDURE DIVISION and continue through the first GOBACK. A

```
CALL 'ENTRY2' USING A, B.
```

would also pass the values of A and B to the subroutine; however, execution would begin on the line following the ENTRY2 statement and continue through the second GOBACK. (12, 14, 16)

6.4 FORTRAN Linkage. The execution of a FORTRAN subroutine or function requires certain changes be made to the program. These changes are the same on both the IBM and Honeywell computer and are discussed below.

6.4.1 Subroutines. A subroutine is a subprogram used to calculate commonly used computations. A discussion of subroutines follows.

6.4.1.1 SUBROUTINE Statement. Each subroutine begins with a SUBROUTINE statement of the form

```
SUBROUTINE sname (var1, var2,...,varN)
```

where sname is the subroutine name, and var1, var2,...,varN represent variable or array names, or the dummy name of another SUBROUTINE or FUNCTION. Var1, var2,...varN serve as both inputs and outputs to the subroutine. For example, in the program

```
SUBROUTINE PROG (A, B, C, D)
C=A+B+C
D=A+C
  .
  .
  .
```

The variables A and B serve as inputs. C is both an input and an output since the value of C is used in a calculation and is also to the left of an assignment operator (=). The original value of C would be passed to the subroutine, added to A and B, and the result stored in C. Upon completion of the subroutine the changed value of C will be returned to the calling program. D serves only as an output since it appears only on the left of an assignment operator. A and B are only inputs since they appear only to the right of an assignment operator.

6.4.1.1.1 Data Element Declarations. Data elements being passed to a subroutine should match the data element types in the subroutine. If the default attributes of the variables in a subroutine do not match the attributes of the variable being passed the subroutine variables must be declared in the subroutine by use of DIMENSION, DOUBLE PRECISION, etc. These statements must be the first ones in the subroutine. For example, if the calling routine is passing the double precision value A to the subroutine, and the subroutine statement is

```
SUBROUTINE SUB (Z)
```

then following the SUBROUTINE statement should be

```
DOUBLE PRECISION Z
```

to insure the data element will be represented within the subroutine in the same manner as the representation in the calling program. If A had not been double precision, the default attribute for Z would match those of A and no additional statements would be necessary. If no inputs or outputs are necessary, var1, var2,...varN may be omitted.

6.4.1.1.2 Passing Functions or Subroutines. In addition to passing variables to a subroutine, functions (see paragraph 6.4.2) or other subroutines may be passed. If subroutine A is defined as

```
    SUBROUTINE A (X, Y, Z)
```

subroutine B as,

```
    SUBROUTINE B (Q, E, C, D)
```

and is called (see paragraph 6.3.1.2) by CALL B (A, E, E, D) then the subroutine A is passed into subroutine B. If B contains the line CALL Q (E, C, D), the subroutine A (which is represented by Q in B) would actually be called.

6.4.1.1.3 RETURN and END Statements. Each subroutine should contain one or more RETURN statements. When a RETURN statement is encountered in the logical execution of the subroutine control is returned to the calling program at the line following the CALL statement. For example if the subroutine contained the statements:

```
        .
        .
        .
    IF (A .EQ. B) RETURN
        .
        .
        .
    RETURN
        .
        .
        .
```

and A did in fact equal B then the subroutine would terminate at the first RETURN. If A did not equal B execution would continue until the second RETURN statement. The last physical statement of the subroutine must be an END statement.

6.4.1.2 CALL Statement. A CALL statement is used to execute a subroutine. It has the form

```
    CALL sname (var1, var2,...,varN)
```

where sname is the name of the subroutine being called, and var1, var2,...varN are the input/output variables, or SUBROUTINE or FUNCTION names. If the subroutine requires no inputs or outputs then the form of the call becomes

```
    CALL sname.
```

Upon termination of the subroutine execution continues at the line following the CALL.

6.4.2 Functions. A FORTRAN function is an independent program which is executed each time its name is encountered. Following is a description.

6.4.2.1 FUNCTION Statement. A function returns one value to the calling routine. A function begins with the statement

```
FUNCTION fname*S (var1, var2,...,varN)
```

where fname is the FUNCTION name, *S (which is optional) is a length specification, and var1, var2,...,varN are variable or array names, SUBROUTINE or other FUNCTION names. The value which is returned is fname. The function must contain fname to the left of an assignment operator. For example

```
FUNCTION A (B, C, D)
A=B+C+D
RETURN
END
```

would take the input values B, C, and D, add them together and return the result as A. If the default attributes for the function name (which are the same default attributes as a variable of the same name) are to be changed the calling program would have a declare statement (INTEGER, REAL, etc) for the function name. This declare would be identical to the declare for variables. For example, if function CALC was to return a double precision result, the calling program would contain the following declare

```
DOUBLE PRECISION CALC
```

The FUNCTION statement would be altered to agree with the declaration as follows:

```
DOUBLE PRECISION FUNCTION CALC (X, Y, Z)
```

If the value to be returned from the function INT was to be INTEGER*2, then INT would be declared as

```
INTEGER*2 INT
```

in the calling routine. The function line would then become

```
INTEGER FUNCTION INT*2 (A, B)
```

6.4.2.2 RETURN and END Statements. The rules for RETURN and END statements for a function are identical to those of a subroutine (see paragraph 6.4.1.1.3).

6.4.2.3 Initiating a Function. A function is initiated in the calling program whenever the function name is encountered. For example, if the calling program contained

```
Z = A (X, Y, P)
```

and function A was

```
FUNCTION A (B, C, D)
A = B+C-D
RETURN
END
```

then X+Y-P would be calculated and stored in Z. If the calling program contained

```
Z = 12.Ø/A(X,Y,P)
```

and the function A is the same as the one above, then Z would contain the result of 12.Ø/(X+Y-P). (10, 15, 17)

6.5 Honeywell COBOL/FORTRAN Linkage. The linkage of COBOL and FORTRAN in the same program is not difficult if done with great care and attention to detail.

6.5.1 Job Control Language. The programmer must keep in mind when compiling two languages for the same execution, concessions must be made to allow peaceful co-existance and smooth execution. The Job Control Language (JCL) performs most of these functions and will be addressed in detail in Section 7.

6.5.2 Basic Structure. When linking COBOL and FORTRAN the rules are the same as when a COBOL driver calls a COBOL subroutine or a FORTRAN driver calls a FORTRAN subroutine. The CALL statements (with arguments) are as follows:

```
(COBOL)
          .
          .
          .
      CALL name USING X, Y, Z.
```

or

```
(FORTRAN) CALL name (X, Y, Z)
          .
          .
          .
```

The CALL statements are very much alike, but in the subroutine the arguments are received in different places. The COBOL subroutine receives the data in its PROCEDURE DIVISION header:

```
        .
        .
        .
    PROCEDURE DIVISION USING X, Y, Z.
        .
        .
        .
```

where as, in the FORTRAN subroutine the data is received in the first line, with the routine's name:

```
    SUBROUTINE name (X, Y, Z)
        .
        .
        .
```

A COBOL main program calling a FORTRAN subroutine would appear:

```
    (COBOL PROGRAM)
        .
        .
        .
      CALL name USING X, Y, Z.
        .
        .
        .
    (FORTRAN PROGRAM)
    SUBROUTINE name (X, Y, Z)
        .
        .
        .
```

A FORTRAN main calling a COBOL subroutine:

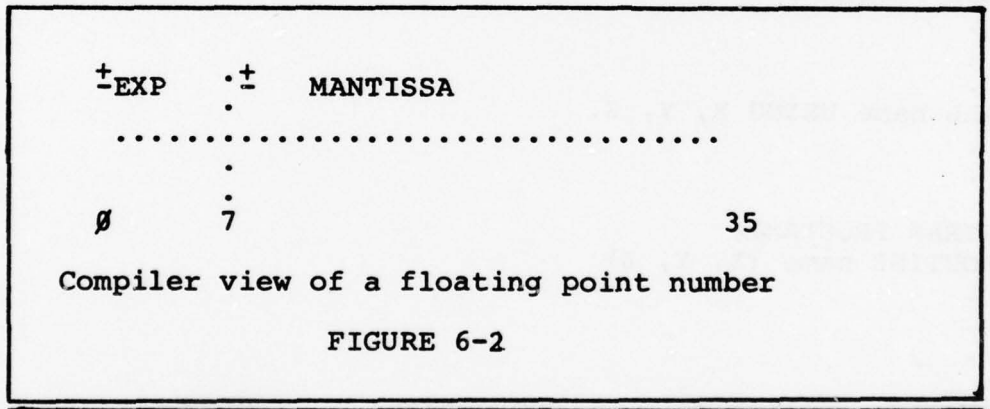
```
        .
        .
        .
    (FORTRAN PROGRAM)
    CALL name (A, B, C)
        .
        .
        .
    (COBOL PROGRAM)
        .
        .
        .
      PROCEDURE DIVISION USING A, B, C.
        .
        .
        .
```

6.5.3 Data Representation. It is imperative that data used in both the main program and subroutine be represented the same way in each. If ZAP is defined in Honeywell COBOL as COMP-1, then it should be defined as INTEGER in FORTRAN, because they mean the same thing (see Section 5).

If the definition of a data item (an argument) in a COBOL program does not match the corresponding definition of the data item in the FORTRAN program, the machine will not flag it as an error.

At compile time, the COBOL and FORTRAN programs will be compiled by their respective compilers. Because the two programs are compiled separately, no comparison is made and the machine is not aware of any difference of definitions.

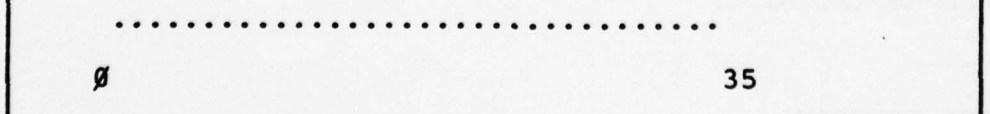
It is at execution time that dissimilar data definitions can become a problem. For example, if variable CASH is defined in a COBOL program as COMP, the compiler allocates one 36-bit, binary word in a floating-point format. The first eight bits represent the exponent, and the other 28 are available for the mantissa. Only the compiler considers the word to be in two, distinct parts, such as Figure 6-2:



Compiler view of a floating point number

FIGURE 6-2

In memory, the actual appearance of this word is



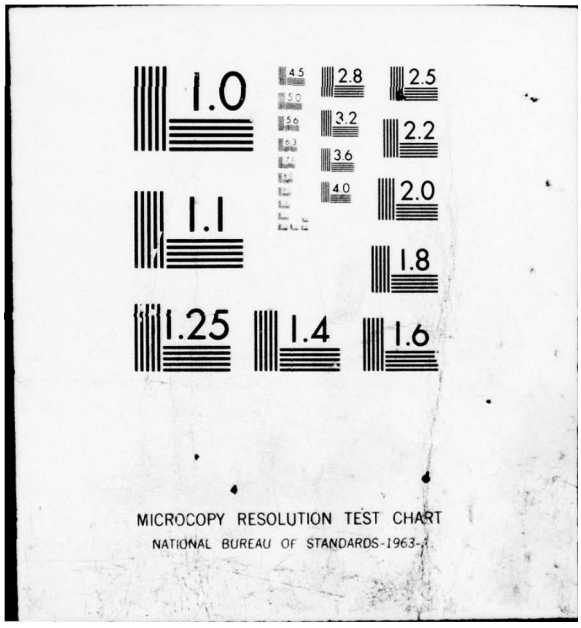
Actual view of any word

FIGURE 6-3

AD-A070 959 STRATEGIC AIR COMMAND OFFUTT AFB NE F/G 9/2
THE INTERFACE OF COBOL AND FORTRAN ON THE HONEYWELL 6080 AND IB--ETC(U)
APR 79 D W LIND, R D GEER, J W WHITE

UNCLASSIFIED NL

2 OF 3
AD
A070959


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Meanwhile, in the FORTRAN subroutine, CASH has inadvertantly been coded as an INTEGER. During compilation of the subroutine, the FORTRAN compiler recognizes CASH will be passed to it for use as an integer: a fixed-point number, appearing in the same manner as Figure 6-3 above.

During execution, each time the COBOL program must use CASH, it will apply the floating-point format.

However, when the FORTRAN routine is called it will access CASH, as required, as an integer and apply the fixed-point format. This is where the problem begins. The FORTRAN routine reads all 36-bits as a single, binary whole number, and not a floating-point number. One can see how distorted the value of CASH can become, by applying a fixed-point format to a floating-point number. Further details may be found in Section 4 of this report.

6.5.4 The Interface. When interfacing COBOL and FORTRAN, on the Honeywell, attention to record length is mandatory. When a record created in COBOL is rewritten from FORTRAN, one computer word (6 bytes) is added to it. This added word contains a SLEWING character for carriage control (output). Notice the following example:

```
      .
      .
      .  (COBOL Program)
FD TSTFYL label records standard.
    Ø1 TEST-FILE    PIC X(36).
      .
      .
      .
WORKING-STORAGE SECTION.

Ø1 REKORD.

   Ø3 FILLER    PIC X    VALUE SPACE.
   Ø3 PART-1    PIC 9(3) VALUE Ø1Ø.
   Ø3 FILLER    PIC X(5) VALUE SPACE.
   Ø3 PART-2    PIC 9(3) VALUE Ø2Ø.
   Ø3 FILLER    PIC X(5) VALUE SPACE.
   Ø3 PART-3    PIC 9(3) VALUE Ø3Ø.
   Ø3 FILLER    PIC X(5) VALUE SPACE.
   Ø3 PART-4    PIC 9(3) VALUE Ø4Ø.
```

(Also see Attachment 5).

The number of bytes in the record "REKORD" is 28. Because the machine works in terms of six-byte words, this record will take a minimum of five words (30 bytes).

Then, if the file is rewritten in FORTRAN, the slew word is added. The word length is now six words or 36 bytes. When control is returned to the COBOL driver any attempt to read this record will cause an abort if the file description does not provide for the increased record length. In the example above, if TEST-FILE had a picture of only 28 characters, the record length would exceed the file description and the processing would stop with an execution activity error.

6.5.4.1 The Taming of the Slew. The programmer can circumvent the potential problem of the extra carriage control word in a number of ways. The first and simplest way is not to write in FORTRAN when interfacing COBOL with FORTRAN. However, because this may not be the best way to accomplish the task at hand, other methods are suggested. Job Control Language provides the first alternative. The "NO SLEW" option on the $ FFILE JCL card prevents automatic slewing. What alternatives are provided by the code itself? In example 6.5.4 above, the file description is modified. TSTFYL is increased by the length of one computer word. The programmer may wish to increase the size of the actual description of the record in the WORKING-STORAGE SECTION as well, leaving six bytes of empty space at the end of the record. Here, caution is advised. Unless specifically told otherwise, FORTRAN will work in terms of WORDS, not bytes and can mutilate ones' data beyond recognition.

6.5.4.2 Padding. If the record ends at a word boundary, adding a FILLER (for the extra word) still ends the record at a word boundary. When the record is used in the FORTRAN program, it will remain intact through the FORTRAN output. Knowledge of how this data is stored is of great help in understanding what happens to it through the course of an execution. See the File Section of CARAYF in Attachment 6. The record to be read is set up as six characters, instead of five, thus, making each record one computer word in length.

6.5.5 A FORTRAN Alternative. When a problem occurs while interfacing two languages, such as COBOL and FORTRAN, it can usually be solved by adjusting one of the two routines. In the previous example, the COBOL program was modified to solve the problem (Attachment 6, File Section). But what if parts of the COBOL program cannot be adjusted? The next logical step is to change the FORTRAN program.

To state a new example, suppose a COBOL program was to pass a table to a FORTRAN subroutine. It is a two-dimensional table of three rows and three columns, consisting of nine elements, each five characters long (alphabetic data) (Attachment 8, File Section).

The character array being passed is seen by the FORTRAN subroutine as a string of 45 contiguous characters. The problem is breaking down the string of characters into their proper elemental lengths. The FORTRAN language on the Honeywell machine provides this capability with the DECODE statement. Its general form is:

DECODE (a,t) list (See Attachment 8)

where t is the FORMAT information describing the manner in which the decoding will occur; a is the item to be decoded; and list is the variable(s) into which the decoded a is to be stored. (10: 4-20)

NOTE: Numeric data is passed on a word for word basis and presents much less of a problem, as long as the data items are represented in both programs in the same way.

The DECODE statement in Attachment 8 is:

```
    DECODE (JARRAY,11Ø) ((IARRAY(I,J),J=1,3),I=1,3)
11Ø FORMAT (9A5)
```

where a is JARRAY (what is to be decoded) and t is format 11Ø, how is it to be separated. The list is another array where the decoded (separated) elements will be stored, followed by the structure of IARRAY: three rows across and three columns down.

Figure 6-4 shows how the FORTRAN output is affected by the automatic slewing without the DECODE statement. (See pg A72)

Additional comments on handling tables/arrays between COBOL and FORTRAN programs can be found in Section 8 of this report.

6.6 IBM COBOL/FORTRAN Linkage. The interface between COBOL/FORTRAN progrmas, and FORTRAN/COBOL programs is very similar to the interfaces between COBOL/COBOL and FORTRAN/FORTRAN.

6.6.1 COBOL Calling FORTRAN. The CALL statement in a COBOL program calling a FORTRAN subprogram remains the same as described in paragraph 6.3. The SUBROUTINE statement in FORTRAN remains the same as described in paragraph 6.4.1.1. Care should be exercised to insure word boundaries (paragraph 4.2) and data types (paragraph 5.5) are consistent between the COBOL and the FORTRAN. Also, the JCL must be modified to accommodate two languages (see paragraph 7.3).

6.6.2 FORTRAN Calling COBOL. The CALL statement in a FORTRAN program calling a COBOL subprogram remains the same as described in paragraph 6.4. The COBOL subroutine would remain the same as described in paragraph 6.3.2. Care should be exercised to insure word boundaries (paragraph 4.2) and data types (paragraph 5.5) are consistent between the FORTRAN and COBOL. Also, the JCL must be modified to accommodate two languages (paragraph 7.3).

```
03-10-79
ALPHA TAECH OIELI
RAVOC FOX DIADD
RLYDE GOLF COBOL
```

```
03-10-79
ALPHA RAVOC RLYDE
TAECH FOX GOLF
OIELI DIADD COBOL
```

Output Without DECODE Statement

FIGURE 6-4

SECTION 7. JOB CONTROL LANGUAGE

7.1 Introduction. Each program submitted to the computer must contain certain Job Control Language (JCL) statements. These JCL statements give the computer information necessary to exeucte the program. This information includes such items as the language in which the subroutines or programs are written, identification of files necessary for execution, location of card data input, etc. The JCL for the IBM machine and the Honeywell machine is quite different and not interchangeable. The complexities and variations of each JCL are great and beyond the scope of this manual. Detailed explanations of each machine's JCL is contained in the appropriate machine's Job Control Language Reference Manual. Following is a brief description of the JCL necessary to link different languages together for execution as a single program.

7.2 IBM JCL. Before two different languages can be executed in one program each language must be compiled separately, then linked and executed.

7.2.1 Program Compilation. One language of the mixed language program must be compiled first. The language which is compiled first is unimportant; however, if the main or driver program is not compiled first some additional control cards are necessary in the execute portion of the job (see paragraph 7.2.3). To compile the program use FTG1C for FORTRAN or VSCOBC for COBOL. For example, to compile the FORTRAN portion use:

```
//       EXEC  FTG1C
//FORT.SYSIN  DD *
```

Following the second card would be the FORTRAN deck.

7.2.2 Program Execution. After one language is compiled, the second language may be compiled and the program executed. To compile and execute use FTG1CLG for FORTRAN or VSCOBCLG for COBOL. For example, to compile the COBOL portion of a FORTRAN/COBOL program, and to execute the entire program use:

```
//       EXEC  VSCOBCLG
//COB.SYSIN   DD *
```

Following the second card would be the COBOL deck. This example assumes the FORTRAN portion of the deck was compiled first using FTG1C and is the driver. (If the first routine compiled using FTG1C or VSCOBC is not the driver, see paragraph 7.2.3). A typical deck setup with COBOL as the driver and a FORTRAN subroutine would be as follows (see also Attachment 3):

```
//       EXEC  VSCOBC
//COB.SYSIN   DD *
         COBOL program
//       EXEC  FTG1CLG
//FORT.SYSIN  DD *
         FORTRAN program
            .
            .
            .
```

7.2.3 ENTRY Statement. If the first program compiled (using FTG1C or VSCOBC) is not the driver, an ENTRY card must be included to specify which subroutine is to be the driver (see Attachment 4). To accomplish this, two additional cards are necessary. Following the last program deck (compiled using FTG1CLG or VSCOBCLG), a

```
//LKED.SYSIN  DD *
```

card must be inserted. Following this would be an

```
ENTRY driver
```

card where driver is the name of the main program (the subroutine name for FORTRAN or PROGRAM-ID for COBOL). The ENTRY card does not start with // and must not begin in column 1. For example, assume a COBOL/FORTRAN program is to be executed. If the FORTRAN program (called FORTSB) is the driver and was compiled second using FTG1CLG, then the deck would appear as:

```
//       EXEC  VSCOBC
//COB.SYSIN   DD *
         (COBOL routine)
//       EXEC  FTG1CLG
//FORT.SYSIN  DD *
         (FORTRAN routine)
//LKED.SYSIN  DD *
         ENTRY FORTSB
               .
               .
               .
```

If a COBOL/FORTRAN program with the COBOL routine (PROGRAM-ID COBSUB) is the driver, and is compiled second using VSCOBCLG, then the deck would be:

```
//      EXEC  FTG1C
//FORT.SYSIN  DD *
        (FORTRAN routine)
//      EXEC  VSCOBCLG
//COB.SYSIN   DD *
        (COBOL routine)
//LKED.SYSIN  DD *
        ENTRY COBSUB
              .
              .
              .
```

7.2.4 Additional IBM JCL. Since the first subroutine or program is compiled using FTG1C or VSCOBC, normal default JCL files are unavailable for use during execution for this language. If the FORTRAN routine is compiled first the following cards must be inserted after the COBOL deck (see Attachment 4) (if no ENTRY card is used) or after the ENTRY card (if the ENTRY card is used):

```
//GO.FTØ5FØØ1  DD *
       (input card data)
//GO.FTØ6FØØ1  DD SYSOUT=A (printer file)
//GO.FTØ7FØØ1  DD SYSOUT=B (punch file)
```

If the COBOL routine is compile first then the following cards must be inserted after the FORTRAN deck (if no ENTRY card is used) or after the ENTRY card (if the ENTRY card is used)(13):

```
//GO.SYSDBOUT  DD  SYSOUT=A
//GO.SYSOUT  DD  SYSOUT=A
```

7.3 Honeywell (JCL).

The JCL used by the Honeywell is unique, as all of its statements have this structure:

```
$       INSTRUCTION  OPTIONS/PARAMETERS
1       8            16
```

The only exception to this uniqueness is ***EOF. This card uses card columns 1-6 and is the last card in the (batch) deck, signifying the physical end of the job; hopper empty status.

The JCL discussed here will be those statements used to successfully execute the test programs in Attachments 5 thru 8.

Because many of the JCL statements are duplicated, there is no point in discussing them more than once. However, if the use or meaning of a statement should change from one program to the next, it will be discussed again with reference to the change or new meaning.

When the Control Card format specifies options, it means the programmer may select system options to get additional information output about the program. It also means if no options are selected, the programmer wishes only the standard default options. They are underlined in the control cards manual.

It is assumed the programmer will supply the first three JCL cards for ALL batch jobs and understands their meanings. They are 1) SNUMB, 2) IDENT, and 3) USERID. Much of the discussions are written with reference to the Honeywell Control Cards Reference Manual. (8)

7.3.1 $ LOWLOAD. Refer to Attachment 5. The $ LOWLOAD card is used to load the program it precedes into memory above the fault cells and control words. It establishes where in core the program will be loaded and releases unused portions of core back to the machine. This is a useful tool for debugging purposes.

7.3.2 $ OPTION. The next card is the $ OPTION FORTRAN card. It alters the standard General Loader options during loading, based on options specified. Here, the FORTRAN option is used. It tells the General Loader there is a FORTRAN program following and to set all options required for loading it.

7.3.3 $ ENTRY. When two or more programs are to be executed together, the first program in the deck will be considered the main program or driver. If the driver is not the first program in the deck, the $ ENTRY option card is added to the JCL preceding the execution activity. This card tells the machine which program is the main program and to enter it first. Thus, the option is the name of the main program or the symdef. The symdef is the first four characters of the program name, preceded by (C.). Attachments 5 and 6 illustrate these two uses of the option.

7.3.4 $ FORTY (FORTRAN). Either the $ FORTY or the $ FORTRAN statement may be used to call the FORTRAN compiler. There are many options available, most of which contribute to debugging.

7.3.5 $ COBOL. The $ COBOL card calls the COBOL compiler, and like the $ FORTRAN card, allows the programmer to specify desired options. However, because one system default option is DECK, NDECK is specified in all test programs used in this report. The option DECK requests a binary object program (card) deck to be punched as output. The EISF option forces compilation of object program for the Extended Instruction Set Computer. It overrides the OBJECT-COMPUTER paragraph. Attachments 6 and 8 also use LSTOU. This option gives a listing of the assembled object program output.

7.3.6 $ EXECUTE. To execute programs, the $ EXECUTE card is required. It must follow all programs, but precede their data. The option field is used to set sense switches on or off, and specify options. Attachment 8 specifies DUMP. This option will, if the activity terminates abnormally, give the slave core dump, program registers, upper SSA, and slave program prefix.

7.3.7 $ LIMITS. Occassionally, a job will attempt to go beyond normal system limits placed on storage, CPU time, lines printed, or I/O time. Any or all of these may be increased by using the $ LIMITS card. Used in Attachment 5, the first comma means an increase in CPU time is not required. The 24K is the maximum memory requested for running the program. The -4K increases the amount of salve core.

7.3.8 $ FILE. The $ FILE cards identify the files used by the program(s). AB is a file code given in the COBOL program, and Ø8 is the file code given the same file in the FORTRAN program. The A1 is a Logical Unit Designator (LUD), used to name a file for later use. The S indicates the file is to be saved.

7.3.9 $ SYSOUT. Output from all test programs herein is to the system printer. The $ SYSOUT cards select OT for printer output from the COBOL program, and Ø6 for printer output from the FORTRAN program.

7.3.10 $ DATA. When submitting programs which include input in the form of cards, the $ DATA is used. In Attachment 6, the COBOL main program selects the input file CARDIN and assigns the letters CI for cards to identify the card reader. The variable field in the $ DATA card includes this file code, thus:

```
$ DATA CI
```

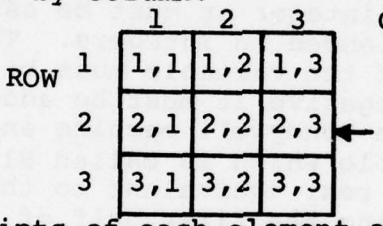
The machine now knows the COBOL program will read cards from the device known as CI and the data to be read follows the $ DATA card.

7.3.11 $ ENDJOB. The $ ENDJOB is the last card of every job. It indicates that the job being processed is a candidate for execution.

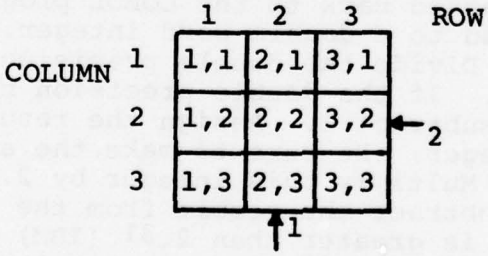
SECTION 8. ADDITIONAL TECHNIQUES

8.1 Introduction. In this Section, a few follow-up techniques are discussed that have been discovered in the research, but not included previously since they did not directly apply. They represent additional information and techniques for consideration by programmers planning to interface COBOL and FORTRAN in the same program.

8.2 Passing a Table from COBOL to FORTRAN: Subscripts: In addition to the problems mentioned earlier regarding the passing of a table from COBOL to FORTRAN, addressing a particular element of the table can also be a nightmare. Because the tables/arrays are built differently in each language, their subscripts are also different. Put more simply, the subscripts that identify a specific element in the COBOL program will not correspond with the subscripts (of the same element) in the FORTRAN program after the table has been passed. For example, COBOL creates the following nine element, two dimensional table, row by column:



where the subscripts of each element are shown and the arrow points to the specific element (2, 3) to be selected for use in the FORTRAN program. When the table is passed, FORTRAN puts it into array form. However, because FORTRAN builds arrays column by row, the table above is rebuilt and now appears thus:



where the subscripts of each element are shown. Arrow 2 points to the specific element desired, and arrow 1 points to the element actually accessed. Notice that the subscripts of the first element (1, 1), the middle element (2, 2), and last element (3, 3), match those of the COBOL table. If one's table is subscripted as simply as these two illustrations, there would be no problem accessing the first, middle, and last elements. But this often is not the case. It is accidental that Row 1, Column 1 in the COBOL table matches Column 1, Row 1 in the FORTRAN array, and so on.

As can be seen by the arrows, addressing element (2, 3) in the FORTRAN subroutine will yield incorrect data. The machine assumes this element is the desired element and gives no error nor comment. Providing there are no other errors in the program, element (2, 3) will be processed, eventually output, and the programmer may not even know the output is faulty. This is why attention to subscripts is so important.

The solution to the problem is simple: reverse the order of the subscripts. If the data in element (2, 3) is the data desired from the COBOL program, reversing the order of the subscripts results in (3, 2) in the FORTRAN program. Notice the position of elements (2, 3) and (3, 2). In memory, the address of element (2, 3) is the same address of element (3, 2).

8.3 Double Word Integers. FORTRAN ordinarily has no capability to use double word integers. However, these integers may be converted to double precision floating point numbers in the FORTRAN program. Attachment 10 is a program that demonstrates how this conversion can be made. The double word integer may be passed to the FORTRAN routine as an integer or must be assigned to working variables that are equivalenced to integers. The integer representing the second half of the variable must be examined for its sign. If the number is negative it must be added to $2.^{32}$ in the IBM machine or $2.^{36}$ in the Honeywell machine and the result stored in a floating point variable which is called B1 in the example. Then the double precision real equivalent to the double word integer is formed by multiplying the first half of the integer by $2.^{32}$ (or $2.^{36}$) and adding B1. Note that in order to insure precision, the precise values of $2.^{32}$ or $2.^{36}$ must be used. $2.^{32}$=4294967296. $2.^{36}$=68719476736.

8.4 Reversing the Double Word Integer Process. If the double word integer must be passed back to the COBOL program, the real number must be converted to a double word integer. This process is relatively simple. Divide the double precision floating point number by $2.^{32}$ or $2.^{36}$. If the double precision number and the quotient are negative subtract 1. Assign the result to the first word of the double integer. Be sure to make the assignment to the integer variable. Multiply this integer by $2.^{32}$ (IBM) or $2.^{36}$ (Honeywell) and subtract the result from the double precision number. If the result is greater than $2.^{31}$ (IBM) or $2.^{35}$ (Honeywell) subtract $2.^{32}$ or $2.^{36}$. If the result is less than $-2.^{31}$ (IBM) or $-2.^{35}$ (Honeywell) add $2.^{32}$ or $2.^{36}$. Store the result in the second integer. To pass the result back to the calling routine, assign the real equivalent addresses of the integer values to the array or dummy arguments.

8.5 Packed Decimals. FORTRAN does not read or perform computations with packed decimals. One of two methods may be used to circumvent this problem. The packed decimals may be converted to computational items in the COBOL program and the computational items passed to the FORTRAN program. Alternately, the packed decimals may be written to a file in the COBOL program. The FORTRAN program may then read this file. The former method is recommended.

8.6 Double Precision Floating Point Numbers. If it is necessary to transfer double precision floating point numbers to a FORTRAN program through single precision real variables, the following procedure can be followed. Assign the passed variables to a two word array in the FORTRAN program. Equivalence a double precision variable to this array. This double precision variable can then be used as a double precision variable equivalent to the COBOL variable. To pass the variable back to COBOL, assign the FORTRAN array elements to the passing variables. See Attachment 11.

8.7 Passing Integers Through Real Variables. If an integer must be passed through a real variable and subsequently used in computation, the following method can be used to insure typing problems are minimized. EQUIVALENCE a real variable to an integer. Assign the passing variable to the real variable. The integer may be used in place of the real variable. Attachment 9 has an example of this method. To pass the integer back to the calling program through the real number, assign the equivalenced real number to the passing variable. Any other simple way of accomplishing this task is preferable to this equivalence method since it is somewhat difficult for the novice programmer to understand.

8.8 Comparisons. If data items are to be compared, they must be properly justified otherwise a comparison will be meaningless. For example, if CATØ is compared to ØCAT the computer will determine that the strings are not equal. The data must be properly justified in the variable's field. Likewise only variables of the same type should be compared when display or character items are involved. The programmer should also keep in mind that if data in variables of different types are compared, one of the datum will be converted to the other type. For example, in FORTRAN the statement

```
IF (A.LT.I) GO TO 7
```

will cause the computer to convert I to the real floating point equivalent before making a comparison with A. Hence, if A and I contain character items, the comparison will not be valid.

8.9 An Illustration.

8.9.1 ARRAY1. The program ARRAY1 was written to demonstrate the way the computer handles data items and the means of handling strange combinations of items. This program uses a number of techniques that are not recommended if more straight forward approaches are available. However, there are circumstances in which these techniques may be the only way to accomplish a task.

8.9.2 PURPOSE. This short program demonstrates some non-trivial data transfer problems. The COBOL subprogram, ARRAY1, defines four items in WORKING-STORAGE at the 77 level. The first item has a picture of X(6) and a value of "77ITEM". The second item has a picture of X(8) and a value of "2ND ITEM". The third item has a picture of X99X and a value of "#77Ø". The last item has a picture of 9(4) and a value of 7777. Note that the first three items are display, but the last item is a computation item of the binary integer type. A table called TABLE-ONE is also defined. Since this table contains items of different types and lengths, it should not be considered a standard COBOL table. The first item in the table has a picture of X(6) and a value of "ARRAY". ITEM-TWO has a PICTURE S9(3) and a value of 123. The third item is a computational signed integer of value 12345. ITEM-FOUR has a PICTURE of S9(6) V 9(4) and is a binary decimal of value 2468.5. The last item is also a binary decimal with a value of -123.01. The objective of this program is to transfer these values to the FORTRAN subroutine FSUB1 which will write the values to SYSOUT. As will become apparent in the subsequent discussion, this transfer requires some non-trivial considerations.

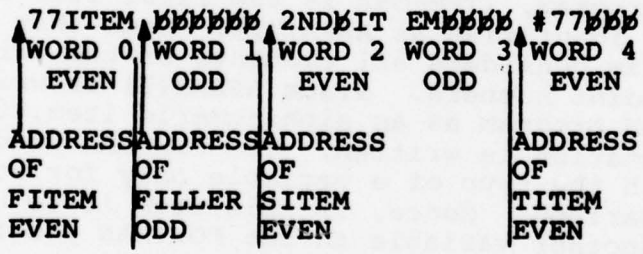
8.9.3 THE HONEYWELL PROGRAM. Attachment 9, Figure 1 is a listing of a short program that accomplishes the objective. Each item is discussed in detail below.

a. FIRST-ITEM. FIRST-ITEM is transferred to the FORTRAN program through the call. The item identification in the FORTRAN program corresponding to FIRST-ITEM is FITEM. Internal to the computer, these two variables correspond to the same address. "77ITEM" fits into one computer word, hence there is no problem with word boundary crossing. Since FIRST-ITEM is "display" in COBOL, it is equivalent to a "character" variable in FORTRAN. This version of the program is written in ANSI standard FORTRAN, which does not use character typing. Therefore, the bits in the address FITEM may be treated as real or integer variables. Real was chosen to accommodate a longer string of characters with double precision. The FORTRAN compiler considers

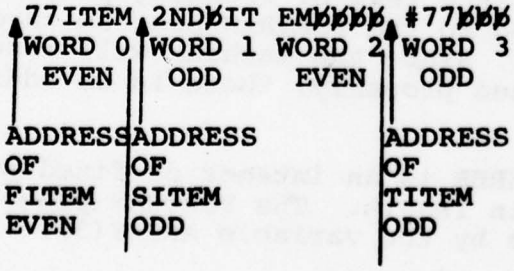
FITEM a real floating point number and, since no arithmetic operations or comparisons are being done, there is no problem. When FITEM is written to SYSOUT, it is redefined for output as a character variable by the FORMAT statement descriptor "A6". Hence, the alphanumeric form of the word is printed. As far as the compiler is concerned, FITEM has been redefined only for output. If FITEM was used again in the program, the compiler would consider FITEM a real variable. It would be possible to compare FITEM with another variable providing the other variable was typed as real. The programmer must keep these matters in mind when writing or maintaining the FORTRAN subroutine because the FORTRAN compiler will not resolve these items.

b. SECOND-ITEM. SECOND-ITEM is transferred to the FORTRAN program as SITEM. Since SECOND-ITEM occupies two words, SITEM must be defined as DOUBLE PRECISION. Unfortunately, double precision words begin on even word boundaries. SECOND-ITEM begins on an odd word boundary if it immediately follows FIRST-ITEM. Hence, a FILLER (blank) word is added after FIRST-ITEM to force SECOND-ITEM to an even boundary. If this FILLER was omitted, the value "77ITEM2ND⌀IT" would be at the location SITEM (at an odd address) because the Honeywell computer will begin at the odd word boundary and read the next lower address first (FIRST-ITEM is lower than SECOND-ITEM). The following diagrams should make this clear:

WITH FILLER



WITHOUT FILLER



c. THIRD-ITEM. THIRD-ITEM is transferred to the FORTRAN subroutine as TITEM. Since it occupies only one word the addressing is straight forward. Although the FORTRAN program considers TITEM a floating point number, the FORMAT statement for the WRITE instruction instructs the compiler to write TITEM as an alphanumeric item which is equivalent to the COBOL DISPLAY item.

d. FOURTH-ITEM. FOURTH-ITEM is the only computational or binary item in WORKING-STORAGE. The Honeywell COBOL COMP-1 usage is a fixed point number which is equivalent to a FORTRAN integer. Hence, when the FORTRAN equivalent to FOURTH-ITEM (ITEM4) is written, the compiler identifies ITEM4 as an integer. Binary items should never have address problems since the machine should always align these items properly regardless of the higher level language used.

e. ITEM-ONE. ITEM-ONE is the first item of an unorthodox table labeled TABLE-ONE. It is a display item that fits into one word. This item is passed to the FORTRAN program as the first item in a FORTRAN array. The FORTRAN array can be used to transfer character strings as well as tables. In this example, only items aligned within word boundaries are being transferred. The IBM equivalent to this program will demonstrate the problem of processing character strings that transcend word boundaries. Note that the entire table is transferred using one address. This address is the address of ITEM-ONE which is also the address of TABLE-ONE. Hence, for all practical purposes, the variables ITEM-ONE and TABLE-ONE are identical (see Attachment 9, Figures 1 and 2). In the FORTRAN program, ITEM-ONE is the first item in the FORTRAN array identified as ARRAY(1). Note that the FORTRAN program considers all elements of the array to be floating point numbers. Since ARRAY(1) is written by the FORTRAN program as an alphanumeric item, the correct representation is written. The FORTRAN program is concerned with the type of a variable only for computation and comparison. Hence, if this item is to be compared to another variable in the FORTRAN program, that other variable must be typed as REAL.

f. ITEM-TWO. ITEM-TWO is a real or floating point binary number. It is passed to the FORTRAN program through the array item ARRAY(2). Since the machine will insure that this number is aligned properly, there is no address or alignment problem.

g. ITEM-THREE. ITEM-THREE is an integer or fixed point number of one word in length. The FORTRAN program identifies this item by the variable ARRAY(3). Since the

FORTRAN program considers this variable a real or floating point item, any attempt to perform computations or comparisons will result in an erroneous operation that the computer will not flag as such. However, ARRAY(3) may be correctly written by specifying that it is an integer in the FORMAT statement. If computations or comparisons are to be performed by the FORTRAN program with ITEM-THREE, the item should be passed separately through a variable that the FORTRAN program types as an integer. Another way around this problem is to assign ARRAY(3) to a real variable and EQUIVALENCE that variable to an integer, e.g.,

```
EQUIVALENCE (X, INT)
      .
      .
      .
X = ARRAY(3)
```

Then the FORTRAN program will recognize INT as an integer variable with the appropriate value. Of these two "tricks-of-the-trade" the former method is much less confusing to someone unfamiliar with the program and is recommended for general use.

h. ITEM-FOUR. ITEM-FOUR must begin at an even address because it is a double precision floating point number. Notice that ITEM-THREE begins at an even address (even word boundary), therefore one word between the location of ITEM-THREE and ITEM-FOUR is not used. Hence, the FORTRAN program variable equivalent to ITEM-FOUR is ARRAY(5). ARRAY(4) is the unused word. ARRAY(5) is not recognized as double precision by the FORTRAN program. However it can be used as a single precision real number (with loss of precision) and written as a double precision item. Note that ARRAY(6) is the lower (least significant) part of the double precision mantissa. Since assigning ARRAY(5) to a double precision variable it will lose ARRAY(6) and EQUIVALENCE cannot be used with subroutine arguments, the only simple way to maintain the precision of ITEM-FOUR is to pass it as a separate double precision number through a variable defined as double precision in the FORTRAN subroutine.

i. ITEM-FIVE. ITEM-FIVE is a single precision floating point item that is recognized in the FORTRAN program by the variable ARRAY(7). This item was included to show that the double precision item ITEM-FOUR occupied ARRAY(5) and ARRAY(6). Note that the Honeywell COBOL compiler indicates the number of words occupied by a

COBOL level Ø1 group. This number of words corresponds to the DIMENSION of the FORTRAN array.

8.9.4 The IBM Program. Attachment 9, Figure 3 is a listing of the ARRAY 1 program that is essentially equivalent to the Honeywell program. The differences between the Honeywell and IBM program are due to the way the compilers handle data items, the lengths of the machine words and the IBM use of half word processing. Each item usage is detailed below.

a. FIRST-ITEM. Unlike the Honeywell program, FIRST-ITEM does not fit into a machine word on the IBM machine, but uses one word and half of the next. Hence, the FORTRAN variable FITEM must be defined as DOUBLE PRECISION. The compiler aligns the first item in a section on a double word boundary (address divisible by 8) which insures the item can be accessed properly by the FORTRAN program.

b. SECOND-ITEM. SECOND-ITEM begins immediately following the location of FIRST-ITEM. This item can be addressed by the FORTRAN produced program in spite of the fact that SECOND-ITEM is not a double word boundary and the FORTRAN variable SITEM is defined as double precision because the IBM 360/370 series of computers are byte oriented. Since the address of SECOND-ITEM is not on a double word boundary, the machine will align the double word length bit string in another storage area in the machine before operating with it. Hence no FILLER is necessary between FIRST-ITEM and SECOND-ITEM. Boundary alignment is desireable however.

c. THIRD-ITEM. Unlike the first two 77 items, THIRD-ITEM fits into one single word. Like these items, THIRD-ITEM is properly passed to the FORTRAN program by the byte addressing system.

d. FOURTH-ITEM. FOURTH-ITEM is a COMP item. It should begin on a half word boundary (address that is divisible by 2). In this case, it does not. Note that the Honeywell equivalent was typed COMP-1. The IBM COMP is a fixed point number like the Honeywell COMP-1. The Honeywell COMP is a floating point representation that may be treated as an integer. The IBM FORTRAN equivalent to the IBM COBOL COMP is a two byte precision integer, INTEGER*2. This FORTRAN integer is ITEM4. Since the address is not on a half word boundary, FOURTH-ITEM will be passed inefficiently to the FORTRAN produced program. Note that Honeywell does not use half word computational items.

e. ITEM-ONE. ITEM-ONE is the first item of the non-standard table called TABLE-ONE. Note that either TABLE-ONE or ITEM-ONE can be used to address the table, Attachment 9, Figure 4. Unfortunately "ARRAY" does not fit into one word. Therefore ARRAY was passed to the FORTRAN produced program through the array elements ARRAY(1) and ARRAY(2). Note that the data was written by writing ARRAY(1) as alphanumeric four (A4) and ARRAY(2) as alphanumeric two (A2). The "AY" in "ARRAY" is left justified in ARRAY(2).

f. ITEM-TWO. This item demonstrates some important concepts. A COMP SYNC item with a picture of S9(3) is a half word item. It should occupy the second half of ARRAY(2), but it could not be addressed by the FORTRAN program because the address of the half word is between ARRAY(2) and ARRAY(3). One solution to this problem would be to type ARRAY as a half word integer and increase the dimension of the FORTRAN program, but this solution would present greater problems with the transfer of subsequent items. Therefore ARRAY(2) is filled with two arbitrary characters and zeros are placed in the first two characters of ARRAY(3) by the COBOL FILLER statements. This filling forces ITEM-TWO into the last two bytes of ARRAY(3). The zeros in the first two bytes permit ITEM-TWO to be read by the FORTRAN program as ARRAY(3) in the correct integer format. In essence the half word binary integer, ITEM-TWO, was converted to a full word binary integer ARRAY(3). There would be a problem using ARRAY(3) in computations since it is typed, by default, as a real number. ARRAY could be typed as integer, but then other items passed through ARRAY would be considered integer by the FORTRAN program. Therefore, either ITEM-TWO must be passed separately for computations or ARRAY(3) "transferred" to an integer field as follows:

```
EQUIVALENCE (X, I)
      .
      .
      .
X = ARRAY(3)
```

Then I may be used in place of ARRAY(3), see listing. (Note that a variable cannot be equivalenced to a dummy variable in a subroutine call). The latter solution to this problem is discouraged because it is a bit esoteric.

g. ITEM-THREE. ITEM-THREE is a full word fixed point or integer number that is aligned on a full word boundary. The address is ARRAY(4) as should be expected. If this item is to be used for computation or comparison, it must be passed separately or reassigned as described in paragraph f.

h. ITEM-FOUR. This item is a double precision floating point number or REAL*8. It begins at ARRAY(5) which is at an address divisible by 8. Although this number can be written, any arithmetic or comparision would lose precision because the FORTRAN program recognizes ARRAY(5) as a single precision number. The best simple way to maintain the precision is to pass this item separately as a double precision item.

i. ITEM-FIVE. ITEM-FIVE is a single precision floating point number. It is on a full word boundary in the seventh word of the table and hence has the variable address: ARRAY(7) in the FORTRAN program.

8.9.5 Implications. Clearly, these simple programs present some interesting problems with important implications. The programs were not easy to debug even when the programmer knew the problems would occur. It is not possible to show all the errors that can occur when these programs are incorrectly written, but of great inporatance is the fact that INCONSISTANCIES BETWEEN CORRESPONDING DATA ITEMS OF DIFFERENT SUBPROGRAMS GENERATED BY DIFFERENT COMPILERS CAN CAUSE ERRONEOUS DATA TRANSFERS THAT ARE SELDOM DETECTED BY THE COMPUTER! The programmer must be very careful to insure the data items are aligned properly in all subprograms and that the USAGE or types are also correct. The following specifics are demonstrated by the programs ARRAY1.

a. DISPLAY or CHARACTER strings are stored in one continuous string unless a computational synchronized item intervenes.

b. The Honeywell machine cannot properly pass a DISPLAY item to a FORTRAN program through a DOUBLE PRECISION PROGRAM variable unless the COBOL item is aligned on a double word boundary. The IBM machine, since it is byte oriented, does not have this limitation, but it will handle the transfer more efficiently if the data is aligned on a double word boundary.

c. Computational binary items transferred through single variables (as opposed to arrays) should always have the boundaries properly aligned on both machines. Efficiency is improved on the IBM machine if computational items are properly aligned (in COBOL terms, the items are SYNCHRONIZED).

d. COBOL tables may be passed to FORTRAN through arrays using the table and array names, but alignment rules must be carefully followed on both the Honeywell and IBM machines.

e. Computational items passed to FORTRAN from COBOL programs, must be correctly identified to the FORTRAN program as to usage or type. If the type is not correct and cannot be changed, special techniques must be used (see paragraph 8.9g and 8.10f).

f. Double precision floating point numbers passed through an array to a FORTRAN program cannot be treated by FORTRAN as double precision number unless the entire array is typed double precision or special techniques are used.

g. FILLER fields may be necessary in COBOL tables to pass data in these tables to FORTRAN programs through arrays.

h. On the IBM machine, it may be desireable to convert a COBOL half word fixed point number to a full word FORTRAN integer (see paragraph 8.2.4f).

8.9.6 Other Analysis. Attachment 9, Figures 5 and 6 show some additional analysis of the IBM treatment of the program ARRAY1. The first SIX words of the COBOL WORKING-STORAGE section are printed by the FORTRAN program. Figure 5 shows the fourth item not synchronized whereas Figure 6 shows the difference synchronization makes. Also the technique of equivalencing real and integer variables in FORTRAN is demonstrated in both Figures. Figures 7 and 8 show the Job Control Language associated with the programs.

8.10 COBOL PROGRAM-ID Name Selection. The COBOL PROGRAM-ID name may consist of between 1 and 8 characters. Honeywell FORTRAN subprogram names may also contain 1-8 characters; however, IBM FORTRAN names may not exceed 6 characters. Therefore, a call from an IBM FORTRAN program to a COBOL program with a PROGRAM-ID containing more than 6 characters is not possible.

One solution (and probably the easiest) to this problem is to always use a PROGRAM-ID which contains 6 or fewer characters. A second solution would be to change PROGRAM-IDs which contain more than 6 characters to a name which contains 6 or less. This would be a valid solution unless other COBOL programs refer to the longer PROGRAM-ID. It may then become impractical to alter all other existing references to this longer PROGRAM-ID. To avoid this problem an ENTRY statement may be used (see paragraphs 6.2.6 and 6.3.3).

The ENTRY name should contain 6 or fewer characters and be inserted into the COBOL program immediately following the PROCEDURE DIVISION statement. For example

```
.
.
.
PROGRAM-ID.  COBOLSUB.
.
.
.
PROCEDURE DIVISION USING VARIABLES.
```

would be altered to

```
.
.
.
PROGRAM-ID.  COBOLSUB.
.
.
.
PROCEDURE DIVISION USING VARIABLES.

ENTRY 'COBSUB' USING VARIABLES.
```

It is possible that the COBOL routine cannot be altered (such as if the COBOL routine only existed in a compiled form). In this case the ENTRY point may not be used. To be able to call a COBOL program with a PROGRAM-ID containing more than 6 characters from IBM FORTRAN requires the use of an alternate method.

One method that may be used is to write a COBOL program with a PROGRAM-ID containing 6 or fewer characters. This COBOL program would in turn call the COBOL program which contains the longer name. For example, assume an IBM FORTRAN program needs to call the COBOL subroutine COMPUTE. In order to call COMPUTE a second COBOL routine (with a PROGRAM-ID containing 6 or fewer characters) could be written. This could appear as follows:

FORTRAN

```
.
.
.
CALL CALLER (A, B)
.
.
.
```

NEW COBOL SUBROUTINE:

```
        PROGRAM-ID.  CALLER.
        .
        .
        .
        PROCEDURE DIVISION USING X, Y.

        CALL 'COMPUTE' USING X, Y.
        .
        .
        .
```

ORIGINAL COBOL SUBROUTINE:

```
        .
        .
        .
        PROGRAM-ID.  CALLER.
        .
        .
        .
        PROCEDURE DIVISION USING Q, W.
```

8.11 Passing COBOL Group Items. COBOL allows a data structure which can be broken into substructures. This data structure is called a group item. Following is an example of a group item:

```
    Ø1  GROUP.
        Ø3  ITEM-1 PIC XX.
        Ø3  ITEM-2 COMP-2
        Ø3  ITEM-3 COMP-1.
```

The group item GROUP is subdivided into ITEM-1, ITEM-2, and ITEM-3. Notice that ITEM-1, ITEM-2, and ITEM-3 each have a different USAGE.

When a group item is passed to FORTRAN, the programmer must provide an array in FORTRAN of sufficient length to contain all of the words in the group item. FORTRAN, however, assumes that each element of the array is of the same data representation. The programmer must manipulate this data in some manner to divide this array into several items, each item having the correct data representation as the corresponding item in COBOL. This manipulation can become lengthy and confusing.

To eliminate this problem, do not pass group level items. Instead, pass each of the sub-items separately. For example, rather than using

```
CALL 'FORT' USING GROUP.
```

substitute

```
CALL 'FORT' USING ITEM-1, ITEM-2, ITEM-3.
```

The FORTRAN routine would then be altered from

```
SUBROUTINE (ARRAY)
```

to appear as

```
SUBROUTINE (ITEM1, ITEM2, ITEM3)
```

where ITEM1 has the same data representation as ITEM-1, ITEM2 as ITEM-2, etc. In this manner the data transfer is straight forward, and complicated manipulation is avoided.

8.12 The STUB Program. When writing programs and subroutines of any nature, it is always best to start at the simplest levels. One should begin with a short, basic program using simple data to make sure the concept of accomplishing some activity is properly satisfied. This little program is called a STUB. It is a particularly useful idea when the program will eventually link with another program, especially if the other program is written in another language. After the stub program has been made to link properly on an uncomplicated level, it can be expanded and the data may become more complex, testing and debugging at each level of sophistication, until the program is complete and runs smoothly using the real data, records or files required of production programs.

ATTACHMENT 1

1. This attachment contains a sample listing of the IBM JCL necessary to compile an IBM COBOL program and a COBOL subroutine. Also included is the source listing and the program execution.

2. STEP1 of the JCL uses VSCOBC to compile the subroutine. STEP2 uses VSCOBCLG to compile the driver and execute the program. Since the first program compiled is not the driver a //LKED.SYSIN DD * card is used to specify the name of the driver (in this case COBSHELL). An ENTRY COBSHELL card follows the //LKED.SYSIN to specify COBSHELL as the driver routine. Had the driver been the first routine to be compiled the //LKED.SYSIN and the ENTRY card would not be required. (Compare the ENTRY card JCL to the JCL in attachment 2).

3. Following the JCL is the source code for the subroutine. Note the use of the LINKAGE SECTION following the WORKING-STORAGE SECTION. The PROCEDURE DIVISION USING...is also used in this subroutine.

4. The source code for the driver demonstrates the use of the CALL statement. The items being passed in the call are defined in WORKING-STORAGE. This definition matches the definition of the items in the LINKAGE SECTION of the subroutine. Through the use of the CALL data are passed to the subroutine.

5. The last items listed are the output from the program execution.

```
 2      //STEP1     EXEC  VSCOBC
 3      XXVSCOBC   PROC CT=0030,CR=192K,
        XX    COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB'
        ***
        ***         VS/COBOL COMPILER PROGRAM PRODUCT
        ***
        ***         IBM PP 5740-CB1  RELEASE 2.2 W/PTF 6
        ***
 4      XXCOB      EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
 5      XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
 6      XXSYSLIB   DD DSN=&COPY,DISP=SHR
 7      XX         DD DSN=&COPY1,DISP=SHR
 8      XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
 9      XXSYSPRINT DD SYSOUT=A
10      XXSYSUDUMP DD SYSOUT=A
11      XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
12      XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
13      XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
14      XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
15      //COB.SYSIN DD    *
16      //STEP2     EXEC  VSCOBCLG
17      XXVSCOBCLG PROC CT=0030,CR=192K,LT=0030,LR=156K,
        XX GT=0030,GR=156K,
        XX COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB',
        XX LINK='SYS1.PGMPROD.VSCOBLIB'
        ***
        ***         VS/COBOL COMPILER PROGRAM PRODUCT
        ***
        ***         IBM PP 5740-CB1  RELEASE 2.2 W/PTF 6
        ***
18      XXCOB      EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
19      XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
20      XXSYSLIB   DD DSN=&COPY,DISP=SHR
21      XX         DD DSN=&COPY1,DISP=SHR
22      XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
23      XXSYSPRINT DD SYSOUT=A
24      XXSYSUDUMP DD SYSOUT=A
25      XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
26      XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
27      XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
28      XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
29      //COB.SYSIN DD    *
30      XXLKED     EXEC PGM=IEWL,REGION=&LR,TIME=&LT,COND=(5,LT,COB),
        XX            PARM=MAP
31      XXSYSLIB   DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
32      XX         DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
33      XX         DD DSN=&LINK,DISP=SHR
34      XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
35      XX         DD DDNAME=SYSIN
```

```
36      XXSYSLMOD  DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(6144,(76,8,1))
37      XXSYSPRINT DD SYSOUT=A
        XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIODS,
        XX            SPACE=(1024,(200,11))
39      //LKED.SYSIN DD   *
40      XXGO       EXEC PGM=*.LKED.SYSLMOD,REGION=&GR,TIME=&GT,
        XX            COND=((9,LT,LKED),(5,LT,COB))
41      XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELETE)
42      XXSYSDBOUT DD SYSOUT=A
43      XXSYSOUT   DD SYSOUT=A
44      //GO.READER DD    *
45      //GO.PRINTER DD   SYSOUT=A
        //
```

ENTRY COBSHELL

```
IDENTIFICATION DIVISION.

PROGRAM-ID.     SHELLS.

AUTHOR.

DATE-WRITTEN.  14 APRIL 1978.

DATE-COMPILED. FEB  6,1979.

REMARKS.        SHELL  SORT
    SHELL SORT, INSTEAD OF ALWAYS COMPARING ADJACENT KEYS, ONE
    COMPARES AND EXCHANGES KEYS THAT MAY BE FAR APART TO BEGIN
    WITH. THE DISORDER OF A FILE IS DEFINED TO BE THE AVERAGE
    DISTANCE A RECORD IS FROM THE LOCATION IT WILL ULTIMATELY
    OCCUPY IN THE SORTED FILE. ONE APPROACH TO THE DESIGN OF A
    SORTING ALGORITHM IS TO AIM AT A REDUCTION OF THIS MEASURE
    TO ITS ULTIMATE VALUE OF ZERO IN A SYSTEMATIC MANNER. SHELL
    SORT EXEMPLIFIES THIS APPROACH.
```

```
ENVIRONMENT     DIVISION.

CONFIGURATION   SECTION.

  SOURCE-COMPUTER.  IBM-370.
  OBJECT-COMPUTER.  IBM-370.


DATA  DIVISION.

WORKING-STORAGE SECTION.

77  I                                  PIC S999     VALUE ZERO COMP SYNC.
77  J                                  PIC S999     VALUE ZERO COMP SYNC.
77  THE-SENTINEL-ELEMENT               PIC S999     VALUE ZERO COMP SYNC.
77  THE-DISORDER-OF-THE-FILE       PIC  S999 VALUE +1   COMP-3.
77  NUMBER-OF-PASSES-TO-SORT-FILE PIC  S999 VALUE ZERO COMP-3.
77  TEMPORARY-STORAGE              PIC S999V99 VALUE ZERO.


LINKAGE      SECTION.

77  NUMBER-ELEMENTS-IN-THE-ARRAY PIC 999                    COMP-3.

01  ARRAY-TABLE.
    02  ARRAY-ELEMENT              PIC S999V99 OCCURS 100 TIMES.
```

```
PROCEDURE   DIVISION
        USING NUMBER-ELEMENTS-IN-THE-ARRAY, ARRAY-TABLE.


DETERMINE-THE-FILE-DISORDER.
    COMPUTE THE-DISORDER-OF-THE-FILE =
        2 * THE-DISORDER-OF-THE-FILE.
    IF THE-DISORDER-OF-THE-FILE IS LESS THAN
        NUMBER-ELEMENTS-IN-THE-ARRAY OR EQUAL TO
        NUMBER-ELEMENTS-IN-THE-ARRAY
        GO TO DETERMINE-THE-FILE-DISORDER.


DETERMINE-NEW-DISORDER.
    COMPUTE THE-DISORDER-OF-THE-FILE =
        (THE-DISORDER-OF-THE-FILE - 1) / 2.
    IF THE-DISORDER-OF-THE-FILE IS EQUAL TO ZERO
        GO TO SORT-COMPLETE.
    COMPUTE NUMBER-OF-PASSES-TO-SORT-FILE =
        NUMBER-ELEMENTS-IN-THE-ARRAY - THE-DISORDER-OF-THE-FILE.


SORT-THE-ARRAY.
    PERFORM SHELL-SORT THRU DO-NOT-SORT
        VARYING I FROM 1 BY 1 UNTIL I EQUAL TO
        NUMBER-OF-PASSES-TO-SORT-FILE.
    GO TO DETERMINE-NEW-DISORDER.


SORT-COMPLETE.
    EXIT PROGRAM.
```

```
SHELL-SORT.
    MOVE I TO J.
  DETERMINE-IF-ARRAY-IN-ORDER.
    COMPUTE THE-SENTINEL-ELEMENT = J + THE-DISORDER-OF-THE-FILE.
    IF ARRAY-ELEMENT (THE-SENTINEL-ELEMENT)
        IS GREATER THAN ARRAY-ELEMENT (J)
       OR IS EQUAL TO ARRAY-ELEMENT (J)
       GO TO DO-NOT-SORT.
    MOVE ARRAY-ELEMENT (J) TO TEMPORARY-STORAGE.
    MOVE ARRAY-ELEMENT (THE-SENTINEL-ELEMENT) TO
         ARRAY-ELEMENT (J).
    MOVE TEMPORARY-STORAGE TO
        ARRAY-ELEMENT (THE-SENTINEL-ELEMENT).
    COMPUTE J = J - THE-DISORDER-OF-THE-FILE.
    IF J IS GREATER THAN ZERO GO TO DETERMINE-IF-ARRAY-IN-ORDER.

DO-NOT-SORT.
    EXIT.
```

```
       IDENTIFICATION DIVISION.

       PROGRAM-ID.    COBSHELL

       AUTHOR.

       DATE-WRITTEN.  MAR 30 1978.

       DATE-COMPILED. FEB  6,1979.

       REMARKS.    SHELL SORT DRIVER
            THIS PROGRAM IS THE DRIVER FOR PERFORMING A SHELL SORT.
           THIS PROGRAM USES THE CALL VERB TO CALL A SUBPROGRAM OR
           SUBROUTINE TO SORT AN ARRAY OF NUMBERS.
```

```
ENVIRONMENT     DIVISION.

CONFIGURATION   SECTION.

  SOURCE-COMPUTER.  IBM-370.
  OBJECT-COMPUTER.  IBM-370.
  SPECIAL-NAMES.    C01 IS TO-TOP-OF-PAGE.

INPUT-OUTPUT    SECTION.

  FILE-CONTROL.
    SELECT INPUT-DATA-FILE ASSIGN TO UR-S-READER.
    SELECT PRINT-FILE      ASSIGN TO UR-S-PRINTER.
```

```
DATA  DIVISION.

FILE  SECTION.

FD  INPUT-DATA-FILE
    LABEL RECORDS ARE OMITTED.

01  INPUT-CARD.
    02  ARRAY-DATA                  PIC S999V99.
    02  FILLER                      PIC X(75).

FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED.

01  A-LINE-OF-PRINT                 PIC X(133).

WORKING-STORAGE SECTION.

77  I                               PIC S999    VALUE ZERO COMP SYNC.
77  J                               PIC S999    VALUE ZERO COMP SYNC.
77  M                               PIC S999    VALUE ZERO COMP SYNC.
77  NUMBER-ELEMENTS-IN-THE-ARRAY PIC S999   VALUE ZERO COMP-3.

01  ARRAY-TABLE.
    02  ARRAY-ELEMENT               PIC S999V99 OCCURS 100 TIMES.

01  SORT-HEADING.
    02  FILLER                      PIC X       VALUE SPACE.
    02  FILLER                      PIC X(32)   VALUE 'THE FOLLOWING
-   'NUMBERS ARE SORTED'.
    02  FILLER                      PIC X(100)  VALUE SPACES.

01  PRINT-RECORD.
    02  FILLER                      PIC X(7)    VALUE SPACES.
    02  PRINT-ARRAY-ELEMENT         PIC ---9.99.
```

```
PROCEDURE  DIVISION.

SET-UP.
    OPEN INPUT INPUT-DATA-FILE  OUTPUT PRINT-FILE.

READ-AN-ELEMENT-INTO-THE-ARRAY.
    ADD 1 TO I, NUMBER-ELEMENTS-IN-THE-ARRAY.
    READ INPUT-DATA-FILE RECORD
        AT END
          GO TO SHELL-SORT-THE-ARRAY.
    MOVE ARRAY-DATA TO PRINT-ARRAY-ELEMENT, ARRAY-ELEMENT (I).
    WRITE A-LINE-OF-PRINT FROM PRINT-RECORD
        AFTER ADVANCING 1 LINES.
    GO TO READ-AN-ELEMENT-INTO-THE-ARRAY.

SHELL-SORT-THE-ARRAY.
    CALL 'SHELLS'
        USING NUMBER-ELEMENTS-IN-THE-ARRAY, ARRAY-TABLE.

WRITE-THE-SORTED-ARRAY.
    WRITE A-LINE-OF-PRINT FROM SORT-HEADING
        AFTER ADVANCING TO-TOP-OF-PAGE.
    PERFORM WRITE-SORTED-NUMBERS
        VARYING M FROM 1 BY 1 UNTIL M IS EQUAL TO
        NUMBER-ELEMENTS-IN-THE-ARRAY.

    STOP RUN.
```

```
WRITE-SORTED-NUMBERS.
    MOVE ARRAY-ELEMENT (M) TO PRINT-ARRAY-ELEMENT.
    WRITE A-LINE-OF-PRINT FROM PRINT-RECORD
        AFTER ADVANCING 1 LINES.
```

0.15
123.45
0.00
0.50
600.00
0.01

THE FOLLOWING NUMBERS ARE SORTED

0.00
0.01
0.15
0.50
123.45
600.00

ATTACHMENT 2

1. This attachment contains a sample listing of the IBM JCL necessary to compile an IBM COBOL program and a COBOL subroutine. Also included is the source listing and the program execution.

2. The DRIVER step compiles the COBOL driver routine using VSCOBC. The SUBRTNE step compiles and executes the subroutine using VSCOBCLG. Since no //LKED.SYSIN card is used the computer assumes the first routine compiled is the driver. (Compare this with the JCL in attachment 1).

3. Following the JCL is the source code for the driver which contains the call statement. The driver is identical to the driver found in attachment 1.

4. Next is the source code for the subroutine. Again, this is identical to the subroutine in attachment 1.

5. The last items listed are the output from the program execution.

```
 2      //DRIVER   EXEC  VSCOBC
 3      XXVSCOBC   PROC CT=0030,CR=192K,
        XX    COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB'
        ***
        ***        VS/COBOL COMPILER PROGRAM PRODUCT
        ***
        ***        IBM PP 5740-CB1  RELEASE 2.2 W/PTF 6
        ***
 4      XXCOB      EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
 5      XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
 6      XXSYSLIB   DD DSN=&COPY,DISP=SHR
 7      XX         DD DSN=&COPY1,DISP=SHR
 8      XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
 9      XXSYSPRINT DD SYSOUT=A
10      XXSYSUDUMP DD SYSOUT=A
11      XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
12      XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
13      XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
14      XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
15      //COB.SYSIN DD    *
16      //SUBRTNE  EXEC  VSCOBCLG
17      XXVSCOBCLG PROC CT=0030,CR=192K,LT=0030,LR=156K,
        XX GT=0030,GR=156K,
        XX COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB',
        XX LINK='SYS1.PGMPROD.VSCOBLIB'
        ***
        ***        VS/COBOL COMPILER PROGRAM PRODUCT
        ***
        ***        IBM PP 5740-CB1  RELEASE 2.2 W/PTF 6
        ***
18      XXCOB      EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
19      XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
20      XXSYSLIB   DD DSN=&COPY,DISP=SHR
21      XX         DD DSN=&COPY1,DISP=SHR
22      XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
23      XXSYSPRINT DD SYSOUT=A
24      XXSYSUDUMP DD SYSOUT=A
25      XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
26      XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
27      XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
28      XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
29      //COB.SYSIN DD    *
30      XXLKED     EXEC PGM=IEWL,REGION=&LR,TIME=&LT,COND=(5,LT,COB),
        XX            PARM=MAP
31      XXSYSLIB   DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
32      XX         DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
33      XX         DD DSN=&LINK,DISP=SHR
34      XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
35      XX         DD DDNAME=SYSIN
```

Attachment 2

```
      XX              SPACE=(6144,(76,8,1))
37    XXSYSPRINT DD SYSOUT=A
38    XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIODS,
      XX              SPACE=(1024,(200,11))
3     XXGO       EXEC PGM=*.LKED.SYSLMOD,REGION=&GR,TIME=&GT,
      XX              COND=((9,LT,LKED),(5,LT,COB))
40    XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELETE)
41    XXSYSDBOUT DD SYSOUT=A
42    XXSYSOUT   DD SYSOUT=A
43    //GO.READER DD     *
44    //GO.PRINTER DD    SYSOUT=A
      //
```

```
       IDENTIFICATION DIVISION.

       PROGRAM-ID.    COBSHELL

       AUTHOR.

       DATE-WRITTEN.  MAR 30 1978.

       DATE-COMPILED. FEB  5,1979.

       REMARKS.    SHELL SORT DRIVER
             THIS PROGRAM IS THE DRIVER  FOR PERFORMING A SHELL SORT.
          THIS PROGRAM USES THE CALL VERB TO CALL A SUBPROGRAM OR
          SUBROUTINE TO SORT AN ARRAY OF NUMBERS.
```

```
ENVIRONMENT     DIVISION.

CONFIGURATION   SECTION.

  SOURCE-COMPUTER.  IBM-370.
  OBJECT-COMPUTER.  IBM-370.
  SPECIAL-NAMES.    C01 IS TO-TOP-OF-PAGE.

INPUT-OUTPUT    SECTION.

  FILE-CONTROL.
    SELECT INPUT-DATA-FILE ASSIGN TO UR-S-READER.
    SELECT PRINT-FILE      ASSIGN TO UR-S-PRINTER.
```

```
DATA  DIVISION.

FILE  SECTION.

FD  INPUT-DATA-FILE
    LABEL RECORDS ARE OMITTED.

01  INPUT-CARD.
    02  ARRAY-DATA                  PIC S999V99.
    02  FILLER                      PIC X(75).

FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED.

01  A-LINE-OF-PRINT                 PIC X(133).


WORKING-STORAGE SECTION.

77  I                               PIC S999    VALUE ZERO COMP SYNC.
77  J                               PIC S999    VALUE ZERO COMP SYNC.
77  M                               PIC S999    VALUE ZERO COMP SYNC.
77  NUMBER-ELEMENTS-IN-THE-ARRAY    PIC S999    VALUE ZERO COMP-3.

01  ARRAY-TABLE.
    02  ARRAY-ELEMENT               PIC S999V99 OCCURS 100 TIMES.

01  SORT-HEADING.
    02  FILLER                      PIC X       VALUE SPACE.
    02  FILLER                      PIC X(32)   VALUE 'THE FOLLOWING
-   'NUMBERS ARE SORTED'.
    02  FILLER                      PIC X(100)  VALUE SPACES.

01  PRINT-RECORD.
    02  FILLER                      PIC X(7)    VALUE SPACES.
    02  PRINT-ARRAY-ELEMENT         PIC ---9.99.
```

```
PROCEDURE  DIVISION.


SET-UP.
    OPEN INPUT INPUT-DATA-FILE  OUTPUT PRINT-FILE.


READ-AN-ELEMENT-INTO-THE-ARRAY.
    ADD 1 TO I, NUMBER-ELEMENTS-IN-THE-ARRAY.
    READ INPUT-DATA-FILE RECORD
        AT END
          GO TO SHELL-SORT-THE-ARRAY.
    MOVE ARRAY-DATA TO PRINT-ARRAY-ELEMENT, ARRAY-ELEMENT (I).
    WRITE A-LINE-OF-PRINT FROM PRINT-RECORD
        AFTER ADVANCING 1 LINES.
    GO TO READ-AN-ELEMENT-INTO-THE-ARRAY.


SHELL-SORT-THE-ARRAY.
    CALL 'SHELLS'
        USING NUMBER-ELEMENTS-IN-THE-ARRAY, ARRAY-TABLE.


WRITE-THE-SORTED-ARRAY.
    WRITE A-LINE-OF-PRINT FROM SORT-HEADING
        AFTER ADVANCING TO-TOP-OF-PAGE.
    PERFORM WRITE-SORTED-NUMBERS
        VARYING M FROM 1 BY 1 UNTIL M IS EQUAL TO
        NUMBER-ELEMENTS-IN-THE-ARRAY.


    STOP RUN.
```

```
WRITE-SORTED-NUMBERS.
    MOVE ARRAY-ELEMENT (M) TO PRINT-ARRAY-ELEMENT.
    WRITE A-LINE-OF-PRINT FROM PRINT-RECORD
        AFTER ADVANCING 1 LINES.
```

17.12.41 FEB 5,1979

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    SHELLS.

AUTHOR.

DATE-WRITTEN.  14 APRIL 1978.

DATE-COMPILED. FEB  5,1979.

REMARKS.       SHELL  SORT
    SHELL SORT, INSTEAD OF ALWAYS COMPARING ADJACENT KEYS, ONE
    COMPARES AND EXCHANGES KEYS THAT MAY BE FAR APART TO BEGIN
    WITH. THE DISORDER OF A FILE IS DEFINED TO BE THE AVERAGE
    DISTANCE A RECORD IS FROM THE LOCATION IT WILL ULTIMATELY
    OCCUPY IN THE SORTED FILE. ONE APPROACH TO THE DESIGN OF A
    SORTING ALGORITHM IS TO AIM AT A REDUCTION OF THIS MEASURE
    TO ITS ULTIMATE VALUE OF ZERO IN A SYSTEMATIC MANNER. SHELL
    SORT EXEMPLIFIES THIS APPROACH.
```

```
ENVIRONMENT     DIVISION.

CONFIGURATION   SECTION.

  SOURCE-COMPUTER.  IBM-370.
  OBJECT-COMPUTER.  IBM-370.


DATA  DIVISION.

WORKING-STORAGE SECTION.

77  I                                PIC S999     VALUE ZERO COMP SYNC.
77  J                                PIC S999     VALUE ZERO COMP SYNC.
77  THE-SENTINEL-ELEMENT             PIC S999     VALUE ZERO COMP SYNC.
77  THE-DISORDER-OF-THE-FILE           PIC  S999 VALUE +1   COMP-3.
77  NUMBER-OF-PASSES-TO-SORT-FILE PIC  S999 VALUE ZERO COMP-3.
77  TEMPORARY-STORAGE                PIC S999V99 VALUE ZERO.


LINKAGE      SECTION.

77  NUMBER-ELEMENTS-IN-THE-ARRAY PIC 999                    COMP-3.

01  ARRAY-TABLE.
    02  ARRAY-ELEMENT             PIC S999V99 OCCURS 100 TIMES.
```

```
PROCEDURE   DIVISION
        USING NUMBER-ELEMENTS-IN-THE-ARRAY, ARRAY-TABLE.


DETERMINE-THE-FILE-DISORDER.
    COMPUTE THE-DISORDER-OF-THE-FILE =
        2 * THE-DISORDER-OF-THE-FILE.
    IF THE-DISORDER-OF-THE-FILE IS LESS THAN
        NUMBER-ELEMENTS-IN-THE-ARRAY OR EQUAL TO
        NUMBER-ELEMENTS-IN-THE-ARRAY
        GO TO DETERMINE-THE-FILE-DISORDER.


DETERMINE-NEW-DISORDER.
    COMPUTE THE-DISORDER-OF-THE-FILE =
        (THE-DISORDER-OF-THE-FILE - 1) / 2.
    IF THE-DISORDER-OF-THE-FILE IS EQUAL TO ZERO
        GO TO SORT-COMPLETE.
    COMPUTE NUMBER-OF-PASSES-TO-SORT-FILE =
        NUMBER-ELEMENTS-IN-THE-ARRAY - THE-DISORDER-OF-THE-FILE.


SORT-THE-ARRAY.
    PERFORM SHELL-SORT THRU DO-NOT-SORT
        VARYING I FROM I BY I UNTIL I EQUAL TO
        NUMBER-OF-PASSES-TO-SORT-FILE.
    GO TO DETERMINE-NEW-DISORDER.


SORT-COMPLETE.
    EXIT PROGRAM.
```

```
SHELL-SORT.
    MOVE 1 TO J.
  DETERMINE-IF-ARRAY-IN-ORDER.
    COMPUTE THE-SENTINEL-ELEMENT = J + THE-DISORDER-OF-THE-FILE.
    IF ARRAY-ELEMENT (THE-SENTINEL-ELEMENT)
        IS GREATER THAN ARRAY-ELEMENT (J)
        OR IS EQUAL TO ARRAY-ELEMENT (J)
        GO TO DO-NOT-SORT.
    MOVE ARRAY-ELEMENT (J) TO TEMPORARY-STORAGE.
    MOVE ARRAY-ELEMENT (THE-SENTINEL-ELEMENT) TO
         ARRAY-ELEMENT (J).
    MOVE TEMPORARY-STORAGE TO
        ARRAY-ELEMENT (THE-SENTINEL-ELEMENT).
    COMPUTE J = J - THE-DISORDER-OF-THE-FILE.
    IF J IS GREATER THAN ZERO GO TO DETERMINE-IF-ARRAY-IN-ORDER.

DO-NOT-SORT.
    EXIT.
```

0.15
123.45
0.00
0.50
600.00
0.01

THE FOLLOWING NUMBERS ARE SORTED

0.00
0.01
0.15
0.50
123.45
600.00

ATTACHMENT 3

1. This attachment contains a sample listing of the IBM JCL necessary to compile an IBM FORTRAN/COBOL mixed program. Also included is the source listing and the program execution.

2. Step FORTRAN compiles the FORTRAN portion of the program using FTG1C. The COBOL step compiles the COBOL portion of the job, and executes the entire program, using VSCOBCLG. Since no //LKED.SYSIN card is used, the first program compiled (in this case, the FORTRAN program) is assumed to be the driver.

3. Following the COBOL program is a //GO.FTØ6FØØ1 DD SYSOUT=A card. This card is necessary for the FORTRAN program execution. During a FORTRAN execution using FTG1CLG this card is normally supplied for printer output. However, neither the FTG1C, nor any COBOL JCL procedure provide this card. It is, therefore, necessary for the programmer to supply this card if the program assumes file code Ø6 is the printer.

4. Following the JCL is the FORTRAN source code. Note that the

```
CALL COBSUB(I)
```

statement is calling the COBOL subroutine named COBSUB.

5. The COBOL source code contains the LINKAGE SECTION which is required when passing values to the COBOL subroutine. The data items defined in the LINKAGE SECTION agree in type with the data items being passed from FORTRAN. The PROCEDURE DIVISION USING... clause is also used in the COBOL subroutine.

6. Following the COBOL source code is the output from the execution. The array, as passed from FORTRAN, is listed followed by the array as received by COBOL. Notice the array structures are different between COBOL and FORTRAN; i.e., FORTRAN rows are COBOL columns and FORTRAN columns are COBOL rows.

```
 2      //FORTRAN  EXEC  FTGIC,PARM.FORT='NOLIST'
 3      XXFTGIC  PROC CT=0030,CR=128K
 4      XXFORT   EXEC PGM=IGIFORT,REGION=&CR,TIME=&CT
 5      XXSYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX              SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
 6      XXSYSPRINT DD SYSOUT=A
 7      XXSYSTERM DD SYSOUT=A
 8      XXSYSUDUMP   DD SYSOUT=A
 9      //FORT.SYSIN  DD  *
10      //COBOL    EXEC  VSCOBCLG
11      XXVSCOBCLG PROC CT=0030,CR=192K,LT=0030,LR=156K,
        XX GT=0030,GR=156K,
        XX COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB',
        XX LINK='SYS1.PGMPROD.VSCOBLIB'
        ***
        ***        VS/COBOL COMPILER PROGRAM PRODUCT
        ***
        ***        IBM PP 5740-CB1  RELEASE 2.2 W/PTF 6
        ***
12      XXCOB    EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
13      XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
14      XXSYSLIB   DD DSN=&COPY,DISP=SHR
15      XX         DD DSN=&COPY1,DISP=SHR
16      XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
17      XXSYSPRINT DD SYSOUT=A
18      XXSYSUDUMP DD SYSOUT=A
19      XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
20      XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
21      XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
22      XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
        XX            UNIT=VIODS
23      //COB.SYSIN  DD  *
24      XXLKED   EXEC PGM=IEWL,REGION=&LK,TIME=&LT,COND=(5,LT,COB),
        XX            PARM=MAP
25      XXSYSLIB   DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
26      XX         DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
27      XX         DD DSN=&LINK,DISP=SHR
28      XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
29      XX         DD DDNAME=SYSIN
30      XXSYSLMOD  DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
        XX            SPACE=(6144,(76,8,1))
31      XXSYSPRINT DD SYSOUT=A
32      XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIODS,
        XX            SPACE=(1024,(200,11))
33      XXGO     EXEC PGM=*.LKED.SYSLMOD,REGION=&GR,TIME=&GT,
        XX            COND=((9,LT,LKED),(5,LT,COB))
34      XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELETE)
35      XXSYSOBOUT DD SYSOUT=A
36      XXSYSOUT   DD SYSOUT=A
37      //GO.FT06F001 DD SYSOUT=A
38      //GO.OUTPUT DD SYSOUT=A
        //
```

```
FORTRAN IV G1  RELEASE 2.0          MAIN                    DATE = 79066

0001                    DIMENSION I(3,4)
0002                    J = 0
0003                    DO 10 K = 1, 3
0004                    DO 20 M = 1, 4
0005                    J = J + 1
0006                    I(K, M) = J
0007                 20 CONTINUE
0008                 10 CONTINUE
0009                    WRITE (6, 30)
0010                 30 FORMAT (' FORTRAN:')
0011                    DO 40 J = 1, 3
0012                 40 WRITE (6, 50) (I(J,1), I(J,2), I(J, 3), I(J,4)
0013                 50 FORMAT (4(5H     , I5))
0014                    CALL COBSUB(I)
0015                    STOP
0016                    END
```

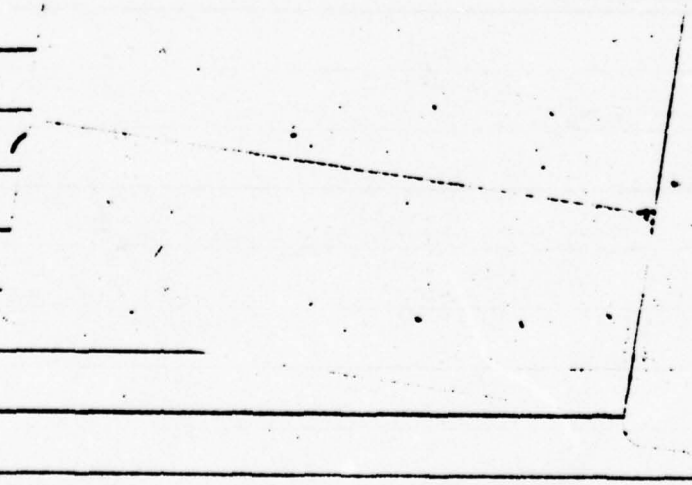
```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.  COBSUB.
00003          AUTHOR.
00004          ENVIRONMENT DIVISION.
00005          CONFIGURATION SECTION.
00006          SOURCE-COMPUTER.  IBM-3033.
00007          OBJECT-COMPUTER.  IBM-3033.
00008          INPUT-OUTPUT SECTION.
00009          FILE-CONTROL.
00010              SELECT OUTPUT-FILE ASSIGN TO UR-S-OUTPUT.
00011          DATA DIVISION.
00012          FILE SECTION.
00013          FD  OUTPUT-FILE LABEL RECORDS ARE OMITTED.
00014          01  A-LINE-OF-PRINT PIC X(133).
00015          WORKING-STORAGE SECTION.
00016          77  FIRST-SUBSCRIPT  PIC S99 COMP SYNC.
00017          01  OUTPUT-RECORD.
00018              03  FILLER PIC X(5) VALUE SPACES.
00019              03  OUT-1  PIC 9(6).
00020              03  FILLER PIC X(5) VALUE SPACES.
00021              03  OUT-2  PIC 9(6).
00022              03  FILLER PIC X(5) VALUE SPACES.
00023              03  OUT-3  PIC 9(6).
00024              03  FILLER PIC X(122) VALUE SPACES.
00025          LINKAGE SECTION.
00026          01  ARRAY-TABLE.
00027          03  MATRIX  OCCURS 4 TIMES.
00028                  05  COLUMNS PIC 9(6) COMP OCCURS 3 TIMES.
00029          PROCEDURE DIVISION USING ARRAY-TABLE.
00030              OPEN OUTPUT OUTPUT-FILE.
00031              MOVE '  COBOL ARRAY:' TO OUTPUT-RECORD.
00032              WRITE A-LINE-OF-PRINT FROM OUTPUT-RECORD AFTER ADVANCING
00033                  1 LINES.
00034              MOVE SPACES TO OUTPUT-RECORD.
00035              PERFORM MOVE-AND-WRITE THRU MOVE-AND-WRITE-EXIT VARYING
00036                  FIRST-SUBSCRIPT FROM 1 BY 1 UNTIL FIRST-SUBSCRIPT IS
00037                  GREATER THAN 4.
00038              CLOSE OUTPUT-FILE.
00039          EXIT-PROGRAM.
00040              EXIT PROGRAM.
00041          MOVE-AND-WRITE.
00042              MOVE COLUMNS (FIRST-SUBSCRIPT, 1) TO OUT-1.
00043              MOVE COLUMNS (FIRST-SUBSCRIPT, 2) TO OUT-2.
00044              MOVE COLUMNS (FIRST-SUBSCRIPT, 3) TO OUT-3.
00045              WRITE A-LINE-OF-PRINT FROM OUTPUT-RECORD AFTER ADVANCING
00046                  1 LINES.
00047          MOVE-AND-WRITE-EXIT.
00048              EXIT.
```

FORTRAN:

1	2	3	4
5	6	7	8
9	10	11	12

COBOL ARRAY:

000001	000005	000009
000002	000006	000010
000003	000007	000011
000004	000008	000012



ATTACHMENT 4

1. This attachment contains a sample listing of the IBM JCL necessary to compile an IBM FORTRAN/COBOL mixed program. Also included is the source listing and program execution.

2. The FTSUB step compiles the FORTRAN portion of the program using FTG1C. The COBOL driver is compiled, and the entire program executed using VSCOBCLG. The second program compiled is the driver so a //LKED.SYSIN card is used (see attachment 1). An FTØ6FØØ1 card (see attachment 3) is added for the FORTRAN program execution. A temporary file, FTØ8FØØ1, is also supplied.

3. The FORTRAN subroutine is next listed. This program reads the temporary file (file code Ø8), changes their values, writes the changed values on the printer, then rewrites the values on the temporary file.

4. Following the FORTRAN subroutine is the COBOL source code. The CALL 'FTSUB' statement in COBOL calls the FORTRAN routine. This COBOL driver reads and writes on the temporary file also. Note the

```
SELECT TSTFYL ASSIGN TO UT-S-FTØ8FØØ1
```

statement uses the FORTRAN name FTØ8FØØ1 for the temporary file. COBOL demands this file name begin with a letter. FORTRAN, however, uses a numeric designator in a READ or WRITE statement which FORTRAN relates to the JCL. For example,

```
READ (nn,10)
```

in FORTRAN relates the file code nn to a JCL DD card FTnnFØØ1. This fact is not easily avoidable in FORTRAN. It is, therefore, easiest to refer to the file in COBOL by using the FORTRAN name.

5. Before the CALL 'FTSUB' statement is a CLOSE statement which closes both PRTOUT and TSTFYL. TSTFYL is the same temporary file that the FORTRAN subroutine will read and write, and is closed to prevent FORTRAN from trying to open the file which COBOL already has open.

```
//FTSUB EXEC FTG1C,PARM.FORT='NOLIST'
XXFTG1C   PROC GT=0030,CR=1260K
XXFORT    EXEC PGM=IGIFORT,REGION=&CR,TIME=&GT
XXSYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
XX           SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
XXSYSPRINT DD SYSOUT=A
XXSYSTERM DD SYSOUT=A
XXSYSUDUMP DD SYSOUT=A
//FORT.SYSIN  DD  *
//GOMAIN EXEC VSCOBCLG
XXVSCOBCLG PROC GT=0030,CR=1720K,LT=0030,LR=150K,
XX GT=0030,CR=150K,
XX COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB',
XX LINK='SYS1.PGMPROD.VSCOBLIB'
***
***        VS/COBOL COMPILER PROGRAM PRODUCT
***
***        IBM PP 5740-CO1  RELEASE 2.2 W/PTF 6
***
XXCOB     EXEC PGM=IKFCBL00,REGION=&CR,TIME=&GT
XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
XXSYSLIB   DD DSN=&&COPY,DISP=SHR
XX         DD DSN=&&COPY1,DISP=SHR
XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
XX           SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
XXSYSPRINT DD SYSOUT=A
XXSYSUDUMP DD SYSOUT=A
XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
XX           UNIT=VIOUS
XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
XX           UNIT=VIOUS
XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
XX           UNIT=VIOUS
XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
XX           UNIT=VIOUS
//COB.SYSIN  DD  *
XXLKED    EXEC PGM=IEWL,REGION=&LR,TIME=&LT,COND=(5,LT,COB),
XX           PARM=MAP
XXSYSLIB   DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
XX         DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
XX         DD DSN=&LINK,DISP=SHR
XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
XX         DD DDNAME=SYSIN
XXSYSLMOD  DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
XX           SPACE=(6144,(700,0,1))
XXSYSPRINT DD SYSOUT=A
XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIOUS,
XX           SPACE=(1024,(200,,1))
//LKED.SYSIN DD *
XXGO      EXEC PGM=*.LKED.SYSLMOD,REGION=&GR,TIME=&GT,
XX           COND=((9,LT,LKED),(5,LT,COB))
XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELE
XXSYSUDUMP DD SYSOUT=A
XXSYSOUT   DD SYSOUT=A
//GO.FT06F001 DD DSN=&&TEMP,UNIT=SYSDA,
//           DCB=(LRECL=50,BLKSIZE=50,RECFM=FB),
//           SPACE=(TRK,(1,1)),DISP=(NEW,DELETE)
//GO.OT  DD  SYSOUT=A
//GO.FT06F001  DD  SYSOUT=A
```

Attachment 4

ENTRY CCALLF

```
0001                    SUBROUTINE FORTS6
0002                    READ (05,1000) I,J,K,L
0003                    I=I+5
0004                    J=J+5
0005                    K=K+5
0006                    L=L+5
0007                    WRITE(6,100)
0008                    WRITE (06,1000) I,J,K,L
0009                    REWIND 08
0010                    WRITE (08,1000) I,J,K,L
0011                100 FORMAT (' FORTRAN ENTERED')
0012               1000 FORMAT(1X,3(I3,5X),I3)
0013                    RETURN
0014                    END
```

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.      CCALLF.
00003          AUTHOR.
00004          INSTALLATION. HQ.SAC/ADWATD
00005          DATE-COMPILED. FEB 20,1979.
00006          SECURITY. UNCLASSIFIED.
00007          REMARKS.  THIS COBOL PROGRAM INTENDS TO CALL A FORTRAN
00008         -          SUBROUTINE, USING A FILE CREATED BY THE MAIN AND
00009         -          USED BY BOTH.  THE COBOL PROGRAM ESTABLISHES
00010         -          INITIAL VALUES AND THE FORTRAN PROGRAM CHANGES
00011         -          THEM.  THERE ARE NO ARGUMENTS PASSED.
00012          ENVIRONMENT DIVISION.
00013          CONFIGURATION SECTION.
00014          OBJECT-COMPUTER.  IBM-370-3033.
00015          SOURCE-COMPUTER.  IBM-370-3033.
00016          INPUT-OUTPUT SECTION.
00017          FILE-CONTROL.
00018              SELECT TSTFYL ASSIGN TO UT-S-FT08F001.
00019              SELECT PRTOUT ASSIGN TO UT-S-OT.
00020          DATA DIVISION.
00021          FILE SECTION.
00022          FD  TSTFYL  LABEL RECORDS STANDARD.
00023          01  TEST-FILE                       PIC X(26).
00024          FD  PRTOUT LABEL RECORDS STANDARD.
00025          01  OUTPUT-LINE                     PIC X(132).
00026          WORKING-STORAGE SECTION.
00027          01  REKORD.
00028                  05 FILLER                   PIC X     VALUE SPACE.
00029                  05 PART-1                   PIC 9(3)          VALUE 010.
00030                  05 FILLER                   PIC X(5)  VALUE SPACE.
00031                  05 PART-2                   PIC 9(3)            VALUE 020.
00032                  05 FILLER                   PIC X(5)  VALUE SPACE.
00033                  05 PART-3                   PIC 9(3)            VALUE 030.
00034                  05 FILLER                   PIC X(5)  VALUE SPACE.
00035                  05 PART-4                   PIC 9(3)            VALUE 040.
00036          PROCEDURE DIVISION.
00037          100-OPEN-THE-FILES.
00038              OPEN OUTPUT PRTOUT, TSTFYL.
00039          200-GO-THE-WORK.
00040              MOVE 'THESE ARE ORIGINAL COBOL VALUES' TO OUTPUT-LINE.
00041              WRITE OUTPUT-LINE AFTER ADVANCING 1 LINES.
00042              MOVE REKORD TO TEST-FILE, OUTPUT-LINE.
00043              WRITE OUTPUT-LINE AFTER ADVANCING 1 LINES.
00044              WRITE TEST-FILE.
00045          300-CLOSE-THE-FILES.
00046              CLOSE PRTOUT, TSTFYL.
00047              CALL 'FORTSB'.
00048          400-REOPEN-THE-FILES.
00049              OPEN INPUT TSTFYL  OUTPUT PRTOUT.
00050          500-CONTINUE-THE-TEST.
00051              READ TSTFYL AT END GO TO 600-CLOSE-THE-FILES.
00052              WRITE OUTPUT-LINE FROM TEST-FILE AFTER 1.
00053          *IF THE VALUES ARE 015, 025, 035, AND 045 THE TEST WORKED.
00054          600-CLOSE-THE-FILES.
```

```
00055                CLOSE PRTOUT, TSTFYL.
00056            700-STOP-THE-TEST.
00057                STOP RUN.
```

THESE ARE ORIGINAL COBOL VALUES

020	020	030	040
15	25	35	45

ATTACHMENT 5

1. This attachment contains a sample listing of the Honeywell JCL necessary to compile and execute a Honeywell COBOL program and a FORTRAN subroutine. Also included is the source listing and the program execution.

2. Cards annotated with an A between the card number and $ are the activities of the job, card numbers 7, 8, and 9. Card 7 is the FORTRAN compile, card 8 is the COBOL compile, and card 9 is the execute. Because the COBOL program is the driver and is not loaded first (card 8), note the use of the $ ENTRY.

3. The source code for the driver and subroutine illustrates a file created in COBOL can be modified by a FORTRAN program and output by both, passing no arguments either way.

4. The last items are the execution's output. Note the COBOL program ouputs are output together. Under "COBOL Reentered", note also that the FORTRAN program has stripped off the leading zeroes.

```
03-13-79

$$    72619 ENTERED 2510P  AT 16.928 FROM SYSTEM=0 CC ROK  0-24-00


0001  $       SNUMC   72619
0002  $       IDENT
0003  $$      USERID
0004  $       COMPILE
0005  $       OPTION  FORTRAN
0006  $       ENTRY   C.COAL
0007  A$      FORTY
0008  A$      OVERL   ELSF,ROBERT
0009  A$      EXECUTE
0010  $       LIMITS  ,24K,=4K                                        N*T =
0011  $       FILE    A8,A15
0012  $       FILE    08,A15
0013  $       SYSOUT  01
0014  $       SYSOUT  06
0015  $       ENDJOB
  TOTAL CARD COUNT THIS JOB = 000102


* SUBEDJECT .ALL    I = 02 O= 06 N*I = 000480

* ACTY-01  SCARD #0007  FORTY     03/13/79    SW=210216000000
  * NORMAL TERMINATION      AT 005152 I=4160 SW=210216000000

      START 17.135        LINES      37       PROC  0.0001        I/O   0.000
      STOP  17.039        LIMIT   12000       LIMIT 0.0500      LIMIT
      SWAP   0.000
      LAPSE  0.001 FC D   TYPE       ELSY      IF/AT      FF/RT      IS/#C MS/#E

                     S* R   0191 *      187         0          0          1     1
                     F*     SYOUT
                     *1 R   0191 *        0         0          0         48    48
                     S* S   0191 *       49         0          2         36    36
                     K*     SYOUT
                     C*     SYOUT

      LIST     37 LINES


* ACTY-02  SCARD #0008  COBOL    03/13/79    SW=010200000000
  * NORMAL TERMINATION      AT 002477 I=4020 SW=010001000000

      START 17.000        LINES     142       PROC  0.0013        I/O   0.012
      STOP  17.074        LIMIT   20000       LIMIT 0.1500      LIMIT
      SWAP   0.012
      LAPSE  0.019 FC D   TYPE       ELSY      IF/AT      FF/RT      IS/#C MS/#E

                     D* S   0191 *       91         2          5         30    36
                     S* R   0191 *      257         0          0          4     4
                     C* R   0191 *       92         0          1          1     1
                     F*     SYOUT
                     *3 R   0191 *      823         0          0        180   180R
                     *2 R   0191 *       27         0          0         12    12
                     C* R   0191 *      432         0          9         45    45
                     K*     SYOUT
```

03-13-79

7261° 01 03-13-79 17.038

```
                1          SUBROUTINE FORTSB
                2          READ (08,1000) I,J,K,L
                3     1000 FORMAT(1X,3(I2,5X),I3)
                4          I=I+5
                5          J=J+5
                6          K=K+5
                7          L=L+5
                8          WRITE(6,100)
                9      100 FORMAT (" FORTRAN ENTERED")
               10          WRITE (06,1000) I,J,K,L
               11          REWIND 08
               12          WRITE (08,1000) I,J,K,L
               13          ENDFILE 08
               14          REWIND 08
               15          CALL FCLOSE (06)
               16          RETURN
               17          END
```

72619 02 PROGRAM-ID.: CCALLF COMPILED 79-03-13 17.07/15 6000 COBOL 51.

COBOL SOURCE LISTING [redacted]
ALT #

```
00001        IDENTIFICATION DIVISION.
00002        PROGRAM-ID.      CCALLF.
00003        AUTHOR.
00004        INSTALLATION. HQ.SAC/ADWATC
00005        DATE-COMPILED. 79-03-13
00006        SECURITY. UNCLASSIFIED.
00007        REMARKS.  THIS COBOL PROGRAM INTENDS TO CALL A FORTRAN
00008      -           SUBROUTINE, USING A FILE CREATED BY THE MAIN AND
00009      -           USED BY BOTH.  THE COBOL PROGRAM ESTABLISHES
00010      -           INITIAL VALUES AND THE FORTRAN PROGRAM CHANGES
00011      -           THEM.  THERE ARE NO ARGUMENTS PASSED.
00012        ENVIRONMENT DIVISION.
00013        CONFIGURATION SECTION.
00014        SOURCE-COMPUTER.  6000 WITH EIS.
00015        OBJECT-COMPUTER.  6000 WITH EIS.
00016        INPUT-OUTPUT SECTION.
00017        FILE CONTROL.
00018            SELECT PRTOUT ASSIGN TO CT FOR LISTING.
00019            SELECT TSTFYL ASSIGN TO A6.
00020        I-O-CONTROL.
00021            APPLY STANDARD ON PRTOUT.
00022            APPLY STANDARD ON TSTFYL.
00023        DATA DIVISION.
00024        FILE SECTION.
00025        FD  PRTOUT LABEL RECORDS STANDARD.

00026        01  OUTPUT-LINE                    PIC X(132).

00027        FD  TSTFYL  LABEL RECORDS STANDARD.

00028        01  TEST-FILE                      PIC X(36).

00029        WORKING-STORAGE SECTION.
00030        01  RECORD.

00031            03  FILLER                    PIC X     VALUE SPACE.
00032            03  PART-1                    PIC 9(3)          VALUE 01
00033            03  FILLER                    PIC X(5)  VALUE SPACE.
00034            03  PART-2                    PIC 9(3)          VALUE 02).
00035            03  FILLER                    PIC X(5)  VALUE SPACE.
00036            03  PART-3                    PIC 9(3)          VALUE 03
00037            03  FILLER                    PIC X(5)  VALUE SPACE.
00038            03  PART-4                    PIC 9(3)          VALUE 04
00039        PROCEDURE DIVISION.
00040        100-OPEN-THE-FILES.
00041            OPEN OUTPUT PRTOUT, TSTFYL.
00042        200-DO-THE-WORK.
00043            MOVE "THESE ARE ORIGINAL COBOL VALUES" TO OUTPUT-LINE.
00044  A-45      WRITE OUTPUT-LINE AFTER ADVANCING 1 LINES.
00045            MOVE RECORD TO TEST-FILE, OUTPUT-LINE.
```

72619 02 PROGRAM-ID. T COBOLF COMPILED 79-03-13 17.07.15 6000 COBOL 51

COBOL ACT # SOURCE LISTING

```
00046            WRITE OUTPUT-LINE AFTER ADVANCING 1 LINES.
00047            WRITE TEST-FILE.
00048
00049        300-CLOSE-THE-FILES.
00050            DISPLAY "CHECK 1".
00051            CLOSE PRTOUT, TSTFYL.
00052            CALL FORTSB.
00053            DISPLAY "CHECK 3".
00054        400-REOPEN-THE-FILES.
00055
00056            OPEN INPUT TSTFYL OUTPUT PRTOUT.
00057
00058            MOVE " COBOL REENTERED" TO OUTPUT-LINE.
00059            WRITE OUTPUT-LINE.
00060        500-CONTINUE-THE-TEST.
00061
00062            READ TSTFYL AT END GO TO 600-CLOSE-THE-FILES.

00063            DISPLAY "CHECK 2".
00064            MOVE TEST-FIL TO OUTPUT-LINE.
00065            WRITE OUTPUT-LINE AFTER ADVANCING 1 LINES.
00066        *IF THE VALUES ARE 015, 025, 035, AND 045 THE TEST WORKED.
00067        600-CLOSE-THE-FILES.

00068            CLOSE PRTOUT, TSTFYL.
00069        700-STOP-THE-TEST.
00070        STOP RUN.
```

*** THE ABOVE LISTING CONTAINS 000 ERROR MESSAGES *****

*** THE ABOVE LISTING CONTAINS 000 WARNING MESSAGES ***

* THE ABOVE LISTING CONTAINS 000 EFFICIENCY MESSAGES *

COMPILATION TIME (MIN): CLOCK= 000.93 PROC= 000.02

00010 OVERFLOW READS 00000 OVERFLOW WRITES 23796 WORDS PERCRY USED

03-13-79

SMLMB # 72019, ACTIVITY # = 33, , REPORT CODE = 63, RECORD COUNT = 00001

THESE ARE ORIGINAL CODOL VALUES

C10	C20	C30	C40
CODOL RELEVEL			
15	25	35	45

03-13-79

SWAPPED = 72519, ACTIVITY # = 33, , REPCNT CCCC = 06, RECORD COUNT = 00001

FORTRAN ENTERED
15 25 35 45

ATTACHMENT 6

1. This attachment contains a sample listing of the Honeywell JCL necessary to compile a Honeywell COBOL driver and FORTRAN subroutine. Also included is the source code listing and the program execution.

2. The activities (A) are the same as Attachment 5: Compile of the COBOL program, compile of the FORTRAN program, and the execution (cards 6, 7, and 8). The $ ENTRY is used with the COBOL PROGRAM-ID (CARAYF), but is not necessary because the COBOL program precedes the FORTRAN program. The COBOL program would be considered the driver without the $ ENTRY because it is the first program encountered.

3. The COBOL source code creates a nine element table (three rows by three columns) of character data. It is listed as output with the subscripts of each element. The table is passed to the FORTRAN routine where it is listed (without subscripts) row by column, and then column by row.

```
03-09-79

$$     68605 ENTERED 2SIGN AT 14.713 FROM SYSTEM-0 CC RCR  0-24-00


0001  $       SIGNON    68605
0002  $       ACCNT
0003  $$      USERID
0004  $       OPTION    FORTRAN
0005  $       ENTRY     CANAVF
0006  A$      JOBCOL    LISF,NOECK,LSTGO
0007  A$      FCRTY
0008  A$      EXECUTE   DUMP
0009  $       DATA      CI
0010  $       LIMITS    ,24K,-4K                                    M*T =
0011  $       SYSOUT    OT
0012  $       SYSOUT    06
0013  $       ENDJOB
  TOTAL CARD COUNT THIS JOB = 000111


* SCHEDULED 002   P= 02 J= 06 M*T = 000480

* ACTY-01  SCARC #0006  COBOL    03/09/79   SW=011200000000
  * NORMAL TERMINATION     AT 002477 I=4020 SW=011011000000

      START 15.214      LINES   1306      PROC  0.0019      I/O   0.003
      STOP  15.522      LIMIT  20000      TIMIT 0.1500      LIMIT
      SWAP   1.298
      LAPSE  0.308 FC D   TYPE      CLSY    IP/AT    FP/AT    IS/BC MS/BE

                   S* R   0191 *     255        0        0        4     4
                   D* R   0191 *      69        0        1        1     1
                   P*     SYSOUT
                   *3 R   0191 *    1416        0        0      100   100
                   *2 R   0191 *      61        0        1       12    12
                   G* R   0191 *    1030        0       14       48    48
                   K*     SYSOUT
                   C*     SYSOUT
                   *1 R   0191 *    1444        0        1       48    48
                   B* S   0191 *     154        0        4       24    24

      .LST   1306 LINES

* ACTY-02  SCARC #0007  FORTY    03/09/79   SW=210216000000
  * NORMAL TERMINATION     AT 005152 I=4000 SW=210216000000

      START 15.525      LINES     30      PROC  0.0001      I/O   0.000
      STOP  15.527      LIMIT  12000      TIMIT 0.0500      LIMIT
      SWAP   0.000
      LAPSE  0.001 FC D   TYPE      CLSY    IP/AT    FP/AT    IS/BC MS/BE

                   B* S   0191 *      23        4        0       24    24
                   S* R   0191 *      90        0        0        1     1
                   P*     SYSOUT
                   *1 R   0191 *       0        0        0       48    48
                   K*     SYSOUT
```

6686.5 01 PROGRAM-ID. CARRAYF COMPILED 79-03-09 15.51.45 6000 COBOL STC-68

COBOL
ALT 0

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.      CARRAYF.
00003          AUTHOR. LT WHITE.
00004          INSTALLATION. HQ.SAC/ADWATD
00005          DATE-COMPILED. 79-03-09
00006          SECURITY. UNCLASSIFIED.
00007          REMARKS.  THIS COBOL PROGRAM CREATES AN ARRAY (TABLE) AND
00008                    PASSES IT TO A FORTRAN SUBROUTINE  AND BACK.
00009
00010
00011          ENVIRONMENT DIVISION.
00012          CONFIGURATION SECTION.
00013          SOURCE-COMPUTER.  6000 WITH EIS.
00014          OBJECT-COMPUTER.  6000 WITH EIS.
00015          INPUT-OUTPUT SECTION.
00016          FILE CONTROL.
00017              SELECT PRTOUT ASSIGN TO CT FOR LISTING.
00018              SELECT CARDIN ASSIGN TO CI FOR CARDS.
00019          I-O-CONTROL.
00020              APPLY STANDARD ON PRTOUT.
00021              APPLY STANDARD ON CARDIN.
00022          DATA DIVISION.
00023          FILE SECTION.
00024          FD PRTOUT LABEL RECORDS STANDARD.

00025          01  IMAGE                          PIC X(132).


00026          FD  CARDIN LABEL RECORDS STANDARD.
00027          01  TEST-DATA.

00028              03 INFO                        PIC X(6).
00029              03 FILLER                      PIC X(74).
00030          WORKING-STORAGE SECTION.
00031          77  SUB-C                          PIC 9  VALUE 1.

00032          77  SUB-R                          PIC 9  VALUE 1.

00033
00034          01  COUNTER-DISPLAY.

00035              03 SUB-R-OUT                   PIC X9 VALUE 0.

00036              03 SUB-C-OUT                   PIC X9 VALUE 0.

00037          01  TEST-TABLE.
00038              03 ROWS  OCCURS 3  TIMES.
00039                 05 CALLUM  PIC X(6)  OCCURS 3 TIMES.

00040          PROCEDURE DIVISION.
00041  A-51    100-OPEN-THE-FILES.
```

```
6606 01 PROGRAM-ID.    CARAVF COMPILED 79-03-09 15.51.15 IS  6000 COBOL   STC-EE

COBOL                  S O U R C E   L I S T I N G
ALT #

00042                  OPEN INPUT CARDIN OUTPUT PRTOUT.
00043              200-INITIALIZE-THE-TABLE.
00044                  MOVE "COBOL ENTERED" TO IMAGE.
00045                  WRITE IMAGE AFTER ADVANCING TO TOP.
00046              250-READ-IN.

00047                  IF SUB-R > 3
00048                      GO TO 400-LIST-THE-TABLE.
00049                  READ CARDIN INTO CALLUP (SUB-R, SUB-C)
                                                                        *
                                                                        *
00050                      AT END GO TO 400-LIST-THE-TABLE.

00051                  IF SUB-C = 3
00052                      ADD 1 TO SUB-R
00053                      COMPUTE SUB-C = 1
00054                      GO TO 250-READ-IN
00055                  ELSE ADD 1 TO SUB-C
00056                      GO TO 250-READ-IN.
00057              400-LIST-THE-TABLE.

00058                  PERFORM 425-WRITE-THE-ROW VARYING SUB-R FROM 1 BY 1
00059                      UNTIL SUB-R > 3.
00060                  IF SUB-R > 3 GO TO 777-CALL-FORTRAN-SUBROUTINE.
00061              425-WRITE-THE-ROW.

00062                  PERFORM 450-WRITE-THE-COLUMNS VARYING SUB-C FROM 1 BY 1
00063                      UNTIL SUB-C > 3.
00064              450-WRITE-THE-COLUMNS.


00065                  MOVE SUB-R TO SUB-R-OUT.
00066                  MOVE SUB-C TO SUB-C-OUT.
00067                  WRITE IMAGE FROM CALLUP (SUB-R, SUB-C) AFTER
00068                      ADVANCING 2 LINES.
00069                  WRITE IMAGE FROM COUNTER-DISPLAY.
00070              777-CALL-FORTRAN-SUBROUTINE.

00071                  CLOSE PRTOUT CARDIN.
00072                  CALL FORTSU USING TEST-TABLE.
00073                  OPEN OUTPUT PRTOUT.
00074                  MOVE " COBOL REENTERED" TO IMAGE.
00075                  WRITE IMAGE.
00076              900-CLOSE-THE-FILES.
00077                  CLOSE PRTOUT.
00078              999-STOP.
00079              STOP RUN.
```

03-05-79 TH

6485 02 03-04-79 15.525

```
      1          SUBROUTINE FORTSB(IARRAY)
      2          DIMENSION IARRAY (3,3)
      3          WRITE(6,100)
      4          WRITE(6,12)((IARRAY(I,J),J=1,3),I=1,3)
      5          WRITE(6,12)((IARRAY(I,J),I=1,3),J=1,3)
      6       12 FORMAT(1H1,3(A6,1X),//,1X,3(A6,1X),//,1X,3(A6,1X))
      7      100 FORMAT (" FORTRAN ENTERED")
      8          CALL FCLOSE (06)
      9          RETURN
     10          END
```

```
COPYRIGHT EXTENDED

ALPHA
1 1

BRAVO
1 2

CHARLY
1 3

DELTA
2 1

ECHO
2 2

FOX
2 3

GULF
3 1

HOTEL
3 2

INDIA
3 3
LOBOL REENTERED
```

USER 9-7

SNUMB = 66005, ACTIVITY # = 03, , REPORT CODE = 06, RECORD COUNT = 000

FORTRAN ENTERED

03-09-79

ALPHA BRAVO CHARLY

DELTA ECHO FOX

GOLF HOTEL INDIA

PROGRAM ECHO HOTEL

CHARLY FOX INDIA

ATTACHMENT 7

1. This attachment contains a sample listing of the Honeywell JCL necessary to compile a Honeywell COBOL program and FORTRAN subroutine. Also included is the source listing and program execution.

2. The activities (A) are the COBOL compile, FORTRAN compile, and execution (cards 7, 8, and 9). The $ ENTRY uses the PROGRAM-ID (CPASSF) of the COBOL program (driver). The COBOL program is loaded first so the $ ENTRY is not required. However, if the order of the programs is changed, perhaps by a second programmer, the $ ENTRY prevents the subroutine from being considered to be the driver.

3. The source code of the driver and subroutine show the passing of various types of data back and forth between the two routines: alphabetic, special character, and numeric. The numeric portion (IC) is modified in FORTRAN and returned to the COBOL driver. The concept of CPASSF is to use FORTRAN to do the math, and let COBOL do the file/record manipulation.

4. The last item is the output of the COBOL program before and after the CALL to the FORTRAN subroutine.

```
$$  47625 ENTERED 2SIOP  AT 19.812 FROM SYSTEM-0 CO RDR  0-24-00

0001  $      SNUMB   47625
0002  $      IDENT
0003  $$     USERID
0004  $      OPTION  FORTRAN
0005  $      OPTION  COBOL
0006  $      ENTRY   CPASSF
0007  A$     COBOL   EISF,NDECK
0008  A$     FORTY
0009  A$     EXECUTE
0010  $      LIMITS  ,24K,-4K                                    M*T =
0011  $      SYSOUT  06
0012  $      SYSOUT  OT
0013  $      ENDJOB
  TOTAL CARD COUNT THIS JOB = 000087

* SCHEDULED .A02   P= 02 U= 06 M*T= 000480

* ACTY-01  $CARD #0007  COBOL    02/16/79   SW=010200000000
  * NORMAL TERMINATION     AT 002477 I=4020 SW=010010000000

      START 20.054      LINES   133      PROC  0.0008      I/O   0.002
      STOP  20.075      LIMIT 20000      LIMIT 0.1500      LIMIT
      SWAP   0.012
      LAPSE  0.021 FC D   TYPE     BUSY    IP/AT    FP/RT    IS/#C MS/#E

                   S* R   D191 *     190        0        0        4     4
                   D* R   D191 *      56        0        1        1     1
                   P*     SYOUT
                   *3 R   D191 *     583        0        0      180   180R
                   *2 R   D191 *      11        0        0       12    12
                   G* R   D191 *     471        0        7       48    48
                   K*     SYOUT
                   C*     SYOUT
                   *1 R   D191 *     724        0        0       48    48
                   B* S   D191 *     113        0        2       24    24

      LIST    133 LINES

* ACTY-02  $CARD #0008  FORTY    02/16/79   SW=210216000000
  * NORMAL TERMINATION     AT 005152 I=4060 SW=210216000000

      START 20.082      LINES    29      PROC  0.0001      I/O   0.000
      STOP  20.084      LIMIT 12000      LIMIT 0.0500      LIMIT
      SWAP   0.000
      LAPSE  0.002 FC D   TYPE     BUSY    IP/AT    FP/RT    IS/#C MS/#E

                   B* S   D191 *      75        2        4       24    24
                   S* R   D191 *      37        0        8        1     1
                   F*     SYOUT
                   *1 R   D191 *       0        0        0       48    48
                   K*     SYOUT
                   C*     SYOUT
```

47625 01 PROGRAM-ID.: CPASSF COMPILED 79-02-16 20.06HIS 6000 COBOL STD 9 W

COBOL ALT #

S O U R C E L I S T I N G

```
00001         IDENTIFICATION DIVISION.
00002            PROGRAM-ID.  CPASSF.
00003         AUTHOR.
00004         INSTALLATION. HQ-SAC-ADWATD.
00005         DATE-COMPILED. 79-02-16
00006         SECURITY. UNCLASSIFIED.
00007         REMARKS.    THIS PROGRAM PASSES DIFFERENT TYPES OF DATA TO A
00008        -    FORTRAN SUBROUTINE.  THERE IT IS MODIFIED AND PASSED BACK
00009        -    TO THE COBOL MAIN PROGRAM.
00010         ENVIRONMENT DIVISION.
00011         CONFIGURATION SECTION.
00012         SOURCE-COMPUTER.  6000 WITH EIS.
00013         OBJECT-COMPUTER.  6000 WITH EIS.
00014         INPUT-OUTPUT SECTION.
00015         FILE-CONTROL.
00016             SELECT PRTOUT ASSIGN TO OT FOR LISTING.
00017         I-O-CONTROL.
00018             APPLY STANDARD ON PRTOUT.
00019
00020         DATA DIVISION.
00021         FILE SECTION.
00022         FD  PRTOUT LABEL RECORD STANDARD.

00023         01   IMAGE           PIC X(132).


00024         WORKING-STORAGE SECTION.
00025         77  AA       PIC A(4)   VALUE " JIM".
00026         77  AB       PIC X      VALUE "-".
00027         77  IC       PIC 9(4)   VALUE 2594.
00028         01  TEST-INFO.

00029             03 FILLER          PIC X VALUE SPACE.
00030             03 NAME            PIC A(4).

00031             03 DASH            PIC X.

00032             03 PHONE           PIC 9(4).

00033         PROCEDURE DIVISION.
00034         100-OPEN-THE-FILES.
00035             OPEN OUTPUT PRTOUT.
00036         200-START.
00037             MOVE "INITIAL VALUES FROM COBOL WORKING STORAGE."
00038                TO IMAGE.
00039             WRITE IMAGE AFTER ADVANCING TO TOP.
00040             MOVE AA TO NAME.
00041             MOVE AB TO DASH.
00042             MOVE IC TO PHONE.
00043
00044         *ESTABLISH ALL VALUES PRESENT AT  START OF PROGRAM (COBOL).
```

```
47625 01 PROGRAM-ID.:   CPASSF COMPILED 79-02-16 20.06MIS  6000 COBOL   STD-68 V

CC                    S O U R C E   L I S T I N G
ALT #

00045
00046              WRITE IMAGE FROM TEST-INFO AFTER 2.
00047              DISPLAY "CHECK 4".
00048              CLOSE PRTOUT.
00049
00050         *CALL A FORTRAN SUBROUTINE AND MODIFY THE DATA.
00051
00052              CALL FORTSB USING AA, AB, IC.
00053              OPEN OUTPUT PRTOUT.
00054              MOVE "FOLLOW UP VALUES RETURNED FROM FORTRAN" TO IMAGE.
00055              WRITE IMAGE AFTER 2.
00056
00057         *SHOW DATA HAS BEEN CHANGED AND PROPERLY RETURNED TO COBOL.
00058
00059              MOVE AA TO NAME.
00060              MOVE AB TO DASH.
00061              MOVE IC TO PHONE.
00062              WRITE IMAGE FROM TEST-INFO AFTER 1.
00063          300-CLOSE.
00064              CLOSE PRTOUT.
00065          STOP RUN.


      ***** THE ABOVE LISTING CONTAINS 000 ERROR MESSAGES *****

        *** THE ABOVE LISTING CONTAINS 000 WARNING MESSAGES ***

          * THE ABOVE LISTING CONTAINS 000 EFFICIENCY MESSAGES *

COMPILATION TIME (MIN): ELAP CLOCK= 000.87       PROC= 000.01

 00000 OVERFLOW READS    00000 OVERFLOW WRITES   23456 WORDS MEMORY USED        0
```

```
      1          SUBROUTINE FORTSB(AA,AB,IC)
      2      C    INCREMENT THE INTEGER BY 1 AND RETURN IT TO COBOL.
      3          IC=IC+1
      4
      5          WRITE(6,100)
      6        100 FORMAT (" FORTRAN ENTERED")
      7
      8          RETURN
      9          END
```

SNUM3 = 71696, ACTIVITY # = 07, , REPORT CODE = 06, RECORD COUNT = 00001

FORTRAN ENTERED

02-16-79

INITIAL VALUES FROM COBOL WORKING STORAGE.

JIM-2594

FOLLOW UP VALUES RETURNED FROM FORTRAN
JIM-2595

ATTACHMENT 8

1. This attachment contains a sample listing of the Honeywell JCL necessary to compile a Honeywell COBOL driver and FORTRAN subroutine. Also included is the source code listing and program execution.

2. Attachment 8 is similar to Attachment 6 with two exceptions. The first is the need for the $ ENTRY, as the COBOL driver is loaded following the FORTRAN subroutine. The second is in the source code.

3. Like Attachment 6, here, the COBOL driver passes a table to the FORTRAN subroutine. The major difference in the source code is the record length. In Attachment 6, one additional character was read in; included in each record, making each record one full computer word long. In Attachment 8, it is assumed the record length cannot be increased, and therefore, must be dealt with in the FORTRAN subroutine as it stands: 5 characters per record. This test program illustrates how the use of the DECODE statement in FORTRAN helps to keep separate elements separate, although each element is less than a full word in length.

4. The last items are the printed output of the two routines. Compare them to those of Attachment 6.

```
03-22-79

$$     61610 ENTERED 2SICF  AT 16.567 FROM SYSTEM-C CD RCR  0-24-00


0001  $       SNUME   61610
0002  $       IDENT
0003  $$      USERID
0004  $       OPTION  FORTRAN
0005  $       ENTRY   CARAYF
0006  A$      FORTY
0007  A$      CCBCL   EISF,NCECK,LSTJL
0008  A$      EXECUTE DUMF
0009  $       DATA    CI
0010  $       LIMITS  ,244,-4K                                        M*
0011  $       SYSOUT  CT
0012  $       SYSOUT  06
0013  $       ENDJOE
  TOTAL CARD COUNT THIS JOB = 000114


* SCHEDULED .A02   P= 02 U= 06 M*T= 000480

* ACTY-01  $CARD #0006  FORTY     03/22/79   SW=210216000000
  * NORMAL TERMINATION      AT 005152 I=4060 SW=210216000000

     START 16.844       LINES     33       PROC  0.0001       I/O    0.000
     STOP  16.845       LIMIT  12000       LIMIT 0.0500       LIMIT
     SWAP   0.000
     LAPSE  0.001 FC C   TYPE      BUSY     IP/AT     FP/RT     IS/#C MS/#E

                  S* R   C191 *      60         0         0         1     1
                  P*     SYSLT
                  *1 R   C191 *       0         0         0        48    48
                  E* S   C191 *      86         0         2        36    36
                  K*     SYSLT
                  C*     SYSLT

      LIST     33 LINES


* ACTY-02  $CARD #0007  CCBCL     03/22/79   SW=011200000000
  * NORMAL TERMINATION      AT 002477 I=4020 SW=011010300000
    A-66
```

```
12  03-12-79  17.377

          1              SUBROUTINE FORTSB(JARRAY)
          2              DIMENSION IARRAY (3,3)
          3              CHARACTER TARRAY*5,JARRAY*45
          4              DECODE (JARRAY,110) ((IARRAY(I,J),J=1,3),I=1,3)
          5          110 FORMAT(9A5)
          6              WRITE(6,100)
          7              WRITE(6,12)((IARRAY(I,J),J=1,3),I=1,3)
          8              WRITE(6,12)((IARRAY(I,J),I=1,3),J=1,3)
          9           12 FORMAT(1H1,3(A5,1X),//,1X,3(A5,1X),//,1X,3(A5,1X))
         10          100 FORMAT (" FORTRAN ENTERED")
         11              CALL FCLOSE (06)
         12              RETURN
         13              END
```

```
COBOL                  S O U R C E   L I S T I N G
ALT #

00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.      CARAVE.
00003          AUTHOR.
00004          INSTALLATION. HQ.SAC/ADWATD
00005          DATE-COMPILED. 79-03-12
00006          SECURITY. UNCLASSIFIED.
00007          REMARKS.  THIS COBOL PROGRAM CREATES AN ARRAY (TABLE) AND
00008        -           PASSES IT TO A FORTRAN SUBROUTINE  AND BACK.
00009
00010
00011          ENVIRONMENT DIVISION.
00012          CONFIGURATION SECTION.
00013          SOURCE-COMPUTER.  6000 WITH EIS.
00014          OBJECT-COMPUTER.  6000 WITH EIS.
00015          INPUT-OUTPUT SECTION.
00016          FILE CONTROL.
00017              SELECT PRTOUT ASSIGN TO CT FOR LISTING.
00018              SELECT CARDIN ASSIGN TO CI FOR CARDS.
00019          I-O-CONTROL.
00020              APPLY STANDARD ON PRTOUT.
00021              APPLY STANDARD ON CARDIN.
00022          DATA DIVISION.
00023          FILE SECTION.
00024          FD PRTOUT LABEL RECORDS STANDARD.

00025          01  IMAGE                              PIC X(132).

00026          FD  CARDIN LABEL RECORDS STANDARD.
00027          01  TEST-DATA.

00028              03 INFO                            PIC X(5).
00029              03 FILLER                          PIC X(75).
00030          WORKING-STORAGE SECTION.
00031          77  SUB-C                              PIC 9  VALUE 1.

00032          77  SUB-R                              PIC 9  VALUE 1.

00033
00034          01  COUNTER-DISPLAY.

00035              03 SUB-R-OUT                       PIC X9 VALUE 0.

00036              03 SUB-C-OUT                       PIC X9 VALUE 0.

00037          01  TEST-TABLE.
00038              03 ROWS  OCCURS 3 TIMES.
00039                 05 CALLUP  PIC X(5)  OCCURS 3 TIMES.

00040          PROCEDURE DIVISION.
00041          100-OPEN-THE-FILES.
```

COBOL
0

SOURCE LISTING

```
00042           OPEN INPUT CARDIN OUTPUT PRTOUT.
00043       200-INITIALIZE-THE-TABLE.
00044           MOVE "COBOL ENTERED" TO IMAGE.
00045           WRITE IMAGE AFTER ADVANCING TO TOP.
00046       250-READ-IN.

00047           IF SUB-R > 3
00048                GO TO 400-LIST-THE-TABLE.
00049           READ CARDIN INTO CALLUM (SUB-R, SUB-C)



00050                AT END GO TO 400-LIST-THE-TABLE.

00051           IF SUB-C = 3
00052                ADD 1 TO SUB-R
00053                COMPUTE SUB-C = 1
00054                GO TO 250-READ-IN
00055              ELSE ADD 1 TO SUB-C
00056                GO TO 250-READ-IN.
00057       400-LIST-THE-TABLE.

00058           PERFORM 425-WRITE-THE-ROW VARYING SUB-R FROM 1 BY 1
00059              UNTIL SUB-R > 3.
00060           IF SUB-R > 3 GO TO 777-CALL-FORTRAN-SUBROUTINE.
00061       425-WRITE-THE-ROW.

00062           PERFORM 450-WRITE-THE-COLUMNS VARYING SUB-C FROM 1 BY 1
00063              UNTIL SUB-C > 3.
00064       450-WRITE-THE-COLUMNS.



00065           MOVE SUB-R TO SUB-R-OUT.
00066           MOVE SUB-C TO SUB-C-OUT.
00067           WRITE IMAGE FROM CALLUM (SUB-R, SUB-C) AFTER
00068              ADVANCING 2 LINES.
00069           WRITE IMAGE FROM COUNTER-DISPLAY.
00070       777-CALL-FORTRAN-SUBROUTINE.

00071           CLOSE PRTOUT CARDIN.
00072           CALL FORTSB USING TEST-TABLE.
00073           OPEN OUTPUT PRTOUT.
00074           MOVE " COBOL REENTERED" TO IMAGE.
00075           WRITE IMAGE.
00076       800-CLOSE-THE-FILES.
00077           CLOSE PRTOUT.
00078       999-STOP.
00079       STOP RUN.
```

03-22-79

CCBCL ENTERED

ALPHA
1 1

BRAVO
1 2

CHRLY
1 3

DELTA
2 1

ECHO
2 2

FOX
2 3

GOLF
3 1

HOTEL
3 2

INDIA
3 3

CCBCL REENTERED

03-22-79

SNUME = 81610, ACTIVITY # = 03, , REPORT CODE = 06, RECORD COUNT

FORTRAN ENTERED

ALPHA BRAVO CHRLY

DELTA ECHO FOX

GOLF HOTEL INDIA

03-22-79

ALPHA DELTA GOLF

BRAVO ECHO HOTEL

CHRLY FOX INDIA

ATTACHMENT 9

This attachment contains some computer listings for use with the program ARRAY1 in Section 8.

Figures 1 and 2 are the Honeywell versions of ARRAY1.

Figures 3 and 4 are the IBM versions of ARRAY1.

Figures 5 and 6 demonstrate the effect of COBOL synchronization.

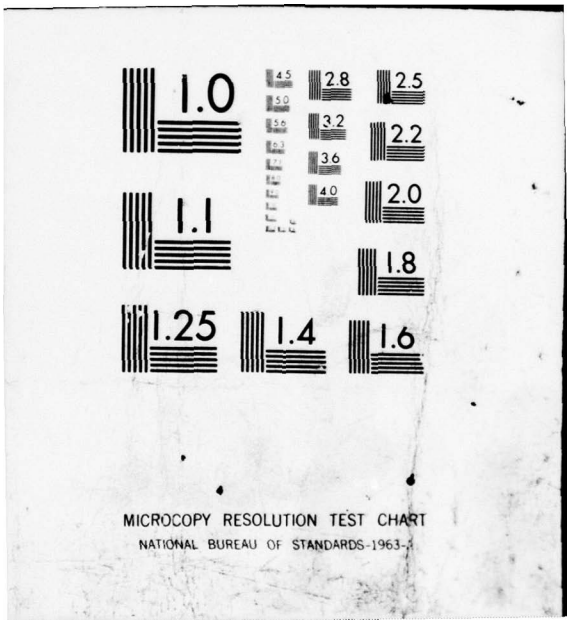
Figures 7 and 8 are the Honeywell and IBM Job Control Languages for these jobs.

AD-A070 959 STRATEGIC AIR COMMAND OFFUTT AFB NE F/G 9/2
THE INTERFACE OF COBOL AND FORTRAN ON THE HONEYWELL 6080 AND IB--ETC(U)
APR 79 D W LIND, R D GEER, J W WHITE

UNCLASSIFIED NL

3 OF 3
AD
A070 959

END
DATE FILMED
8 -79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
03-23-79

BU638 J1 PROGRAM-ID.:   ARRAY1 COMPILED 79-03-21 23.82HIS  6000 COBOL   STD-

COBOL
SLT #                    S O U R C E   L I S T I N G

00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.
00003          AUTHOR. DAVID W LIND.
00004          INSTALLATION. WWMCCS.
00005          DATE-WRITTEN. 79-02-12.
00006          DATE-COMPILED. 79-03-21
00007          SECURITY. UNCLASSIFIED.
00008          ******************************************************************
00009          ENVIRONMENT DIVISION.
00010          CONFIGURATION SECTION.
00011          OBJECT-COMPUTER. 6000 WITH ETS.
00012          ******************************************************************
00013          DATA DIVISION.
00014          WORKING-STORAGE SECTION.
00015          77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00016          77  FILLER PICTURE IS X(6) VALUE IS SPACES.
00017          77  SECOND-ITEM PICTURE IS X(8) VALUE IS "2ND ITEM".
00018          77  THIRD-ITEM PICTURE IS X99) VALUE IS "377 ".
00019          77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP-1 VALUE IS 7777.
00020          01  TABLE-ONE.
00021              02 ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARR[illegible]".
00022              02 ITEM-TWO PICTURE IS S9(3) COMP SYNC VALUE IS 123.
00023              02 ITEM-THREE PICTURE IS S9(7) COMP-1 SYNC VALUE IS 12345.
00024              02 ITEM-FOUR PICTURE S9(6)V9(4) COMP-2 SYNC VALUE IS 246[illegible].5.
00025              02 ITEM-FIVE PICTURE S9(4)V9(2) COMP-2 SYNC VALUE IS -123.[illegible]
00026          ******************************************************************
00027          PROCEDURE DIVISION.
00028          PARAGRAPH-1.
00029              CALL FSUB1 USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,

00030                                  FOURTH-ITEM, ITEM-ONE.
00031              STOP RUN.

00018
00019
00020      CONTAINS      42 CHARACTERS      7 WORDS
00021      STARTS IN CHARACTER POSITION       1
00022      STARTS IN CHARACTER POSITION       7
00023      STARTS IN CHARACTER POSITION      13
00024      STARTS IN CHARACTER POSITION      25
00025      STARTS IN CHARACTER POSITION      37
00026
00027
00028
00029
***** ER ARGUMENT IS NOT LEVEL 77 OR 01
       RESULTS MAY BE UNPREDICTABLE
00030
00031
```

FIGURE 1 (1 of 2)

```
THIS LIST IS UNCL

79   23.166

 1          SUBROUTINE SUBS1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
 2          DIMENSION ARRAY(7)
 3          DOUBLE PRECISION SITEM
 4          WRITE (06,10) FITEM, SITEM, TTTEM, ITEM4, ARRAY(1), ARRAY(2),
 5         *                ARRAY(3), ARRAY(5), ARRAY(7)
 6          WRITE (06,20)
 7       10 FORMAT (6X, A6, 6X, A8, 6X, A4, 6X, I4, //, 6X, A6, 6X, F8.2, 6X
 8         *I8, 6X, D12.4, 6X, E8.2)
 9       20 FORMAT (1H ,'END OF PROGRAM')
10          RETURN
11          END

03-22-79

 SNUMB = 30638, ACTIVITY # = 03, , REPORT CODE = 06, RECORD COUNT = 0000004

      77ITEM        2ND ITEM        #77        7777

      ARRAY          123.00          12345        0.24E5[ 04        -123.01
 END OF PROGRAM
```

FIGURE 1 (2 of 2)

```
COBOL
ALT #                    S O U R C E   L I S T I N G

00001        IDENTIFICATION DIVISION.
00002        PROGRAM-ID. ARRAY1.
00003        AUTHOR.
00004        INSTALLATION. WWMCCS.
00005        DATE-WRITTEN. 79-02-12.
00006        DATE-COMPILED. 79-02-15
00007        SECURITY. UNCLASSIFIED.
00008       ******************************************************************
00009        ENVIRONMENT DIVISION.
00010        CONFIGURATION SECTION.
00011        OBJECT-COMPUTER. 6000 WITH EIS.
00012       ******************************************************************
00013        DATA DIVISION.
00014        WORKING-STORAGE SECTION.
00015        77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00016        77  FILLER PICTURE IS X(6) VALUE IS SPACES.
00017        77  SECOND-ITEM PICTURE IS X(8) VALUE IS "2ND ITEM".
00018        77  THIRD-ITEM PICTURE IS X99X VALUE IS "#77 ".
00019        77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP-1 VALUE IS 7777.
00020        01  TABLE-ONE.
00021            02 ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARRAY ".
00022            02 ITEM-TWO PICTURE IS S9(3) COMP SYNC VALUE IS 123.
00023            02 ITEM-THREE PICTURE IS S9(7) COMP-1 SYNC VALUE IS 12345.
00024            02 ITEM-FOUR PICTURE S9(5)V9(4) COMP-2 SYNC VALUE IS 2468.5.
00025            02 ITEM-FIVE PICTURE S9(4)V9(2) COMP-2 SYNC VALUE IS -123.01.
00026       ******************************************************************
00027        PROCEDURE DIVISION.
00028        PARAGRAPH-1.
00029            CALL FSUB1 USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,
00030                             FOURTH-ITEM, TABLE-ONE.
00031            STOP RUN.
```

```
 1            SUBROUTINE FSUB1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
 2            DIMENSION ARRAY(7)
 3            DOUBLE PRECISION SITEM
 4            WRITE (06,10) FITEM, SITEM, TITEM, ITEM4, ARRAY(1), ARRAY(2),
 5           *              ARRAY(3), ARRAY(5), ARRAY(7)
 6            WRITE (06,20)
 7           10 FORMAT (6X, A6, 6X, A8, 6X, A4, 6X, I4, /, 6X, A6, 6X, F8.2, 6X
 8           *18, 6X, D12.4, 6X, F8.2)
 9           20 FORMAT (1H ,'END OF PROGRAM')
10            RETURN
11            END
```

```
      77ITEM      2ND ITEM      #77      7777

      ARRAY          123.00          12345      0.2469D 04      -12  1
 END OF PROGRAM
```

FIGURE 2

```
 1                          15.28.42          FEB 21,1979


00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID. ARRAY1.
00003          AUTHOR.
00004          INSTALLATION. IRICOMS.
00005          DATE-WRITTEN.
00006          DATE-COMPILED. FEB 21,1979.
00007          SECURITY. UNCLASSIFIED.
00008          ENVIRONMENT DIVISION.
00009          CONFIGURATION SECTION.
00010          OBJECT-COMPUTER. ILM-370-3033.
00011          DATA DIVISION.
00012          WORKING-STORAGE SECTION.
00013          77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00014          77  SECOND-ITEM PICTURE IS X(8) VALUE IS "2ND ITEM".
00015          77  THIRD-ITEM PICTURE IS X99X VALUE IS "$77 ".
00016          77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP VALUE IS 7777.
00017          01  TABLE-ONE.
00018              02  ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARRAY
00019              02  FILLER PICTURE XX.
00020              02  FILLER PICTURE 99 COMP VALUE ZERO.
00021              02  ITEM-TWO  PICTURE IS S9(3) COMP SYNC VALUE IS +123.
00022              02  ITEM-THREE PICTURE IS 9(5) COMP SYNC VALUE IS 12345.
00023              02  ITEM-FOUR COMP-2 SYNC VALUE IS 2.46E+3.
00024              02  ITEM-FIVE COMP-1 SYNC VALUE IS -1.23E+2.
00025          PROCEDURE DIVISION.
00026          PARAGRAPH-1.
00027              CALL "FSUB1" USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,
00028                                 FOURTH-ITEM, TABLE-ONE.
00029              STOP RUN.


RELEASE 2.0          FSUB1          DATE = 79052          15/28/52

          SUBROUTINE FSUB1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
          DIMENSION ARRAY(7)
          DOUBLE PRECISION FITEM,SITEM
          INTEGER*2 ITEM4
          WRITE (6,10) FITEM, SITEM, TITEM, ITEM4, ARRAY(1), ARRAY(2),
         *             ARRAY(3),ARRAY(4),ARRAY(5), ARRAY(7)
          WRITE (6,20)
       10 FORMAT (6X,6(6X,A6),6X,A4,6X,I4,//, 6A,A4,A2, 6X,I4,6X,
         *1P, 6X, D12.4, 6X, F5.2)
       20 FORMAT (1H , 'END OF PROGRAM')
          RETURN
          END


77ITEM    2ND ITEM    $77    7777

ARRAY     123    12345    0.2469E+04    -123.21
END OF PROGRAM
```

FIGURE 3.

```
PP 57+0-CB1 RELEASE 2.2 JULY 24, 1978                    IBM OS/VS COBOL

  1                          17.28.25        FEB 20,1979


00001           IDENTIFICATION DIVISION.
00002           PROGRAM-ID. ARRAY1.
00003           AUTHOR.
00004           INSTALLATION. TRICOMS.
00005           DATE-WRITTEN.
00006           DATE-COMPILED. FEB 20,1979.
00007           SECURITY. UNCLASSIFIED.
00008           ENVIRONMENT DIVISION.
00009           CONFIGURATION SECTION.
00010           OBJECT-COMPUTER. IBM-370-3033.
00011           DATA DIVISION.
00012           WORKING-STORAGE SECTION.
00013           77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00014           77  SECOND-ITEM PICTURE IS X(8) VALUE IS "2ND ITEM".
00015           77  THIRD-ITEM PICTURE IS X99X VALUE IS "#77 ".
00016           77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP VALUE IS 7777.
00017           01  TABLE-ONE.
00018               02  ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARRAY
00019               02  FILLER PICTURE XX.
00020               02  FILLER PICTURE 99 COMP VALUE ZERO.
00021               02  ITEM-TWO  PICTURE IS S9(3) COMP SYNC VALUE IS +123.
00022               02  ITEM-THREE PICTURE IS 9(5) COMP SYNC VALUE IS 12345.
00023               02  ITEM-FOUR COMP-2 SYNC VALUE IS 2.4635E+3.
00024               02  ITEM-FIVE COMP-1 SYNC VALUE IS -1.2301E+2.
00025           PROCEDURE DIVISION.
00026           PARAGRAPH-1.
00027               CALL "FSUB1" USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,
00028                                  FOURTH-ITEM, ITEM-ONE.
00029               STOP RUN.


RELEASE 2.0                FSUB1                DATE = 79051          17/28/31

      SUBROUTINE FSUB1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
      DIMENSION ARRAY(7)
      DOUBLE PRECISION FITEM,SITEM
      INTEGER*2 ITEM4
      WRITE (06,10) FITEM, SITEM, TITEM, ITEM4, ARRAY(1), ARRAY(2),
     *              ARRAY(3),ARRAY(4),ARRAY(5), ARRAY(7)
      WRITE (06,20)
   10 FORMAT (0X,A6,0X,A8,6X,A4,6X,I4,//, 6X,A4,A2, 6X,I4,6X,
     *I8, 6X, D12.4, 6X, F8.2)
   20 FORMAT (1H , 'END OF PROGRAM')
      RETURN
      END


 77ITEM      2ND ITEM      #77       7777

  ARRAY          123          12345          0.2469E+04          -123.01
END OF PROGRAM
```

FIGURE 4

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID. ARRAY1.
00003          AUTHOR.
00004          INSTALLATION. TRICOMS.
00005          DATE-WRITTEN.
00006          DATE-COMPILED. MAR 16,1979.
00007          SECURITY. UNCLASSIFIED.
00008          ENVIRONMENT DIVISION.
00009          CONFIGURATION SECTION.
00010          OBJECT-COMPUTER. IBM-370-3033.
00011          DATA DIVISION.
00012          WORKING-STORAGE SECTION.
00013          77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00014          77  SECOND-ITEM PICTURE IS X(8) VALUE IS "2ND ITEM".
00015          77  THIRD-ITEM PICTURE IS X99 VALUE IS "#77".   NO SYNCH
00016          77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP VALUE IS 7777.
00017          01  TABLE-ONE.
00018              02  ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARRAY "
00019              02  FILLER PICTURE XX.
00020              02  FILLER PICTURE 99 COMP VALUE ZERO.
00021              02  ITEM-TWO  PICTURE IS S9(3) COMP SYNC VALUE IS +123.
00022              02  ITEM-THREE PICTURE IS 9(5) COMP SYNC VALUE IS 12345.
00023              02  ITEM-FOUR COMP-2 SYNC VALUE IS 2.46853E+3.
00024              02  ITEM-FIVE COMP-1 SYNC VALUE IS -1.2331E+2.
00025          PROCEDURE DIVISION.
00026          PARAGRAPH-1.
00027              CALL "FSUB1" USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,
00028                                 FOURTH-ITEM, TABLE-ONE.
00029              STOP RUN.


              SUBROUTINE FSUB1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
              DOUBLE PRECISION FITEM,SITEM
              DIMENSION ARRAY(7),FITEM(3)
              INTEGER*2 ITEM4
              EQUIVALENCE (XARRAY , IARRAY)
              XARRAY = ARRAY(3)
              IARRAY = IARRAY + 1
              WRITE (06,10)FITEM(1),FITEM(2),FITEM(3),
             *SITEM,TITEM, ITEM4, ARRAY(1), ARRAY(2),
             *                   ARRAY(3),ARRAY(4),ARRAY(5), ARRAY(7)
              WRITE (06,30) IARRAY
              WRITE (06,20)
           10 FORMAT (6X,3A8,6X,A8,6X,A4,6X,I2,//, 6X,A4,A2, 6X,I2,6X,
             *I8, 6X, D12.4, 6X, E8.2)
           20 FORMAT (1H , 'END OF PROGRAM')
           30 FORMAT (6X,'IARRAY IS ', I5)
              RETURN
              END
```

FIGURE 5 (1 of 2)

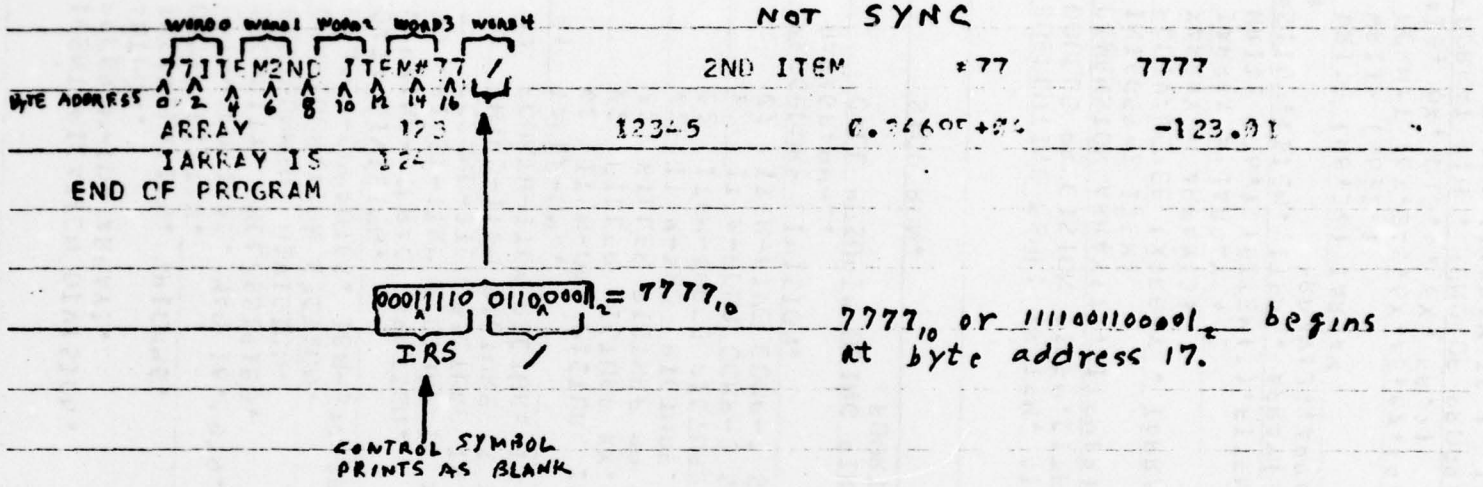


FIGURE 5 (2 of 2)

```
5740-C01 RELEASE 2.2 JULY 24, 1978                    IBM OS/VS COBOL

                                12.23.38        MAR 19,1979

00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID. ARRAY1.
00003          AUTHOR.
00004          INSTALLATION. TRICOMS.
00005          DATE-WRITTEN.
00006          DATE-COMPILED. MAR 19,1979.
00007          SECURITY. UNCLASSIFIED.
00008          ENVIRONMENT DIVISION.
00009          CONFIGURATION SECTION.
00010          OBJECT-COMPUTER. IBM-370-3033.
00011          DATA DIVISION.
00012          WORKING-STORAGE SECTION.
00013          77  FIRST-ITEM PICTURE IS X(6) USAGE IS DISPLAY VALUE "77ITEM".
00014          77  SECOND-ITEM PICTURE IS X(6) VALUE IS "2ND ITEM".
00015          77  THIRD-ITEM PICTURE IS X99 VALUE IS "*77".
00016          77  FOURTH-ITEM PICTURE IS 9(4) USAGE IS COMP SYNC VALUE IS 7777
00017          01  TABLE-ONE.
00018              02  ITEM-ONE PICTURE IS X(6) USAGE IS DISPLAY VALUE "ARRAY "
00019              02  FILLER PICTURE XX.
00020              02  FILLER PICTURE 99 COMP VALUE ZERO.
00021              02  ITEM-TWO  PICTURE IS S9(3) COMP SYNC VALUE IS +123.
00022              02  ITEM-THREE PICTURE IS 9(5) COMP SYNC VALUE IS 12345.
00023              02  ITEM-FOUR COMP-2 SYNC VALUE IS 2.4685E+3.
00024              02  ITEM-FIVE COMP-1 SYNC VALUE IS -1.2301E+2.
00025          PROCEDURE DIVISION.
00026          PARAGRAPH-1.
00027              CALL "FSUB1" USING FIRST-ITEM, SECOND-ITEM, THIRD-ITEM,
00028                                 FOURTH-ITEM, TABLE-ONE.
00029              STOP RUN.

RELEASE 2.2                    FSUB1                    DATE = 79078          12/.

          SUBROUTINE FSUB1 (FITEM, SITEM, TITEM, ITEM4, ARRAY)
          DOUBLE PRECISION FITEM,SITEM
          DIMENSION ARRAY(7),FITEM(3)
          INTEGER*2 ITEM4
          EQUIVALENCE (XARRAY , IARRAY)
          XARRAY = ARRAY(3)
          IARRAY = IARRAY + 1
          WRITE (06,10)FITEM(1),FITEM(2),FITEM(3),
         *SITEM,TITEM, ITEM4, ARRAY(1), ARRAY(2),
         *                   ARRAY(3),ARRAY(4),ARRAY(5), ARRAY(7)
          WRITE (06,30) IARRAY
          WRITE (06,20)
       10 FORMAT (6X,3A8,6X,A8,6X,A4,6X,I4,//, 6X,A4,A2, 6X,I4,6X,
         *I8, 6X, D12.4, 6X, F8.2)
       20 FORMAT (1H , 'END OF PROGRAM')
       30 FORMAT (6X,'IARRAY IS ', I5)
          RETURN
          END
```

FIGURE 6 (1 of 2)

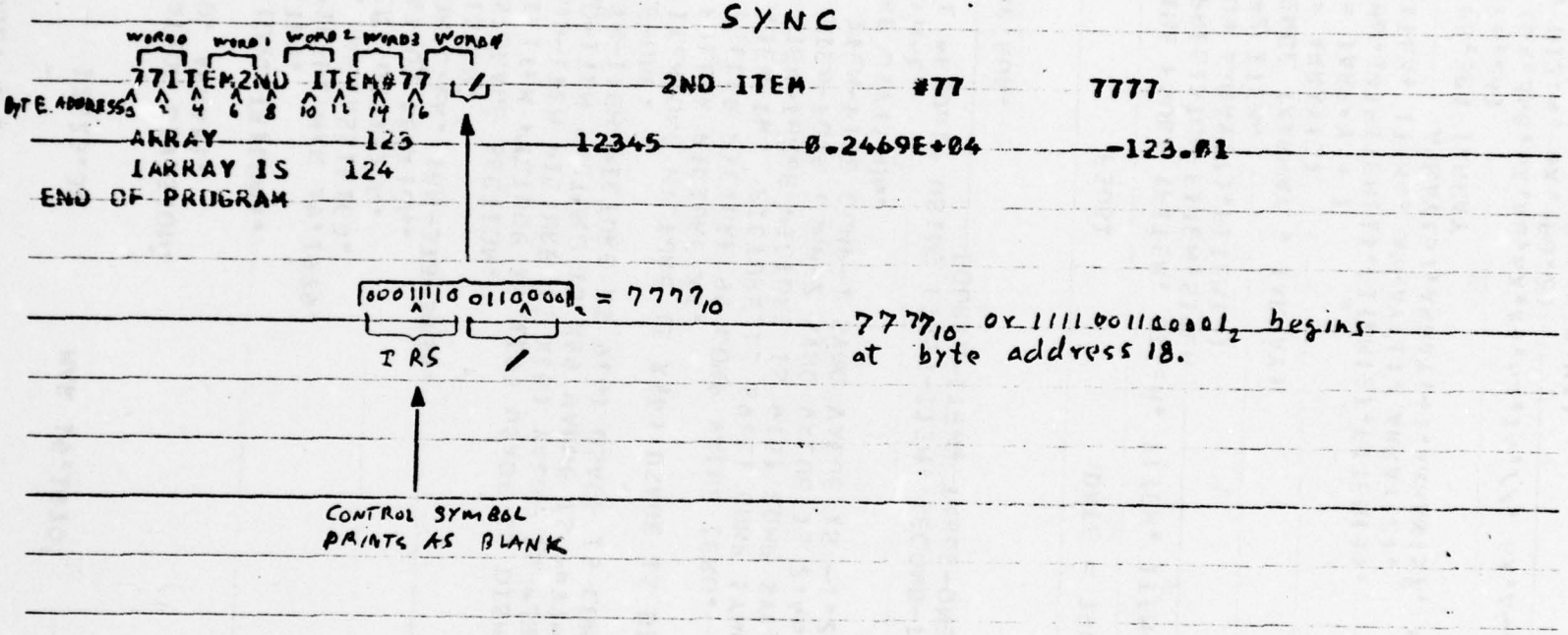


FIGURE 6 (2 of 2)

```
0001  $       SNUMB   46617
0002  $       IDENT
0003  $$      USERID
0004  $       OPTION  FORTRAN
0005  $       OPTION  COBOL
0006 A$       COBOL   E,SF,NDECK,LSTOU,ON6
0007 A$       FORTY   XREF,MAP,LSTOU
0008 A$       EXECUTE
0009  $       SYSOUT  06
0010  $       ENDJOB
  TOTAL CARD COUNT THIS JOB = 000052


* SCHEDULED .AD2    F= 02 U= 06 M*T= 000480

* ACTY-01  $CARD #0006  COBOL     02/15/79   SW=011300000000
  * NORMAL TERMINATION      AT 002477 I=4020 SW=011010000000

      START 17.904       LINES    317       PROC  0.0004        I/O   0.001
      STOP  17.906       LIMIT  20000       LIMIT 0.1500        LIMIT
      SWAP   0.000
      LAPSE  0.002 FC D   TYPE      BUSY     IP/AT     FP/RT     IS/#C MS/#E

                   S* R   D191 *     195         0         0         2     2
                   D* R   D191 *      91         0         1         1     1
                   P*     SYOUT
                   *3 R   D191 *     312         0         0       180   180
                   *2 R   D191 *      19         0         0        12    12
                   G* R   D191 *     160         0         5        48    48
                   K*     SYOUT
                   C*     SYOUT
                   *1 R   D191 *     366         0         0        48    48
                   8* S   D191 *       8         0         1        24    24

    LIST    317 LINES


* ACTY-02  $CARD #0007  FORTY     02/15/79   SW=311256000000
  * NORMAL TERMINATION      AT 005152 I=4060 SW=311256000000

      START 18.103       LINES    211       PROC  0.0001        I/O   0.000
      STOP  18.104       LIMIT  12000       LIMIT 0.0500        LIMIT
      SWAP   0.000
      LAPSE  0.001 FC D   TYPE      BUSY     IP/AT     FP/RT     IS/#C MS/#E

                   8* S   D191 *      26         1         3        24    24
                   S* R   D191 *     169         0         0         1     1
                   P*     SYOUT
                   *1 R   D191 *      83         0         0        48    48
                   K*     SYOUT
                   C*     SYOUT

    LIST    211 LINES
```

FIGURE 7

```
1
2     //COBOL    EXEC  VSCOBC,PARM.COB='QUOTE'
3     XXVSCOBC  PROC CT=0030,CR=192K,
      XX    COPY='SYS1.COPYLIB',COPY1='SYS1.COPYLIB'
      ***
      ***        VS/COBOL COMPILER PROGRAM PRODUCT
      ***
      ***        IBM PP 5748-CB1  RELEASE 2.2 W/PTF 6
      ***
4     XXCOB      EXEC PGM=IKFCBL00,REGION=&CR,TIME=&CT
5     XXSTEPLIB  DD DSN=SYS1.PGMPROD.VSCOBOL,DISP=SHR
6     XXSYSLIB   DD DSN=&COPY,DISP=SHR
7     XX         DD DSN=&COPY1,DISP=SHR
8     XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
      XX           SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
9     XXSYSPRINT DD SYSOUT=A
10    XXSYSUDUMP DD SYSOUT=A
11    XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(460,(1024,120)),
      XX           UNIT=VIODS
12    XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(460,(1024,120)),
      XX           UNIT=VIODS
13    XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(460,(1024,120)),
      XX           UNIT=VIODS
14    XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(460,(1024,120)),
      XX           UNIT=VIODS
15    //COB.SYSIN DD  *
16    //FORTRAN  EXEC  FTG1CLG,PARM.FORT='NOLIST'
17    XXFTG1CLG  PROC CT=0030,CR=156K,LT=0030,LR=156K,
      XX             GT=0030,GR=156K,
      XX  LINK='SYS1.PGMPROD.FORTLIB'
18    XXFORT     EXEC PGM=IG1FORT,REGION=&CR,TIME=&CT
19    XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
      XX           SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
20    XXSYSPRINT DD SYSOUT=A
21    XXSYSTERM  DD SYSOUT=A
22    XXSYSUDUMP DD SYSOUT=A
23    //FORT.SYSIN DD  *
24    //SYSIN     DD *                    GENERATED STATEMENT
25    XXLKED     EXEC PGM=IEWL,REGION=&LR,TIME=&LT,COND=(5,LT,FORT),
      XX             PARM=MAP
26    XXSYSLIB   DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
27    XX         DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
28    XX         DD DSN=&LINK,DISP=SHR
29    XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
30    XX         DD DDNAME=SYSIN
31    XXSYSLMOD  DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
      XX           SPACE=(6144,(76,8,1))
32    XXSYSPRINT DD SYSOUT=A
33    XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIODS,
      XX           SPACE=(1024,(200,11))
34    XXGO       EXEC PGM=*.LKED.SYSLMOD,REGION=&GR,TIME=&GT,
      XX             COND=((9,LT,LKED),(5,LT,FORT))
35    XXSTEPLIB  DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
36    XXFT05F001 DD DDNAME=SYSIN
37    XXFT06F001 DD SYSOUT=A
38    XXFT07F001 DD SYSOUT=B
39    XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELETE
      //
```

FIGURE 8

ATTACHMENT 10

This attachment includes the Honeywell and IBM JCL and the COBOL and FORTRAN programs discussed in paragraph 8.3.

```
//  EXEC  VSCOBC
XXVSCOBC   PROC CT=2000,CR=150K,
XX   COPY='SYS1.COPYLIB',COBVS='SYS1.COPYLIB'
***
***          VS/COBOL COMPILER PROGRAM PRODUCT
***
***          IBM PP 5?-?-CN1   RELEASE 2.3 W/PTF 4
***
XXCOB      EXEC PGM=IKFCBL??,REGION=6CR,TIME=6CT
XXSTEPLIB  DD DSN=&VS1.PGMPROD.VSCOBOL.DISP=SHR
XXSYSLIB   DD DSN=&COPY.DISP=SHR
XX         DD DSN=&COBVS1.DISP=SHR
XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
XXSYSPRINT DD SYSOUT=A
XXSYSUDUMP DD SYSOUT=A
XXSYSUT1   DD DSN=&&SYSUT1,SPACE=(400,(1024,120)),
XX            UNIT=VIO00
XXSYSUT2   DD DSN=&&SYSUT2,SPACE=(400,(1024,120)),
XX            UNIT=VIO00
XXSYSUT3   DD DSN=&&SYSUT3,SPACE=(400,(1024,120)),
XX            UNIT=VIO00
XXSYSUT4   DD DSN=&&SYSUT4,SPACE=(400,(1024,120)),
XX            UNIT=VIO00
//COB.SYSIN  DD *
//  EXEC  FTG1CLG,PARM.FORT='NOLIST'
XXFTG1CLG  PROC CT=2000,CR=150K,LT=0030,LR=150K,
XX             LT=2030,LR=150K,
XX  LINK='SYS1.PGMPROD.FORTLIB'
XXFORT     EXEC PGM=IGIFORT,REGION=6CR,TIME=6CT
XXSYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
XX            SPACE=(3200,(57,9),RLSE),DCB=BLKSIZE=3200
XXSYSPRINT DD SYSOUT=A
XXSYSTERM  DD SYSOUT=A
XXSYSUDUMP DD SYSOUT=A
//FORT.SYSIN  DD  *
XXLKED     EXEC PGM=IEWL,REGION=6LR,TIME=6LT,COND=(5,LT,FORT),
XX            PARM=MAP
XXSYSLIB   DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
XX         DD DSN=SYS1.PGMPROD.VSCOBLIB,DISP=SHR
XX         DD DSN=&LINK,DISP=SHR
XXSYSLIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
XX         DD DDNAME=SYSIN
XXSYSLMOD  DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
XX            SPACE=(6144,(76,8,1))
XXSYSPRINT DD SYSOUT=A
XXSYSUT1   DD DSN=&&SYSUT1,UNIT=VIO00,
XX            SPACE=(1024,(200,11))
XXGO       EXEC PGM=*.LKED.SYSLMOD,REGION=6GR,TIME=6GT,
XX            COND=((0,LT,LKED),(F,LT,FORT))
XXSTEPLIB  DD DSN=SYS1.PGMPROD.FORTLIB,DISP=SHR
XXFT??F001 DD DDNAME=SYSIN
XXFT??F002 DD SYSOUT=A
XXFT??F??  DD SYSOUT=B
XXGOSET    DD DSN=&&GOSET,UNIT=SYSDA,SPACE=(TRK,0),DISP=(MOD,DELETE
//
```

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.     CDEARAY.
00003          AUTHOR.
00004          INSTALLATION. HQ.SAC/ADWATD
00005          DATE-COMPILED. MAR 29,1979.
00006          SECURITY. UNCLASSIFIED.
00007          ENVIRONMENT DIVISION.
00008          CONFIGURATION SECTION.
00009          SOURCE-COMPUTER.  IBM-370-3033.
00010          OBJECT-COMPUTER.  IBM-370-3033.
00011          INPUT-OUTPUT SECTION.
00012          DATA DIVISION.
00013          WORKING-STORAGE SECTION.
00014          01  TEST-TABLE.
00015              03  ITEM-2 PIC S9(5) COMP SYNC VALUE +34567.
00016              03  ITEM-3 COMP-1 VALUE 1.65432E+3.
00017              03  ITEM-4 PIC X(21) VALUE 'ABCDEFGHIJKLMNOPQRSTU'.
00018              03  FILLER PIC XXX VALUE SPACES.
00019              03  FILLER PIC XXXX.
00020              03  ITEM-5 PIC S9(14) COMP SYNC VALUE -7777777777.
00021          PROCEDURE DIVISION.
00022              CALL 'FORTSB' USING TEST-TABLE.
00023          000-STOP.
00024              STOP RUN.
```

```
1                SUBROUTINE FORTSB (A)
2                DIMENSION A(11)
3                DOUBLE PRECISION DBPR, B1
4                EQUIVALENCE (I,X), (I10,Y), (I11,Z)
5                X=A(2)
6                Y=A(10)
7                Z=A(11)
           C  THE FOLLOWING CODE CONVERTS FROM DOUBLE INTEGER TO REAL*8.
8                B1 = I11
9                IF (I11.LT.0) B1=.4294967296D10 + I11
0                DBPR = I10*.4294967296D10 +B1
1                WRITE(6,10)I,A(2),A(3),A(4),A(5),A(6),A(7),A(8),A(10),A(11)
2                WRITE (26,20) DBPR
           C  THE FOLLOWING CODE CONVERTS FROM REAL*8 TO DOUBLE INTEGER.
3                IB = DBPR/.4294967296D10
4                IF (DBPR.LT.0..AND.IB.LE.0)IB=IB-1
5                B1 = DBPR - IB*.4294967296D10
6                IF (B1.GE. .2147483648D10) B1 = B1 - .4294967296D10
7                IF(B1.LE.-.2147483648D10)B1=B1+.4294967296D10
8                I10 = IB
9                I11 = B1
0                A(10) = Y
1                A(11) = Z
2                WRITE(6,10)I,A(2),A(3),A(4),A(5),A(6),A(7),A(8),A(10),A(11)
3                RETURN
4             10 FORMAT(' FORTRAN: ',I10,1X,F10.3,1X,6A4,1X,2I15)
5             20 FORMAT (6X,'DBPR = ',D18.10)
6                END
```

```
FORTRAN:      34567     1654.32Ø ABCDEFGHIJKLMNOPQRSTU    -2   812156815
     DBPR = -Ø.7777777777D+1Ø
FORTRAN:      34567     1654.32Ø ABCDEFGHIJKLMNOPQRSTU    -2   812156815
```

-27-79 23.600

```
      SUBROUTINE FORTS8 (A)
      DIMENSION A(2)
      DOUBLE PRECISION DBPR, B1
      EQUIVALENCE (I,X), (I10,Y), (I11,Z)
      Y=A(1)
      Z=A(2)
C  THE FOLLOWING CODE CONVERTS FROM DOUBLE INTEGER TO REAL*8.
      B1 = I11
      IF (I11.LT.0) B1=.68719476736D11 + I11
      DBPR = I10*.68719476736D11 + B1
      WRITE(6,10)A(1),A(2)
      WRITE (06,20) DBPR
C  THE FOLLOWING CODE CONVERTS FROM REAL*8 TO DOUBLE INTEGER.
      IB = DBPR/.68719476736D11
      IF (DBPR.LT.0..AND.IB.LE.0)IB=IB-1
      B1 = DBPR - IB*.68719476736D11
      IF (B1.GE. .34359738368D11)B1 = B1 - .68719476736D11
      IF(B1.LE.-.34359738368D11) B1 = B1+.68719476736D11
      I10 = IB
      I11 = B1
      A(1)=Y
      A(2)=Z
      WRITE(6,10)A(1),A(2)
      CALL FCLOSE (06)
      RETURN
   10 FORMAT(' FORTRAN: ',2I15)
   20 FORMAT (6X,'DBPR = ',D18.10)
      END
```

```
04-27-79

 SNUMB = 17632, ACTIVITY # = 03, , REPORT CODE = 06, RECORD COUNT = 000003

FORTRAN:                -2    -9058301041
     DBPR =  -0.77777777780 11
FORTRAN:                -2    -9058301041
```

```
04-27-79

$$    17532 ENTERED 2SIOP  AT 21.618 FROM SYSTEM-U CO RDR  6-24-00

0001  $       SNUMB   17632
0002  $       IDENT   FA101D0SC/30/77066X0XF   ,U/ADWATE/WHITE,
0003  $$      USERID  ADOPG43$/UZZ    FEB 79
0004  $       OPTION  COBOL
0005  $       OPTION  FORTRAN
0006  $       ENTRY   CARAVF
0007 A$       COBOL   EISF,NDECK,NLSTIN ← SUPPRESS COBOL SOURCE LISTING
0008 A$       FORTY
0009 A$       EXECUTE
0010  $       SYSOUT  06
0011  $       ENDJOB
  TOTAL CARD COUNT THIS JOB = 000059
```

ATTACHMENT 11

The listings in this attachment demonstrate the transfer of a double precision variable to FORTRAN through an array of single precision variables. The output shows the value of the double precision variable and the character equivalents of the variables through which it is passed.

```
03-22-79

81616 01  03-22-79   19.500

000000          1          DOUBLE PRECISION A
000000          2          A = 777777777.
000000          3          WRITE (06,20)
000002          4          DO 10 I=1,20
000010          5          A = A + I*100000010.
000013          6          CALL SUB(A)
000021          7       10 CONTINUE
000025          8       20 FORMAT (13X,'DBPR',15X,'X1',8X,'X2')
000031          9          STOP
000031         10          END


03-22-79

81616 01  03-22-79   19.501

000000          1          SUBROUTINE SUB(X)
000006          2          DIMENSION X(2),Y(2)
000006          3          DOUBLE PRECISION DBPR
000006          4          EQUIVALENCE (DBPR,Y(1))
000006          5          Y(1)=X(1)
000006          6          Y(2)=X(2)
000010          7          WRITE (06,10) DBPR,X(1),X(2)
000012          8       10 FORMAT (6X,D18.10,6X,A6,6X,A6)
000026          9          RETURN
000026         10          END

          DBPR                X1          X2
  0.87777778700 09       7DKLR'      HQ0000
  0.10777778070 10       7U0=NH      %00000
  0.13777778370 10       7V8\>(      U00000
  0.17777778770 10       7NPW':      000000
  0.22777779270 10       84K4        >00000
  0.28777779870 10       85G3)2      600000
  0.35777780570 10       8E-^A%      E00000
  0.43777781370 10       8D4,Z\      I00000
  0.52777782270 10       8D_N84      T00000
  0.62777783270 10       8EN#U;      G00000
  0.73777784370 10       8FX'

  0.85777785570 10       8G

  0.98777786870 10       8MOHGJ      0000
  0.11277778830 11       8N 6R
  0.12777778980 11       8NXT$<      ^0000
  0.14377779140 11       8O'\.
  0.16077779310 11       8P\9,K      V^0000
  0.17877779490 11       8U(IN4      7+0000
  0.19777779680 11       8UO1LX      X 0000
  0.21777779880 11       8V4^_8      R 0000
```

GLOSSARY

Algorithm	- A precise characterization of a method of solving a problem in a finite number of steps. (19)
Alignment	- Arrangement of data on a word boundary.
ANSI	- American National Standards Institute.
Array (Table)	- A collection of data items, often of the same data type.
Bit	- Acronym of BInary digiT.
Byte	- A group of binary digits treated or operated on as a unit.
COBOL	- Computer Language: COmmon Business Oriented Language.
COBOL Sentence	- A sequence of one of more statements specifying action to be taken, ending with a period and a space.
* COBOL Verb	- A word that expresses an action to be taken by a COBOL compiler or object program.
Code	- Instructions to the computer written in a computer language.
* Compile Time	- The time at which a source program is translated, by a compiler, to an object program.
Control	- The program which is currently executing.
Double-Word	- Two consecutive full-words.
Double-Word Boundary	- A double-word whose address is an even multiple of 8 (IBM) or 2 (Honeywell).
File	- A collection of data records. (19)
Fixed-Point	- The binary point (decimal) is assumed at the left end of the number; it cannot move. (19)
Floating-Point	- Decimal may "float" (move left or right) depending on value of exponent. (19)

	FORTRAN	- Computer language: FORmula TRANslation.
	Full-Word	- 4 consecutive bytes on the IBM, or 6 consecutive bytes on Honeywell.
	Full-Word Boundary	- A full-word whose address is an even multiple of 4 on IBM, or 1 on Honeywell.
	Function	- Allows a programmer to specify a variety of activities as a single operation, i.e., square root. (19)
	Half-Word	- 2 consecutive bytes.
	Half-Word Boundary	- A half-word whose address is an even multiple of 2 (IBM).
	Hardware	- Actual physical components of the computer.
	Interface	- Executed in conjunction with.
	JCL	- Job Control Language.
*	Language	- A set of sequences over a finite alphabet. (19)
	Linking	- Combining 2 or more programs into a form which may be executed as a single program.
	Loader	- That portion of the computer which transfers a program into main memory in a form suitable for execution. (14)
*	Mass Storage	- A storage medium on which data may be organized and maintained in both a sequential and non-sequential manner.
	Module	- Logically self-contained and discrete part of a larger program. (19)
	Padding	- Adding characters to a data item, usually to make it a full-word or end on some word boundary (see "Slack Bytes").
	Program	- A specification of the sequence of computational steps in a particular language. (19)
*	Random Access	- An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*	Record	- The most inclusive data item; in COBOL, a Ø1 level item.
*	Reserved Word	- A COBOL word specified in the list of words which may be used in COBOL source programs, but not as user defined words or names.
	Slack Bytes	- In COBOL - Bytes containing meaningless data inserted between data items to insure correct word alignment. (14)
	Slew	- Carriage control.
	Software	- Programs, programming, and programming languages. (19)
	Subroutine	- A logically separate part of a program which performs a specific task. (19)
*	Subscript	- An integer whose value identifies a particular element in a table.
*	Variable	- A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric item.
	Word	- Addressable segment of the machines memory. (19)
	Word Boundary	- A word whose address is a multiple of 1 (Honeywell) or 2 (IBM).

* ANSI Definitions

REFERENCES

1. AFP 13-2, Guide for Air Force Writing. Washington, D.C.: 1 November 1973.

2. Automatic Data Processing Glossary. Washington, D.C.: Bureau of the Budget, U.S. Government, 1974.

3. BN86, Macro Assembler Program. Waltham, Massachusetts: Honeywell Information Systems, Inc., March 1973.

4. BN9Ø, General Loader. Waltham, Massachusetts: Honeywell Information Systems Inc., July 1972.

5. Brooks, F.P. and Iverson, K.E., Automatic Data Processing. New York: John Wiley and Sons, Inc., 1969.

6. DD25, Revision Ø, COBOL Reference Manual. Waltham, Massachusetts: Honeywell Information Systems Inc., June 1975.

7. DD26, Revision Ø, COBOL User's Guide. Waltham, Massachusetts: Honeywell Information Systems Inc., July 1975.

8. DD31, Revision Ø, Control Cards Reference Manual. Waltham, Massachusetts, Honeywell Information Systems Inc., March 1974.

9. D81, Revision 2, IBM 360/370 or Univac Series 70 to Level 66 File Conversion. Waltham, Massachusetts: Honeywell Information Systems Inc., November 1974.

10. DDØ2A, Revision Ø, FORTRAN. Waltham, Massachusetts: Honeywell Information Systems Inc., January 1975.

11. GA22-7001-3, IBM System/370 System Summary. Poughkeepsie, New York: IBM System Products Dir, Product Publications.

12. GC26-3857-1, Systems: IBM VS COBOL for OS/VS. Palo Alto, Calif: IBM Corp, May 1978.

13. GC28-Ø692-3, Systems: OS/VS2 MVS JCL. New York: IBM Corp, October 1977.

14. GC28-6396-5, IBM OS Full American National Standard COBOL. Palo Alto, Calif: IBM Corp, June 1975.

15. GC28-6397-3, IBM System/360 Disk Operating System: FORTRAN IV Programmer's Guide. Palo Alto, Calif: IBM Corp, May 1975.

16. GC28-6399-3, IBM OS Full American National Standard COBOL Compiler and Library, Version 2 Programmer's Guide. New York: IBM Corp, November 1974.

17. GC28-6515-10, IBM System/360 and System/370 FORTRAN IV Language. New York: IBM Corp, May 1974.

18. Kalish, Daniel. Two Interfaces for Recursive Subprograms in FORTRAN. Massachusetts Institute of Technology, Cambridge, Massachusetts, 29 December 1972.

19. Ralston, A. and Meek, C.L., Encylopedia of Computer Science. New York: Petrocelli/Charter, 1976.

20. Report: FORTRAN Compiler Validation Summary Report. IBM G Compiler. Federal COBOL Compiler Testing Service, Dept of the Navy, Washington, D.C., 17 May 1977.

21. Report: FORTRAN Compiler Validation Summary Report. IBM H Compiler. Federal COBOL Compiler Testing Service, Dept of the Navy, Washington, D.C., 17 May 1977.

22. Series 60 Level 66 Summary Description. Waltham, Massachusetts: Honeywell Information Systems, 1975.

Your suggestions and comments on this report are welcomed. Mail them to

ADWA
Deputy Chief of Staff, Data Systems
Headquarters Strategic Air Command
Offutt AFB, Nebraska 68113

All comments and suggestions become the property of the United States Government which will retain the nonexclusive right to use and reproduce the information.

Comments or Suggestions: