

CONTROL DATA

JG

351 9016

**CONTROL DATA®
6000 COMPUTER SYSTEMS
JOVIAL COMPILER**

REFERENCE MANUAL

PREFACE

This publication describes and illustrates the use of the JOVIAL (J3) compiler language (Version 1.0) for the CONTROL DATA® 6000 series computers. The JOVIAL compiler operates under the control of the SCOPE 3.3 operating system.

The language is versatile in application and permits unusual flexibility in the structuring of data. A JOVIAL program is composed of declarations, which describe the structure of data and the organization of the program, and statements, which specify processing operations to be performed.

This manual is organized to show the JOVIAL characters and elements first, then the more complex formulas and the syntax of statements, and finally the syntax of declarations. The input/output statements and declarations are combined in a separate section. The permissible combinations of statements and declarations into complete programs follow, and the communications pool facility is described. The final sections describe the compilation and execution of a JOVIAL program.

It is assumed that the reader is an experienced programmer and has some basic knowledge of Control Data 6000 series computers. An understanding of data and processing declarations is assumed in the discussion of statements; an understanding of statements and declarations is assumed in the discussion of formulas.

The following publications may be consulted for further information.

<u>Title</u>	<u>Publication Number</u>
Control Data 6000 Series Computers SCOPE 3.3 Operating System Reference Manual	60305200
Control Data 6000 Series Computers FORTRAN Extended Reference Manual	60176600
Control Data 6000 Series Computer COMPASS Reference Manual	60190900

CONTENTS

1	INTRODUCTION	1-1	3	FORMULAS	3-1
	Language	1-1		Numeric Formulas	3-1
	Program Description	1-1		Sequence of Operation	3-2
	Program Structure	1-1		Mode of Results	3-3
	Notation	1-3		Literal Formulas	3-4
2	BASIC ELEMENTS	2-1		Boolean Formulas	3-4
	Characters	2-1		Relational Boolean Formulas	3-4
	Symbols	2-1		Numeric Relationals	3-5
	JOVIAL-Defined Symbols	2-2		Literal Relationals	3-6
	Arithmetic Operators	2-2		Entry Relationals	3-6
	Relational Operators	2-2		Status Relationals	3-6
	Logical Operators	2-3		Logical Boolean Formulas	3-7
	Sequential Operators	2-3		Shorthand Notation	3-8
	File Operators	2-3		Sequence of Evaluation	3-8
	Assignment Operator	2-4		Functions	3-9
	Functional Modifiers	2-4		Comments	3-10
	Separators	2-5		DEFINE Directives	3-10
	Brackets	2-5	4	STATEMENTS	4-1
	Declarators	2-6		Statement Forms	4-1
	Directives	2-6		Statement Labels	4-2
	Descriptors	2-6		Statement Label List	4-2
	User-Defined Symbols	2-7		Assignment Statements	4-2
	Data Type Properties	2-7		Numeric	4-3
	Constants	2-9		Literal	4-6
	Names	2-14			
	Variables	2-16			
	Scope of Symbol Definition	2-27			

Status	4-8	5 DATA DECLARATIONS	5-1
Boolean	4-9	Item Declarations	5-1
Entry	4-9	Integer	5-1
Exchange Statements	4-9	Fixed-Point	5-2
Numeric	4-10	Floating-Point	5-4
Literal	4-10	Literal	5-5
Status	4-11	Status	5-6
Boolean	4-11	Boolean	5-7
Entry	4-12	Implicit Item Declarations	5-7
Control Statements	4-12	Mode Declarations	5-8
GOTO	4-12	Arrays	5-10
Conditional (IF)	4-13	Array Declaration	5-11
IFEITH and ORIF	4-14	Constant List	5-12
Loop Statements	4-16	Referencing Arrays	5-16
FOR Clause	4-16	Tables	5-17
FOR Statement	4-17	Types of Tables	5-18
One-Component FOR Statement	4-18	Table Structure	5-18
Two-Component FOR Statement	4-19	Table Size	5-19
Three-Component FOR Statement	4-20	Table Packing	5-21
Parallel FOR Statements	4-21	No Packing	5-21
Nested FOR Statements	4-22	Dense Packing	5-22
FOR ALL Clause	4-23	Programmer Packing	5-24
Test Statement	4-24	Overlay Utilization	5-25
Program Control Statement	4-25	Table Declaration	5-25
STOP	4-25	Abbreviated Table Declarations	5-26
DIRECT-JOVIAL Statements	4-26	Table Header Declaration	5-26
DIRECT-JOVIAL Pseudo-Instructions	4-27	Ordinary Table Declarations	5-27
ASSIGN Pseudo-Instruction	4-27	Defined Table Declaration	5-30
		Like Table Declaration	5-34

Constant List	5-35	OPEN OUTPUT Statement	6-12
Referencing Tables	5-39	Transferring Data	6-12
Table Modifiers	5-40	INPUT Statement	6-12
ENTRY or ENT	5-40	OUTPUT Statement	6-14
NENT	5-40	Closing Files	6-14
NWDSSEN	5-40	SHUT INPUT Statement	6-14
Data Allocation	5-41	SHUT OUTPUT Statement	6-16
Overlay Declaration	5-41	Short Forms	6-16
6 INPUT/OUTPUT	6-1	Examples of I/O Data Forms	6-16
General	6-1	Object-Time Execution I/O	6-19
JOVIAL Files	6-1	Execution Control Card	6-19
Carriage Control	6-2	FORTTRAN-Formatted Output	6-19
File Input/Output	6-2.1	Formatted Output Examples	6-21
File Declaration	6-2.2		
SCOPE File Name	6-3	7 PROCESSING DECLARATIONS	7-1
Buffer Size	6-4	Switches	7-1
File Status	6-5	Index Switch Declaration	7-1
File Positioning	6-7	Index Switch Call	7-2
File-Positioning Statement	6-7	Item Switch Declaration	7-3
Organization of Data For Transfer	6-8	Item Switch Call	7-4
Data Forms	6-9	Closed Forms	7-6
Simple Items	6-10	CLOSE Routine	7-6
Arrays	6-10	Procedures	7-8
Table Items	6-10	Procedure Declaration	7-8
Parallel Items	6-10	Parameter Passing	7-13
Variable Items	6-10	Procedure Call	7-14
Table Entries	6-11	Functions	7-16
Tables	6-11	Function Call	7-16
Input/Output Statements	6-11	Exit from Closed Forms	7-17
Opening Files	6-12		
OPEN INPUT Statement	6-12	8 COMPOOL	8-1

COMPOOL Specification	8-1	10 JOVIAL CONTROL CARD	10-1
Data Declaration	8-1	Card Format	10-1
Common Declaration	8-2	Parameters	10-1
Subprogram Declaration	8-2	Source Input	10-1
COMPOOL Creation	8-3	Binary Output	10-2
COMPOOL Reference	8-4	COMPOOL	10-2
Data Reference	8-5	List	10-2
Subprogram Reference	8-5	Optimization	10-3
COMPOOL Examples	8-6	Monitor	10-3
Program Structure	8-6	Optional Prime	10-4
Main Program	8-6	COMPOOL Assembly	10-4
Subprogram	8-7	Terminate Compilation	10-4
9 DEBUGGING AIDS	9-1	Single Statement Scheduling	10-5
Monitor Statement (MONITOR)	9-1	Program Name	10-5
Run-Time Error Monitor Routine	9-8	Overlay Transfer Address	10-5

APPENDICES

A COMPILER ERROR MESSAGES	A-1	G LIBRARY SUBPROGRAMS	G-1
Source Diagnostic Messages	A-1	FORTRAN Library Functions	G-1
Termination Messages	A-13	FORTRAN Library Subroutines	G-4
B SAMPLE DECKS	B-1	JOVIAL Library Procedures	G-5
C COMPILER LIMITATION AND RESTRICTIONS	C-1	DECODE	G-5
D HINTS ON COMPILER USE	D-1	ENCODE	G-6
E CALLING SEQUENCES AND ERROR TRACING	E-1	HOLSTC	G-6
F CHARACTER SET	F-1	OVRLOD	G-7
		REMQUO	G-8
		SEGLOD	G-9
		STCHOL	G-9

	JOVIAL Library Function	G-10	L	DIRECT CODE ASSEMBLY LANGUAGE	L-1
	REM	G-10		Counters	L-1
H	SAMPLE LISTINGS	H-1		Origin	L-1
I	SAMPLE PROGRAMS	I-1		Location	L-1
	Program 1	I-2		Position	L-1
	Data Input	I-3		Source Statements	L-2
	Data Output to Printer	I-4		Comment	L-2
	Program 2	I-5		Instruction	L-3
	Data Input	I-6		Symbols	L-3
	Data Output to Printer and Punch	I-6		Registers	L-3
	Program 3	I-7		Address Expressions	L-4
	Data Input	I-8		Forcing Upper	L-4
	Program Output to Punch	I-8		Pseudo Instructions	L-4
				Storage Instruction	L-5
J	FIXED-POINT SCALING	J-1		BSS	L-5
	Addition and Subtraction	J-2		BSSZ	L-5
	Multiplication	J-2		JOVIAL Directives	L-5
	Division	J-3		DIRECT	L-5
	Exponentiation	J-3		JOVIAL	L-5
K	NUMERIC BIT PATTERNS	K-1		ASSIGN	L-6
	CHAR Examples	K-1		Central Processor Operation Codes	L-6
	MANT Examples	K-2			
	Numeric Exchange Statement Example	K-3	M	PROGRAM OVERLAYS AND SEGMENTS	M-1
	Floating-Point and Integer Item Exchange	K-3		Overlays	M-1
	Fixed-Point Simple Item Exchange	K-3		Creating JOVIAL Overlays	M-2
	Packed Table Fixed-Point Item Exchange	K-3		Loading Overlays	M-3
				Program Overlay Examples	M-3
				Segments	M-9

Creating JOVIAL Segments	M-10	N	JOVIAL/INTERCOM INTERFACE	N-1
Loading Segments	M-11			

FIGURES

1-1	Elements of the JOVIAL Language	1-2	B-1	Compile and Execute With INPUT File	B-1
5-1	Parallel Table Structure	5-20	B-2	Syntax Check Program Library Corrections	B-2
5-2	Serial Table Structure	5-20	B-3	Compile With Binary on SVE	B-2
9-1	No MONITOR Specification	9-5	B-4	Compilation to Produce Binary Deck	B-3
9-2	MONITOR to OUTPUT	9-6			
9-3	MONITOR to Separate File	9-7			

TABLES

2-1	Size Limits	2-8	A-2	Source Diagnostic Messages	A-2
2-2	JOVIAL Primitives	2-15	A-3	Termination Messages	A-14
2-3	Hollerith Data Movement	2-24	D-1	Extracting a Source Field	D-1
3-1	Operand Types and Results	3-4	F-1	Character Set	F-1
5-1	Item Description	5-9	F-2	Cross Reference Representations	F-2
6-1	Operand Forms for Input Statements	6-13	G-1	FORTRAN Library Functions	G-2
6-2	Operand Forms for Output Statements	6-15	G-2	FORTRAN Library Subroutines	G-4
6-3	Formatted Output Routines	6-20	L-1	Central Processor Operation Codes	L-7
A-1	Diagnostic Message Classes	A-1			

This manual describes the JOVIAL (J3) language for CONTROL DATA[®] 6000 series computers. It assumes a general knowledge of programming.

LANGUAGE

JOVIAL is a procedure-oriented language designed to make program writing easier. It uses self-explanatory English words and the familiar notations of algebra and logic. There are no card column restrictions on the format. Commands can be freely intermixed with the symbols of a program.

The JOVIAL language provides a consistent notation to designate and manipulate numeric values in both fixed- and floating-point representation, character values, status values, Boolean values, table values, and multidimensional values. Therefore, it is a language that can represent scientific and engineering problems involving numeric computations, business problems involving large data files, and logically complex problems involving symbolic data. Because the JOVIAL language allows the user to control storage allocation, it is particularly suitable for problems requiring an optimum balance between storage space and execution time. The organization of the JOVIAL language elements is shown in Figure 1-1.

PROGRAM DESCRIPTION

A JOVIAL program is composed of a set of declarations describing the data to be processed, and a set of statements describing the processing rules. These two sets of descriptions are independent to a large extent; changes in one set of descriptions do not necessarily entail changes in the other. Data descriptions may follow the statements which use them. However, a warning diagnostic is given at the place of declaration. The declarations and statements are formed from the elements of the language: names, constants, variables, and formulas. These elements are formed from the JOVIAL character set according to the rules of the language.

PROGRAM STRUCTURE

Two types of JOVIAL program compilations are possible: a main program or a subprogram. The main program consists of an executable program called by the operating system and

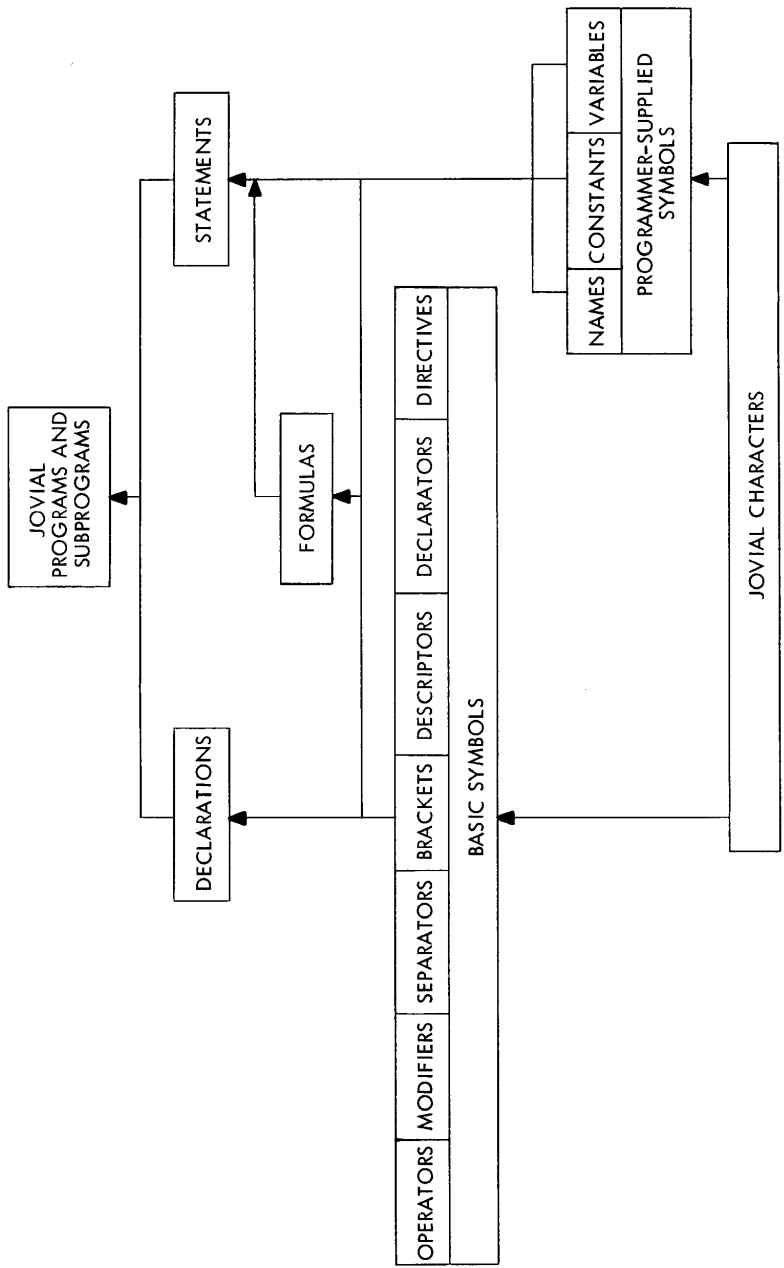


Figure 1-1. Elements of the JOVIAL Language

optional procedures. A subprogram is a procedure which is compiled alone and can only be called by a main program. Subprograms would be used for large systems, or where a single program contains many procedures that make it unwieldy. Program maintenance can be simplified by placing the procedures as subprograms. To use subprograms, a COMMUNICATIONS POOL (COMPOOL) must be used. The COMPOOL describes the subprogram procedures, their input and output parameters, and any data used by the subprograms and the main program. The COMPOOL is used by the compiler to obtain information on data and procedures not described in the program or subprogram being compiled. Thus, if a particular procedure must be changed and the data base and procedure parameters remain the same, only that subprogram containing the procedure needing to be modified has to be recompiled.

Every program or subprogram is composed of declarations and statements. Declarations describe the structure of data and organization of the program, and statements specify the processing operations to be performed.

NOTATION

In the discussion of the language that follows, a standard notation is used to describe the JOVIAL language forms. This notation is not part of the language; it indicates the order in which the elements appear and the options permitted the user.

- Upper-case words – JOVIAL words with a predefined meaning for the compiler are written in upper-case letters. These words must be included in the form in which they appear and must be spelled correctly.
- Lower-case words – Generic terms for classes of language elements are written with lower-case letters. The user supplies the specific elements according to the description of the generic term supplied in the accompanying text. A dash is used to separate a series of words.
- Brackets [] – Any word or phrase that can be included in or omitted from a JOVIAL form at the user's option is enclosed in brackets.
- Braces { } – Optional words or phrases are stacked one above the other and enclosed in braces when one of the stacked items must be chosen.
- Punctuation – Any punctuation included in the forms is part of the structure of the form, and must be included unless specifically noted otherwise.
- Spaces – Wherever one space is shown, the user can supply more than one space. Generally, spaces separate symbols but are not included in symbols.

Every JOVIAL program is composed of characters and symbols. There are two types of symbols: JOVIAL-defined and user-defined.

CHARACTERS

The JOVIAL character set is composed of:

- The 26 letters of the English alphabet
- Ten numerals, 0 through 9
- Twelve marks:) + * . \$ ' (- / , = blank

NOTE

On the 6000 series hardware, the prime (') is represented by the not-equal sign (≠).

SYMBOLS

Symbols are the words and punctuation marks of JOVIAL. They are composed of one or more characters which are usually set off by one or more blanks before and after each symbol. In some cases, blanks are not necessary, but they may always be used in order to see the symbols more easily. There are two types of symbols in statements and declarations: JOVIAL-defined (or basic) symbols having unchangeable meaning, and user-defined symbols where the user determines the meaning.

Basic symbols are operators, separators, brackets, functional modifiers, declarators, descriptors, and directives. Symbols are separated from one another by spaces; there are exceptions to this which will be indicated later, but spaces are never incorrect between symbols.

All separators and certain operators and brackets are ideograms composed of a set of JOVIAL marks. Other basic symbols are composed of two or more letters of the alphabet to form names. These names are called primitives and cannot be used as user-defined names. In addition, there is a set of descriptors composed of single-letter codes which are JOVIAL-defined and reserved for their own exclusive meaning.

JOVIAL-DEFINED SYMBOLS

JOVIAL-defined symbols may be grouped as follows:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Sequential Operators
- File Operators
- Assignment Operator
- Separators
- Brackets
- Functional Modifiers
- Declarators
- Directives
- Descriptors

ARITHMETIC OPERATORS

Arithmetic operators denote the five arithmetic operations. The symbols and their meanings are listed below.

+	Addition
-	Subtraction
*	Multiplication
/	Division
** or (* *)	Exponentiation

RELATIONAL OPERATORS

Relational operators denote comparative relationships between quantities and produce Boolean results. Relational operators are denoted by the following mnemonics.

EQ	Is equal to
GR	Is greater than
GQ	Is greater than or equal to
LQ	Is less than or equal to
LS	Is less than
NQ	Is not equal to

LOGICAL OPERATORS

Logical operators denote the three basic operations of Boolean algebra. The Boolean operators and their functions are:

AND	Conjunction
OR	Union
NOT	Negation

SEQUENTIAL OPERATORS

The sequential operators are listed below and explained immediately thereafter.

IF	If
IFEITH	If either
ORIF	Or if (else)
GOTO	Go to
FOR	Establish iteration loop
TEST	Test loop iteration counter
RETURN	Return from function or procedure
STOP	Stop execution, return to operating system

FILE OPERATORS

File operators manipulate data files. A description of the particular manipulation performed by each of these operators is given, as well as the mnemonics.

OPEN	Open a file
SHUT	Shut a file
INPUT	Input data
OUTPUT	Output data

ASSIGNMENT OPERATOR

The assignment operator is ASSIGN. It manipulates data elements by assigning a value in the midst of direct code.

FUNCTIONAL MODIFIERS

Functional modifiers express certain characteristics of program elements. The functional modifiers and their meanings are listed below.

ALL	All entries of a table
POS	File position
ENTRY or ENT	A single occurrence of all the items of a table
NENT	Number of entries within a table
NWDSEN	Number of words per entry of a table
BIT	Designates a particular bit or a group of contiguous bits
BYTE	Designates a single character or a group of contiguous characters
ODD	Low-order bit test
LOC	Location in address units
ABS	Absolute value
CHAR	Signed exponent of a floating variable
MANT	Signed mantissa of a floating variable

SEPARATORS

Separators form the punctuation of JOVIAL. The various separator symbols and functions performed by each are listed below.

.	Separates a statement from its name; used as a decimal point in numeric constants; also used to distinguish use of names in certain contexts.
,	Separates elements of lists, such as switch lists and parameter lists.
=	Assigns a value.
==	Exchanges values.
(blank)	Used as a character within a programmer-supplied name to effect readability.
...	Separates a pair of constants used to specify a range; also successive entries of a table in an I/O statement.
\$	Terminates a statement.

BRACKETS

Brackets are used to delimit groups of symbols. Listed below are the symbols and mnemonics that may be used as brackets. A brief description of each is also given.

()	Parenthesis
(/ /)	Absolute value
(\$ \$)	Subscript
(* *)	Exponent
' ' ' '	Comment
BEGIN END	Compound statement
START TERM	Program
DIRECT JOVIAL	Direct code
IFEITH END	Alternative statement

DECLARATORS

Declarators describe the characteristics of program elements such as tables, subroutines, and files. The declarators and the characteristics they describe are listed below.

ITEM	Declare a single data item
OVERLAY	Arrange data in storage to share the same beginning location
TABLE	Declare a group of items in tabular structure
STRING	Declare a string data item
ARRAY	Declare a data array
SWITCH	Declare a computed logical branch
FILE	Declare a file for I/O operations
PROC	Declare a procedure or function
CLOSE	Declare a CLOSE routine

DIRECTIVES

Directives are instructions to the compiler to perform certain functions. The MODE directive changes the default data declaration mode, and the DEFINE directive substitutes one set of symbols for another.

DESCRIPTORS

Descriptors are letters that denote certain characteristics in declarative contexts. Some descriptors are used in more than one context; for example, P is used in table declarations to indicate parallel structure, while in item declarations it denotes values to which data will be initialized. The descriptors and their uses are listed below.

<u>Preset</u>	<u>Use</u>
A	Fixed-point; operational register
B	Boolean; binary
D	Dense packing
F	Floating-point

<u>Preset</u>	<u>Use</u>
H	Hollerith code
I	Integer
L	Like table
M	Dense packing
N	No packing
P	Parallel; preset
R	Rigid; rounded
S	Serial; status; signed
T	Transmission code
U	Unsigned
V	Variable

USER-DEFINED SYMBOLS

User-defined symbols are those constructed by the programmer. These symbols fall into the general categories of constants, names, and variables, and are used to identify data. The data types have distinct properties which are described briefly before the rules for forming symbols are defined. Table 2-1 shows the maximum sizes allowed for items used to define symbols.

DATA TYPE PROPERTIES

Data type properties can be defined as the mathematical and the machine representation for the following data types:

- **Numeric Data**
 - Integer
 - Fixed-Point
 - Floating-Point
- **Literal Data**
- **Status Data**
- **Boolean Data**

TABLE 2-1. SIZE LIMITS

Item Type	Maximum Size
Numeric item	60 bits
Floating-point item used in multiplication, division, and exponentiation	47 bits
Mantissa (part of floating-point)	48 bits
Characteristic (part of floating-point)	11 bits
Literal item	1500 bits 250 bytes 25 words

NUMERIC DATA

Numeric data can be represented as integer, fixed-point, or floating-point data.

INTEGER

Integer data is always an exact integral value (represented in one's complement) that is positive, negative, or zero. Integers may have a precision not greater than 59 bits; signed integers may be composed of 60 bits including the sign bit.

Integer values may range from $-(2^{48}-1)$ through $+(2^{48}-1)$; addition and subtraction both accept operands which give results ranging from $-(2^{59}-1)$ through $+(2^{59}-1)$. However, loop variables, the results of arithmetic formulas used to compute loop variables in FOR statements, and address computations (or subscripts) are limited to the range $-(2^{17}-1)$ through $+(2^{17}-1)$.

FIXED-POINT

Fixed-point data always assumes the exact value times a power of 2 and is limited to a precision of 48 bits. In positive numbers, a scaling factor will indicate the number of fractional bits. In the case of negative numbers, the scaling factor will indicate the number of bits to be truncated.

FLOATING-POINT

Floating-point data is an approximation of a real number; this approximation may be a positive, negative, or zero number. The internal representation is a sign bit followed by an 11-bit exponent biased by 2000_8 and a 48-bit integer. Nonzero real values may range from (10^{**322}) to $(10^{**(-293)})$, and have a precision of approximately 15 decimal digits.

LITERAL DATA

Literal data is a string of bytes. Each byte is a graphic character with the machine representation of a binary bit pattern or code. The maximum length of a literal is 250 characters. Two types of character encoding are available:

- H for Hollerith code, which may be composed of any combination of display code characters.
- T for transmission code (STC)[†], which may be composed of any combination of the alphabetic, numeric, or JOVIAL marks.

STATUS DATA

Status data has the same qualities as unsigned integer data; it is used to specify the conditions that a status item can assume.

BOOLEAN DATA

Boolean data may be only 1 or 0 indicating a true or false, on or off, yes or no condition.

CONSTANTS

Constants are values that do not change during program execution. They include:

- Numeric Constants
 - Integer constants
 - Decimal
 - Octal
 - Functional modifiers

[†]The STC character set is a required feature for J3 JOVIAL compilers. The purpose is to enable easier transfer of programs from one computer to another. STC characters have the same bit representation on all computers. Note, that due to the use of 0g by the SCOPE operating system, STC blank is defined as 05g on the 6000 computer.

NENT (Rigid Table)
NWDSN
LOC

- Floating-point constant
- Fixed-point constant
- Literal Constant
- Status Constant
- Boolean Constant

NUMERIC CONSTANTS

INTEGER CONSTANTS

An integer number can be a positive, negative, or zero number and is always represented in binary in the machine. There are three forms of integers: decimal, octal, and functional modifiers.

Decimal Constants

Decimal constants represent numbers in the base 10 number system. The general form is an optional sign, a number, and an optional power of 10. The format is:

$[\pm] \text{number}_1 [\text{E number}_2]$

where

number_1
[number_2] = any combination of digits 0 through 9

The decimal constants are evaluated as number_1 multiplied by 10 raised to the power of number_2 , and have a maximum of 48 bits of precision.

Example:

$2\text{E}3 = 20\text{E}2 = 2000$

Octal Constants

Octal constants represent integers in the base 8 system. The format is:

O(number)

When octal constants are used in literal formulas, the number of numerals may not exceed 500.

Example:

O(20202) O(110011) O(12345670)

NENT Constant Functional Modifier

NENT is an integer constant functional modifier only when it is used with a rigid table; with variable tables, it is an integer variable. NENT denotes the number of entries in a rigid table. The size of the table is determined at compilation time, and is not dynamically variable during execution time. The size of the table is based on the value specified in the table declaration. As a part of the table, an integer item is automatically defined and assigned to the storage location immediately preceding the first data word of the table. The value of NENT is obtained as follows:

NENT (name)

where

name = table name or name of a variable in the table

NWDSSEN Constant Functional Modifier

NWDSSEN is a constant indicating the number of words in a table entry. The form is:

NWDSSEN (name)

where

name = a table or item within a table

LOC Constant Functional Modifier

The LOC functional modifier is an integer constant that is the starting address within the user's field length in central memory for the named operand.

LOC is dependent upon the position of the name within the JOVIAL object program and the position of the program within the routines loaded. It will be a constant only for a given load and execution. Recompiling the JOVIAL source program to produce a new object program, or reloading the existing object program may result in a different value for the constant.

The format is:

LOC $\left(\begin{array}{l} \text{Statement-name.} \\ \text{Table-name} \\ \text{Table-item name} \\ \text{Array-name} \\ \text{File-name} \end{array} \right)$

- A statement-name is always followed by a period.
- Table or array-name may not be subscripted because LOC always specifies the first data word in a table.

- Table-item name gives the location of the first word of the first occurrence of an item in a table.
- File-name gives the first word of the file environment table (FET), by which the executing program communicates with SCOPE I/O.

LOC is demonstrated below in a parallel form; i. e., each example gives the corresponding octal location in the compiler-generated storage map. Also listed are actual octal execution locations (as shown by the monitor output), which shows code location 0 of the program loaded in the user's field length at location 100, etc. These values are normally listed separately, but are combined here to show the actual values generated without going into detail.

Examples:

- START \$ ' DEMONSTRATION OF LOC OPERATOR ' ' *STORAGE MAP LOCATION*
ITEM FIX A 20 S 10 \$ 0444
ARRAY ARY 10 10 I 18 S \$ 0445
TABLE TAB V 50 \$ 0612
BEGIN
ITEM TOP F \$ 0612
ITEM BOTTOM F \$ 0674
END
FILE F1 H O R 128 V(OK) JWF \$ 0000
ITEM LOK I 18 S \$ MONITOR LOK \$ 0756
LAB1. 0774
GOTO LAB3 \$ 0777
LAB2. 0775
FIX = LOK \$ 0444
LAB3. 0777

The monitor output of the actual locations are:

- LOK = LOC (FIX) \$ 0544
LOK = LOC (ARY) \$ 0545
LOK = LOC (TAB) \$ 0712
LOK = LOC (TOP) \$ 0712
LOK = LOC (BOTTOM) \$ 0774
LOK = LOC (LOK) \$ 1056
LOK = LOC (LAB1.) \$ 1074
LOK = LOC (LAB2.) \$ 1075
LOK = LOC (LAB3.) \$ 1077
LOK = LOC (F1) \$ 0100
STOP \$
TERM \$

FLOATING-POINT CONSTANT

A floating-point constant can represent a wide range of real numbers, and consists of an integer and a characteristic. The floating constant is a decimal number optionally followed by the letter E and a number:

[±] decimal-number [E [±] number]

The decimal number is a combination of one or more numerals ranging from 0 through 9. The floating constant is evaluated as a decimal number which is multiplied by 10 raised to the power of the number after E, where an E is used.

Example:

$$3.14159 = 314159E-5 = .314159E1$$

FIXED-POINT CONSTANT

A fixed-point constant is a real number whose representation includes the letter A followed by a plus or minus, and a number indicating the bits to follow the decimal point. If the sign is a negative, the number following the sign indicates the least significant bits that will be truncated. Fixed-point constants are converted to a maximum of 48 bits of precision. The format is:

floating-constant A[±] number

where

number = any combination of the numerals 0 through 9

Examples:

<u>Fixed-point constants</u>	<u>Binary</u>
● 2.A4=2.24 A0	=10
● 4.0A-2=.6E1A-2	=100

LITERAL CONSTANTS

A literal constant can represent alphabetic, numeric, special characters, or any combinations of them. Each byte of the constant string is represented in the machine by the corresponding binary configuration. Two types of character coding are available:

- Display code
- Transmission code

They are specified by H for display code and T for transmission code. The format is:

number $\left(\begin{array}{c} H \\ T \end{array} \right)$ (string)

where

number = number of bytes in the string; must be greater than 0 and equal to or less than 250.

string = the list of characters to be encoded; it may be all blanks.

Examples:

- 5H(ERROR)
- 20T(COPY TAPE TO PRINTER)

Hollerith constants may contain the full display code character set. Transmission code constants contain only the JOVIAL character set (see Appendix F).

STATUS CONSTANTS

A status constant is any one of a set of values that can be assumed by a status variable. Status constants are always defined in sets and associated with a particular status item. The format is:

V (letter
name)

where

letter or name = any name or loop variable name used in the same program, but not a duplicate of a letter or name in the same status constant set.

The computer represents a status constant as a binary integer. The compiler increments the value of the status constant in a set beginning with zero.

Examples: In the status constant set, V(X) V(Y) V(Z),

- V(X) = 0
- V(Y) = 1
- V(Z) = 2

BOOLEAN CONSTANTS

A Boolean constant represents one of two values of Boolean algebra, 1 or 0,

where

1 = true, yes, on, positive

0 = false, no, off, negative

NAMES

Names are user-supplied identifiers of elements in a program such as tables, switches, and statements, and have no inherent meaning. Names must have the following qualities:

- Consist of two or more letters, numerals, or primes
- Begin with a letter
- Not end with a prime
- Not contain two consecutive primes
- Not be identical to any basic symbol

PRIMITIVES

Certain names which fulfill the requirements for symbols are reserved for use by the JOVIAL compiler. They may not be used as programmer-defined names. These are called primitives and are shown in Table 2-2.

TABLE 2-2. JOVIAL PRIMITIVES

ABS	ENTRY	LQ	OUTPUT
ALL	EQ	LS	OVERLAY
AND	FILE	MANT	POS
ARRAY	FOR	MODE	PROC
ASSIGN	GOTO	MONITOR	RETURN
BEGIN	GQ	NENT	SHUT
BIT	GR	NOT	START
BYTE	IF	NQ	STOP
CHAR	IFEITH	NWSEN	STRING
CLOSE	INPUT	ODD	SWITCH
DEFINE	ITEM	OPEN	TABLE
DIRECT	JOVIAL	OR	TERM
END	LOC	ORIF	TEST
ENT			

PRIME-PREFIXED PRIMITIVES

To provide a means of expanding the JOVIAL(J3) language to include new primitives without requiring changes to programs written under earlier versions of the compiler, all new primitives added to JOVIAL compilers must be prefixed by a prime. The 6000 series JOVIAL

compiler will accept all primitives with a prime; for example, 'ABS has the same effect as ABS. The P option on the JOVIAL control card will modify this so as not to accept the primitives LOC and MONITOR unless they are prefixed by a prime. Thus, if programs written for compilers not containing these two extensions as primitives use LOC and MONITOR as JOVIAL names, the programs can be compiled on the 6000 series computers by using the P option on the JOVIAL control card.

LOOP VARIABLE NAMES

Loop variable names are a special case of name symbols. A loop variable is the iteration counter of a FOR loop. A loop variable name is:

letter

Any letter can be used as a loop variable name. Even letters that are used in certain contexts as descriptors can be used. In JOVIAL, single letter symbols are recognized by context, not form.

VARIABLES

The basic element of data is the item, which can occur singly or as a member of a string, table, or array. A variable is an item that can change in value during program execution. Variables are classified in two groups:

- Named variables
- Functional modifier variables

Variables are also categorized according to type:

- Numeric
- Literal
- Boolean
- Status
- Entry

Definitions of the named and functional modifier variables are given first, followed by the descriptions of each variable category.

DEFINITIONS

NAMED VARIABLES

Named variables can occur singly, as multidimensional vectors known as arrays, or as members of a group in a single-dimensional structure known as a table.

Variables that occur singly are called simple variables and are identified by a name. The format of a simple variable is:

name

A loop variable is the iteration counter of a FOR loop. A loop variable is identified by a loop variable name consisting of a single letter succeeded by a FOR statement. Loop variables are integer-type variables defined only within a loop. The format is:

letter

Any letter can be used including letters that are also used in certain contexts as descriptors. Single letter symbols are recognized by context, not by form.

Variables that are members of a group are called indexed variables, and are identified by a name followed by an index. An index is a list of numeric formulas (one for each dimension) separated by commas, which specifies a particular occurrence of the variable. Each component of the index is called a subscript; hence, an indexed variable is also called a subscripted variable. The format of an indexed variable is:

name (\$index\$)

Example:

	<u>Simple Variables</u>	<u>Loop Variables</u>	<u>Indexed Variables</u>
•	XX	FOR X = 1\$	XX(\$1,3\$)
•	ITEMA	FOR C = 10,-2,0\$	ITEMA(\$B\$)

The following list summarizes the types of named variables in an outline form.

1. Numeric

Integer: simple, loop, indexed
Fixed: simple, indexed
Floating: simple, indexed

- 2. Literal
 - Simple
 - Indexed
- 3. Boolean
 - Simple
 - Indexed
- 4. Status
 - Simple
 - Indexed

FUNCTIONAL MODIFIER VARIABLES

A functional modifier variable is a JOVIAL primitive that is used to obtain special kinds of information about JOVIAL data structures, or to change the bit pattern of a portion of a data structure. The functional modifiers themselves can be variables, depending on the type of data to which they are applied. Variable functional modifiers may be numeric, Boolean, or entry. A functional modifier variable can be used any place where a named variable of the same type may be used.

The following list summarizes the types of functional modifier variables permitted in the JOVIAL program.

- 1. Numeric
 - POS
 - BIT
 - NENT
 - CHAR
 - MANT
- 2. Literal
 - BYTE
- 3. Boolean
 - ODD
- 4. Entry
 - ENTRY or ENT

USAGE

NUMERIC VARIABLES

A numeric variable is a variable which is represented in the same way as an integer constant, a floating-point constant, or a fixed-point constant. Loop variables are always numeric integer variables. Numeric variables can be named variables or functional modifier variables.

The functional modifiers that introduce numeric variables are always integer variables. They are POS, BIT, NENT, CHAR, MANT. These are detailed below.

POS Functional Modifier

The functional modifier POS is used to reference the logical record position of the specified file. The format is:

POS (file-name)

When POS is used to the right side of an expression, e. g. ,

HOLD = POS(FILEA)\$

the current logical record position of the file is placed in the variable HOLD. When POS is used to the left of an equal sign, it acts as an operator and the file is positioned to the location computed from the right of the equal sign.

A value of zero specifies that the file is to be rewound and positioned at the first record. That is, if the POS of a file were zero and the file were open for input, the first record would be available to be read; if the file were open for output, the first logical record on that file would be available for write operations.

Examples:

- | | |
|--|--|
| ● HOLD = POS(FILEA)\$ | The current logical record position is saved in the variable HOLD. |
| ● IF POS (FILEA) EQ 50\$
GOTO EOJ\$ | Transfers control to EOJ after 50 logical records are on file. |
| ● POS (FILEA) = 0 \$ | Rewinds the file. |
| ● POS (FILEA) = 30\$ | Positions the file to logical record 30. |

BIT Functional Modifier

The machine representation of a variable is a string of bits. BIT indicates a substring of a literal integer or fixed variable that is a contiguous set of bits. A BIT functional modifier is itself an unsigned integer variable, and is governed by the same rules as integers in assignment statements. The format of BIT is:

BIT (\$index\$) (named-variable)

where

index – indicates one or two numeric formulas, separated by commas. The first numeric formula in the index designates the first (or only) bit position of the named variable. The second, if present, indicates the number of bits in a substring that begins with the first bit position.

named-variable – can be defined as a fixed, integer, Boolean, status, or literal variable.

BIT is not valid for floating-point variables. Bits are numbered from zero starting at the left-most bit of the item. If the value of the second numeric formula in the index is zero, the value of the BIT functional modifier is zero.

No check is made during compilation or execution to determine if the numeric formula in the indexes specify bits within the named variable. If they are outside the range of the named variable, the BIT modification will be performed giving undefined results. Particular care should be taken when using variables to compute the indexes; if the formulas yield values which are outside of the field length, an out-of-range error (error mode = 1) will result during execution. If that occurs, the job will be terminated.

Examples:

Given a signed integer variable ALPHA defined as I 60 S:

- BIT(\$0,10\$(ALPHA) Specifies the first 10 bits including the sign.
- BIT(\$ 10\$(ALPHA) Specifies the eleventh bit.
- BIT(\$AA,19\$(ALPHA) Starting with the bit specified by numeric variable AA, the next 19 bits of ALPHA are specified. If AA is not an integer, it will be truncated to an integer. If AA has a value less than zero or greater than 40, the results will be undefined.

Given a fixed-point variable defined as A 37 S 10:

- BIT(\$1,26\$(FIX) Specifies the integer bits excluding the sign bit.
- BIT(\$26,10\$(FIX) Specifies the fractional bits of FIX.

Given a Hollerith variable defined as H 4:

- BIT(\$6,18\$(HOL) Specifies bits 6 through 24, which are the same portion as bytes 1 through 3.

NOTE

BYTE may not be used with numeric values.

NENT Functional Modifier

NENT is an integer variable when used with variable tables, and has its value set during execution of the program. This furnishes a standard way of maintaining a counter for the number of active entries in the table.

The value of NENT is obtained as follows:

NENT (name)

where

name = table-name or the name of a variable in the table

The user must specify a maximum number of active entries for each variable table in a table declaration in his program. The actual number of active entries is variable during program execution. Since the value of NENT is undefined (for variable tables) before it is set during program execution, the user must insure that it has been set before referencing it. The value should be a positive integer no greater than the maximum number of entries.

With the exception of I/O, setting the NENT of a variable table and its use are completely up to the user. For table output, the NENT of a variable table must be set prior to the output operation as entries 0 to NENT-1 will be output. On input, the NENT of a variable table will be set to the number of entries read in. (Input/output of tables are described in detail in Section 5.) In all other instances it is the user's responsibility to maintain the NENT at the value desired if he intends to make use of it for table operations, such as the FOR ALL operation.

Values which are not between 0 and NENT-1 are accepted by ENTRY operations but give undefined results. An attempt will be made to use the NENT of a variable table which is set to

a value that is out-of-range. If the formulas yield values which are outside of the field length, an out-of-range error will result during execution. If the above occurs, the job will be terminated.

Examples:

- NENT(TABLEA) = NENT (TABLEA) +1\$ An entry has been added to the table and NENT has been incremented.
- NENT(TABLEA) = NENT (TABLEA) -1\$ An entry has been removed from the table and NENT has been decremented.
- NENT(TABLEA) = 0\$ NENT is set to zero to initialize the counter.

CHAR Functional Modifier[†]

The CHAR functional modifier operates on a floating-point variable to produce a signed integer value, which is the characteristic of the floating variable. The format is:

CHAR (floating-point-variable-name)

If the floating-point item is in a table or an array, the item must be subscripted.

Examples:

- INT = CHAR(FLOAT)\$ The integer item INT is set to the value representing the power of 2 by which the fractional part of FLOAT is multiplied.
- CHAR(FLOAT1)=CHAR(FLOAT1) +1\$ The characteristic of FLOAT1 is increased by 1, which raises it by a power of 2. Thus, the value represented by the variable is doubled.

MANT Functional Modifier[†]

The MANT functional modifier operates on a floating-point variable to produce a fixed-point variable, which is the mantissa or fractional part of the floating value. The MANT results have the characteristics of a 48-bit signed integer and may be so treated. The format is:

MANT (floating-variable-name)

If the floating-point item is in a table or array, it must be subscripted appropriately.

CHAR and MANT are defined in such a way that the following is a true statement:

AA EQ MANT (AA) *2** CHAR(AA)

[†] See Appendix K for examples of numeric bit patterns.

Examples:

- INT = MANT(FLOAT)\$ The integer item INT is set to the value of the fractional part of the floating item FLOAT.
- MANT(FLOAT) = 1234567
890123 \$ The fractional part of FLOAT is set to the value of 1234567890123₁₀\$.

LITERAL VARIABLE

A literal variable is a variable whose representation is the same as a literal constant. The value is a binary bit pattern coded in either display code or transmission code to represent JOVIAL characters. (See Appendix F for display and transmission codes.) A literal variable is either a named variable or the functional modifier BYTE.

BYTE Functional Modifier

BYTE, operating on a literal variable, specifies a portion of this variable as another literal variable. It operates in a manner analogous to the BIT modifier. The representation of a one-character literal item is a string of six bits representing the single character. The bytes of an n-byte literal are numbered from left to right starting with zero and ending with n-1. BYTE obeys the move rules governing literal variables in assignment statements described on page 4-6. The format is:

BYTE(\$index\$(named-literal-variable))

where

index - is one or two numeric formulas separated by a comma.
 The first formula indicates the position of the first or only byte. The second formula, if present, indicates the number of bytes in a substring that begins with the first specified byte. If the second formula is 0, BYTE has the value of a blank of the type of the named-literal-variable. The specified byte position should be less than or equal to the number of bytes in the named item. The number of bytes specified should not exceed the number of bytes between the specified first byte position and the end of the item.

named-literal-variable - is the variable name.

Any byte modification for byte position outside the range of the literal variable will be performed, but large values can give undefined results and may cause termination of execution.

Examples:

- `BYTE($0, 2$)(AA) == BYTE ($6, 2$)(BB)$` The value of the first two bytes of AA is exchanged with the value of bytes 7 and 8 of BB.
- `BYTE(9)(ITEMA(A)) = 1H(Z)$` The tenth byte of ITEMA subscripted by A is set to a Hollerith Z. If ITEMA has less than 10 bytes, the result of this assignment is not predictable.

If the receiving field is shorter than the sending field, the bytes are right-justified and left-truncated in the receiving field. If the receiving field is longer than the sending field, the bytes remaining to the left are filled with the same type of blanks that are used in the sending field. BYTE moves within the same variable or within variables which are overlaid will be correctly moved.

Examples:

- `BYTE ($5, 7$)(HOL) = BYTE ($7, 9$)(HOL2)$` The nine bytes starting at byte 7 of HOL2 are moved to the seven bytes starting at byte 5 of HOL. Because the field receiving the byte modification is shorter, the nine bytes being moved are right-justified, truncating the left-most 2 bytes (see Table 2-3).

TABLE 2-3. HOLLERITH DATA MOVEMENT

Bytes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
HOL 2 (Before)	A	A	A	A	A	A	A	1	2	3	4	5	6	7	8	9	A	A
HOL	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D			
HOL 2	A	A	A	A	A	A	A	1	2	3	4	5	6	7	8	9	A	A
HOL (After)	D	D	D	D	D	3	4	5	6	7	8	9	D	D	D			

- `START $`
`ITEM HOL H 11 P 11H(DATA) $`
`MONITOR HOL $`
`FOR A = 10,-1,1 $`
`BEGIN`
`IF BYTE(0)(HOL) NQ 1H() $`
`GOTO XXX $`
`BYTE($0,A$) (HOL) = BYTE($1,A$) (HOL) $`
`BYTE(A) (HOL) = 1H() $`
`END`
`XXX.`
`TERM $`

The monitored Hollerith data for the above example is output in the form shown below. This occurs when the contents of the literal variable, HOL, is left-justified by a loop. When field one is moved to the left, byte A will remain the same. The next instruction sets byte A to blank. When the first byte is found to be non-blank or when the loop has iterated through all 11 bytes, the loop is terminated.

```

HOL A = 10 = DATAA
HOL      = DATA
HOL A = 9  = DATAA
HOL      = DATA
HOL A = 8  = DATAA
HOL      = DATA
HOL A = 7  = DATAA
HOL      = DATA
HOL A = 6  = DATAA
HOL      = DATA
HOL A = 5  = DATAA
HOL      = DATA
HOL A = 4  = DATAA
HOL      = DATA

```

BOOLEAN VARIABLE

A Boolean variable is a variable with a value having the same representation as a Boolean constant, either a 1 or 0. It is either a named variable or the functional modifier variable ODD.

ODD Functional Modifier Variable

The ODD modifier operates on a named numeric variable or a loop variable to produce a Boolean variable with the value 1 (true) if the named variable has odd value, or 0 (false) if the named variable is even.

For fixed-point variables, the ODD modifier is the least significant fractional bit, not the integer bit. The format of ODD is:

$$\text{ODD} \left(\begin{array}{l} \text{loop variable} \\ \text{named integer variable} \\ \text{named fixed-point variable} \end{array} \right)$$

The ODD modifier can be included in any Boolean formula according to the rules governing such formulas (see Section 3). The ODD functional modifier can be set by the user to 1 or 0, thus modifying the value of the variable.

Example:

- IF ODD (NUM) OR NOT ODD (COUNT) \$

where

ODD and NUM = integers.

This statement is true if NUM is odd or if COUNT is even.

ODD(NUM) = 0\$

The ODD of NUM and the least significant bit of NUM are both set to 0.

STATUS VARIABLE

A status variable is a named variable with the same configuration as a status constant. The value is expressed by a binary integer. Each possible integer value of the status variable must be one of the list of constants associated with the variable.

ENTRY VARIABLE

A table consists of one or more entries with each containing one or more items. The items can be mixtures of the various types. An ENTRY variable is a string of bits that are the representation of an entire entry in a table. The format is:

ENTRY ({ table-name } (\$index\$))
ENT ({ item-name })

where

ENTRY and ENT are interchangeable

index = a single numeric formula

The table to which ENTRY refers is identified by the table name or the name of any item in the table. The particular entry is designated by the index.

The value of the functional modifier depends on the structure of the table and the current value of the items in the specified entry. It cannot be classified as a particular data type such as integer or literal. Because one entry can contain several types of data items, it can have no properties of its own other than zero or nonzero; all bits are zero, or one or more bits are set.

ENTRY permits the manipulation of an entire entry in a table. Entries can be moved from one table to another regardless of structure. For instance, an entry can be moved from a serial table to a parallel table. The types of items in a table and their packing is not considered by the compiler; only the total entry size is considered. Two entries can be compared using the operators EQ or NQ only.

Examples:

- `ENTRY(DATA($ 5$)) = 0$` The sixth entry of table DATA is set to 0.
- `ENT(TAB($A)) == ENT
(TAB($A+1$))$` The entry in table TAB indicated by the subscript A is exchanged with the subsequent entry in the table.
- `IF ENT(LIST (0)) EQ 0 $` If the first entry in table LIST equals 0, the statement is true.
- `IF ENT(TAB(0)) EQ
ENT(TAB(1))$` The statement is true if all bits in the first two entries in TAB are equal to each other.

Table structure and entry structure are discussed in Section 5.

SCOPE OF SYMBOL DEFINITION

User-defined symbols have two levels of scope of definition. They are:

- Local scope
- Global scope

A symbol has global scope when declared explicitly in a COMPOOL declarative statement during program compilation or implicitly by mode definition outside a procedure or function. Global scope symbols and their properties can be used anywhere in the program.

A symbol declared within a procedure or function is defined as having local scope provided the symbol is valid only within the specific procedure or function in which it is declared.

The areas or contexts in which user-defined symbols may be used are divided into four categories as shown below:

- Statements, switches, CLOSEs, programs, and subprograms
- Items, arrays, tables, files, procedures, and functions
- Status constants (these cannot be duplicated in the same status constant list)
- Device names used in file declarations

Each symbol must be unique within each category, however, any symbol in one category may be duplicated in any or all of the other three categories. The following conditions are applicable:

- The declaration of a local scope symbol within a procedure or function takes precedence over a declaration of the same type in global scope. However, if a

symbol is not declared within a procedure and is declared in a global definition, the global definition is used. A procedure or function can use any locally defined definition or any global definition that has not been over-ridden by a local definition of the same symbol.

- A CLOSE does not involve a change in scope of definition; therefore, symbols declared inside a CLOSE have the same scope of definition as the CLOSE.
- A loop variable is a special class of symbols and may be duplicated provided that two symbols of the same initial letter are not active within the scope of definition at the same time. That is, the loop variable cannot be activated while it is still active within a main program or procedure. However, it can be active in a main program and in a procedure or function called from within the region of activity of the main program loop variable.
- The same symbol can be declared in a main program or subprogram and in a COMPOOL used in compiling the program or subprogram. In the event of multiple definitions, the compiler recognizes the definitions in the following order:
 1. Declaration in the main program, but if there are conflicts between main program declarations and procedure or function declarations, those of the latter group have precedence.
 2. COMPOOL specified during compilation of main program or subprograms.
 3. Mode declarations.
- Variables which are used before being declared will use the variable declaration, not that of the mode which was active at the time the variable was first used. A warning diagnostic will be issued following the data declaration. Mode resolution is performed after all statements have been examined for variable declarations (page 5-9).

Examples:

The following example demonstrates the use of items, arrays, tables, file procedures, and functions using local and global scope of definition.

<pre> START \$ 'SCOPE OF DEFINITION' FILE OUT H O V 128 V(OK) HOL \$ ITEM HOL H 40 \$ PROC IN (VALA, VALB) \$ BEGIN ITEM IN F \$. . END </pre>	<p>OUT is declared with HOL as a device name.</p> <p>Item HOL has global scope.</p> <p>The function IN is declared within the main scope of the program (global scope). It may be called from anywhere within the program.</p>
---	--


```

PROC PROCA (VBL = VBL) $
BEGIN
ARRAY HOL 20 10 F $
.
.
.
HOL ($5, 7$) = 40.37 $
END

```

The declaration of HOL as a floating-point array has local scope and is known only within procedure PROCA. Since it is declared within the procedure, the main program is never searched to determine if HOL is defined there.

```

PROC PROCB (VBL, VBL = VBL) $
BEGIN
HOL = 3H(END) $
.
.
.
END

```

HOL is not declared within PROCB, therefore, the global definition is used. Any references to HOL within PROCB will refer to the 40-byte Hollerith variable declared in the main program.

```

PROC PROCC (=VBL, VBL) $
BEGIN
ITEM HOLD F $
.
.
.
HOLD = IN (50, 100) $
.
.
.
END

```

No declaration for IN exists within the local scope of procedure PROCC, therefore, a floating-point function which exists in the global scope is used.

```

PROC PROCD (IN) $
BEGIN
ITEM IN I 18 S $
ITEM PROD I 18 S $
PROD = IN * 37 $
IF PROD LS 1000 $
RETURN
PROD = 1000 $
END

```

The local definition of IN as an integer item takes precedence over the global definition of IN as a floating-point function.

```

PROC PROCE (HOL) $
BEGIN
ITEM HOL H 120 $
.
.
.
OUTPUT OUT HOL $
END
TERM $

```

The local definition of HOL as a 120-byte literal takes precedence over the global definition of HOL as a 40-byte literal. File OUT is written on the SCOPE file HOL.

A formula is a combination of variables, constants, operators, and brackets producing a single value on evaluation. Variables and constants are formulas and can be combined with operators and brackets to make other formulas.

The types of formulas are the same types as those of constants and variables: numeric, literal, Boolean, status, and entry.

NUMERIC FORMULAS

Numeric formulas are composed of numeric constants, numeric functional modifiers, and numeric variables combined with arithmetic operators to form a numeric relationship that expresses a single value. The constants and variables within a numeric formula are called operands. Parentheses are used to group symbols and to control the sequence of operations.

Arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	} exponentiation
(* *)	
ABS ()	} absolute value
(/ /)	

The bracketed exponentiation (* *) is interchangeable with the operator **, so the formulas below are equivalent:

AA**BB or AA(*BB*)

In this example, AA is raised to the power BB.

Similarly, the functional modifier ABS() or the brackets (/ /) both specify absolute value for the enclosed numeric formula:

ABS(A - B) or (/A - B/)

In each example, the result of A-B is absolute.

SEQUENCE OF OPERATION

Numeric formulas are evaluated in the following order:

- Parentheses, from the innermost pair to the outermost
- Negation
- Exponentiation
- Multiplication and division, from left to right
- Addition and subtraction, from left to right

Negation is differentiated from subtraction in this manner: if a term is provided from which the term following the minus sign can be subtracted, the minus sign indicates subtraction. In all other cases, the minus sign indicates negation.

Examples:

- Negation: 2^{*-5} , $15/-10$, $-AA$
- Subtraction: $2^{**4}-5$, $15/5-10$, $BB-AA$

Contrary to the usual rules of algebra, negation is performed before exponentiation. Under usual algebra rules the expression $-3^{**2}-5$ would be evaluated as $-(3^{**2})-5$ yielding $-9-5$ which reduces to -14 . JOVIAL treats a leading minus sign as a negation of the expression which follows immediately, causing the expression to be evaluated as $(-3^{**2})-5$ which yields $9-5$ reducing to 4 .

Likewise, the expression $-2^{**2}+5$ will be evaluated as follows. Usual algebra rules will evaluate it as $(-2^{**2})+5$ yielding $4+5$ or 9 . JOVIAL will evaluate it as $-(2^{**2})+5$ yielding $-4+5$ or 1 . If there is any doubt as to the order of evaluation the parentheses should be used, as shown above, since they are always acceptable.

The following examples illustrate the high priority of negation and low priority of subtraction:

Examples:

- $2+-2^{**2} = 6$
- $-2^{**2}+1 = 5$

Parentheses () and (/ /) have the highest priority. The following example shows how grouping by parentheses affects the sequence of operations:

$$(AA+2) / ((/XTERM/) + (PI ** (2** CHI))) - 3.25$$

The order of evaluation is:

2 ** CHI = a (Lower-case letters indicate temporary results.)
PI ** a = b
(/XTERM/) = c
c + b = d
AA + 2 = e
e / d = f
f - 3.25 = final result

Some further examples will help clarify the use of parentheses and the priority of operations:

$$\begin{array}{ll} 2 + 3 ** 2 = 11 & 2 + (3 ** 2) = 11 \\ - (2 + 3) ** 2 = 25 & -2 + 3 ** 2 = 7 \\ 2 / 2 / 2 = .5 & 2 / (2/2) = 2 \end{array}$$

MODE OF RESULTS

Three types of JOVIAL computations are possible: integer, fixed-point, and floating-point. Numeric formulas may contain each of these types of values.

Arithmetic computations are performed to yield integer, fixed-point, or floating-point results. The mode of the results depends on the type of the operands that are combined. The possible combinations and their results are detailed in Table 3-1.

Code to perform conversions between operand types is automatically provided by the compiler. Execution time is conserved when the operand types are the same, e.g., integer and integer. Although this is probably not feasible for the entire program, it is advisable to attempt it in the areas of heaviest computation. In terms of the hardware, use of all floating-point variables is the most efficient. The rules for integer and fixed-point results are given in Appendix J.

TABLE 3-1. OPERAND TYPES AND RESULTS

Operand Types	Mode of Results
Integer and Integer	Integer
Integer and Fixed-Point	Fixed-point (integers can be treated as if they were fixed-point variables with zero fractional bits carried to infinite precision)
Fixed-Point and Fixed-Point	Fixed-Point
Floating-Point and Integer	Floating-Point
Floating-Point and Fixed-Point	Floating-Point
Floating-Point and Floating-Point	Floating-Point

LITERAL FORMULAS

A literal formula is an octal constant, literal constant, literal variable, or a BYTE functional modifier. Literal constants or variables are either display code or transmission code (see Appendix F).

Example:

O(0506) 13H(TOTAL EXPENSE) 11T(VALUE OF 22)

BOOLEAN FORMULAS

A Boolean formula can always be reduced to the value 1 or zero. The simplest Boolean formula is a Boolean variable. Boolean formulas can always be formed by combining non-Boolean constants, variables, formulas, and file names, with relational operators. This is called a relational Boolean formula. Logical operators can be combined with one or more Boolean formulas to form a complex formula called a logical Boolean formula. The ODD functional modifier is a Boolean formula.

RELATIONAL BOOLEAN FORMULAS

Relational formulas provide facility for comparing quantities and taking a different action based on the comparison. The general form is:

formula₁ relational operator formula₂

The result of a relational formula is always true or false.

The relational operators are:

EQ	Is equal to
NQ	Is not equal to
GR	Is greater than
GQ	Is greater than or equal to
LS	Is less than
LQ	Is less than or equal to

The formulas on either side of a relational operation must be compatible and non-Boolean. Compatible means that both formulas are of the same type; i. e. , numeric, literal, entry, or status.

NUMERIC RELATIONALS

Integer relational comparisons are done in integer compare mode.

Relational comparisons with fixed-point operands, or fixed-point and integer operands are done in fixed-point mode. That is, the binary points are aligned and the comparison made to the precision of the least precise operand, except when an integer is compared to a fixed-point operand with negative scaling factor. In this case, the comparison is carried out to the binary point. In either case, the integer operand is treated as a fixed-point operand with zero fractional bits to infinite precision. If either or both operands are floating-point, the comparison is done in floating-point mode. Comparing a floating-point formula for equality is extremely risky because of the floating-point hardware algorithms. Instead, the better method would be to subtract and take the absolute value of the difference and compare it against a small constant. Note also that the binary point alignment required for comparison of a fixed-point operand with an integer operand may cause the loss of the high-order bits during the comparison.

Examples:

- IF INT EQ 4 \$ Tests whether the value of the integer variable INT = 4.
- IF FIX (\$3\$) LS FIX(\$6\$) \$ Tests whether the fixed-point value of FIX(\$3\$) is less than that of FIX(\$6\$).
- IF ABS(FLOAT1 - FLOAT2)
LS 0.000000001 \$ Two floating-point variables are compared by taking the absolute value of the difference and comparing against the desired degree of accuracy.

LITERAL RELATIONALS

When comparisons involve literal formulas of different length, the shorter formula is prefaced with blanks of the type of the formula. Comparison is made only to the length of the longest operand. Hollerith formulas may be compared only for equality or inequality. Transmission code formulas may use all comparisons including greater than or less than.

Examples:

- IF HOL EQ 10H(NUMBER)\$ Compares the value of HOL with a 10-byte literal.
- IF BYTE(\$5\$(STC) LQ O(37)\$ Checks to see if byte 5 of the transmission code item (STC) is less than or equal to octal 37; that is, to determine if it is an alphabetic character.
- IF HOL EQ HOL2\$ Compares two Hollerith variables.

ENTRY RELATIONALS

Entries may be compared only for equality, inequality, or zero. If a comparison involves entries of different lengths, the shorter is prefaced with octal zeros.

Examples:

- IF ENTRY(\$5\$(TABA) EQ 0 \$ Determines if all bits of entry 5 of table TABA are set to zero.
- IF ENT (\$XX\$(TABA) NQ ENTRY (\$7\$(TABB)\$ Determines if the entry specified by variable XX of table TABA is not equal to entry 7 of table TABB.

STATUS RELATIONALS

Status relational formulas have a restricted order. The format is:

$$\left\{ \begin{array}{l} \text{status-variable} \\ \text{file-name} \end{array} \right\} \quad \text{relational-operator} \quad \left\{ \begin{array}{l} \text{status-constant} \\ \text{status-variable} \\ \text{status-function} \end{array} \right\}$$

Examples:

FILEA	Already declared with status values V(READY), V(EOF), and V(ERROR).
CHRR	Status item CHRR already declared with status constant values V(A), V(B), V(C), V(D), V(E), V(F), V(G).
● IF FILEA EQ V(EOF) \$	Tests whether an end-of-file was encountered on the last operation on FILEA.
● IF CHRR LS V(D) \$	Tests whether the status variable CHRR is set to a value less than that of status constant V(D), that is, whether CHRR is set to V(A), V(B), V(C).

Note that the value of a status constant for a particular status variable depends on the order in which it was declared on the status constant list.

LOGICAL BOOLEAN FORMULAS

Logical Boolean formulas are formed by combining Boolean formulas with logical operators to produce a complex Boolean formula, or to reverse the value of a Boolean formula.

<u>Logical Operators</u>	<u>Description</u>
AND	Combines two Boolean formulas to produce a formula which is evaluated as true if both formulas joined by AND are true.
OR	Combines two Boolean formulas to produce a formula which is true if either formula joined by OR is true.
NOT	Precedes a Boolean formula to produce a formula which reverses the value of the formula following NOT.

Examples:

● FORM1 AND FORM2	If both FORM1 and FORM2 are true, the result is true.
● FORM1 OR FORM2	If either FORM1 or FORM2 or both FORM1 and FORM 2 are true, the result is true.

- IF NOT FORM1

If FORM1 is true, the formula is false. If FORM1 is false, the formula is true.

SHORTHAND NOTATION

A shorthand notation may be used for numeric and literal relationals to represent the conjunction of two relational formulas whose right and left operands are identical. This notation consists of omitting the AND operator and one occurrence of the duplicate operand. The AND operator and the duplicate operand are implied though not explicitly stated.

Explicit

IF 21 LQ AGE AND AGE LQ 30

IF AA LS I + 2 AND I + 2 LQ BB (\$1\$)

Implicit

IF 21 LQ AGE LQ 30

IF AA LS I + 2 LQ BB (\$1\$)

SEQUENCE OF EVALUATION

Parentheses can be used to determine the hierarchy of evaluation of a Boolean formula; otherwise, evaluation is in the following sequence:

1. NOT
2. AND
3. OR

Evaluation of components at the same hierarchy level takes place from left to right. Evaluation is complete as soon as evaluation of any part of the formula has determined the result. For instance, in a set of formulas connected by ANDs, the value of the entire set is false as soon as any component is found to be false. Similarly, a set of formulas connected by ORs is true as soon as one formula is found to be true.

Example:

- (FORM1 OR FORM2) AND (FORM3 OR FORM4) OR FORM5

If FORM1 and FORM3 are both true, no other formula need be tested to determine if the entire formula is true.

If FORM1 and FORM2 are both false, FORM3 and FORM4 need not be tested to prove the AND relation false; but FORM5 must be evaluated to determine whether the entire formula is true or false.

NOT complements all relational and logical operators in the formula immediately following it:

NOT AFORM EQ 50 Is equivalent to AFORM NQ 50

Parentheses can be used to group formulas:

NOT(AFORM EQ 50 AND BOOL) Is equivalent to AFORM NQ 50 AND NOT BOOL

If parentheses are omitted, only the formula immediately following NOT is complemented:

NOT AFORM EQ 50 AND BOOL Is equivalent to AFORM NQ 50 AND BOOL

FUNCTIONS

A function is not a true formula, but is a special form of a procedure that is similar to a formula in that it represents a single value and can appear as a component of a formula. A function is a call to a procedure that returns a single value which can be numeric, Boolean, literal, or status. For full details of functions, see Section 6. The format of a function is:

function-name ([parameter list])

The name is a JOVIAL name which identifies the function; if there are no parameters, the name must be followed by a pair of empty parentheses to identify it as a function. The function is automatically invoked to compute the function value each time the name appears in a formula. The function type and its properties are specified in the declaration. A function of a particular type may be used any place where a variable of that type can be used.

Examples:

- COS (ANGLE) The library function that computes the cosine of a floating-point number is invoked yielding a floating-point result.
- SEARCH() The SEARCH function is invoked if it has been declared in the program being compiled, or in a COMPOOL used when the program was compiled. This function has no input parameters. It uses variables common to the function and to the main program.
- RANGE(LATITUDE, LONGITUDE) The RANGE function, which has two input parameters, computes a single output value.

COMMENTS

Comments can be interspersed between or within JOVIAL statements and declarations. Comments are enclosed within a pair of double primes. The string of characters may not contain:

- Two primes in succession
- A terminal prime
- A \$ except in the context of index brackets: (\$ \$)

The compiler permits comments in source program listings, but otherwise considers them as spaces. Comments are legal anywhere a space is legal, except:

- Within literal constants
- Within the name field of a DEFINE statement
- Within another comment

Examples:

- A comment can be used to name the program and to give the date and subject of a program:

```
START $ ' ' JOHN DOE 1/4/71 PROGRAM TO COMPUTE PROBABILITY  
OF SURVIVAL' '
```

- Comments can be used to describe:

```
ITEM ACTYPE H 1 0 0 $ ' 'AIRCRAFT TYPE' '  
ITEM ACNUM H 4 0 6 $ ' 'AIRCRAFT NUMBER' '
```

- Comments can be embedded within a statement:

```
IF ' 'CURRENT' ' RANGE GR' 'EATER THAN MAXIMUM' ' LIMIT$
```

In this case, the compiled statement is:

```
IF RANGE GR LIMIT$
```

DEFINE DIRECTIVES

The DEFINE directive allows the user to assign a name to a string of characters which can thereafter be referenced by this name. With this feature, the user can abbreviate lengthy expressions, make simple additions to the language and create symbolic parameters. The form is:

```
DEFINE name ' 'character string' '$
```

The name is any user-defined name. The character string can include any JOVIAL characters except the double prime. The pair of double primes is required to enclose the character string. Whenever the defined name appears following its definition, it is replaced by the string of characters. The definition remains in effect until the end of the program or until the name is redefined in a subsequent define directive. A defined name may be redefined, but cannot be shut off by using its own name as the definition:

```
DEFINE ITEMA ' 'ITEMA' ' $      Is a circular definition and must be avoided.
```

The following type of circular definition should also be avoided:

```
DEFINE AA ' 'BB' '
DEFINE BB ' 'AA' '
```

The defined name can be used in any context except within a status or literal constant, a comment, or in direct code other than assignment statements. No defined name can expand to more than 320 characters.

Examples:

- ```
DEFINE TOTAL ' 'FORM1
+ ITEMA *HOURS ' '$
```

 A subsequent reference such as `IF TOTAL GR 1000$` is equivalent to `IF FORM1 + ITEMA * HOURS GR 1000$`.
- ```
DEFINE INTEGER ' 'I 60 U ' '$
```

 Any subsequent reference to `INTEGER`, such as `ITEM AA INTEGER $`, is equivalent to `ITEM AA I 60 U $`.
- ```
DEFINE FIELDA ' 'BYTE
($16, 5$)(HOL)' '$
```

`IF FIELDA EQ 5H(START)$` is equivalent to `IF BYTE ($16, 5$)(HOL) EQ 5H(START)$`
- ```
DEFINE SUM ' 'TOTAL =
VARA + VARB + VARD+
FUNC(VARE)$' '$
```

 This example has the effect of inserting the line of code to set `TOTAL` to the sum of variable `VARA`, `VARB`, `VARD`, and function `FUNC` operating on variable `VARE`. This statement will be compiled in line each time the word `SUM` occurs in the source.

JOVIAL programs consist of declarations that define data and program structures, and of statements that indicate the operations to be performed upon the data.

Data structures to be used by program statements may be explicitly described in a data declaration within the program, or within a COMPOOL used when the program is compiled. Data declarations are used to define items, arrays, tables, and files. The use of an undefined name within the program or the COMPOOL will cause it to be MODE-defined at that point implicitly within the current MODE directive description. Arrays and tables must be defined explicitly. Items, arrays, and tables are described in Section 5; files, in Section 6.

Program structures are the closed sections of code which may be invoked from elsewhere within the program to perform a specified task. Program structures must be declared within the program, or described in a COMPOOL used in compiling the program which calls them. Program structures consist of subprograms and closed forms (closed routines, procedures, and functions). Closed forms are described in Section 7; subprograms, in Section 8.

STATEMENT FORMS

The types of statement forms are simple, named, compound, and complex. Simple and compound statements are called independent statements.

- A simple statement is a single statement, such as an assignment or a GOTO statement.
- A named statement is a statement preceded by a statement label or label list to permit control to be transferred to the named statement.
- A compound statement is two or more statements grouped within BEGIN-END brackets; this is treated as a single statement.
- A complex statement is a statement composed of a statement clause, i. e., IF, FOR, IFEITH, ORIF followed by a simple or compound statement. Complex statements provide the conditional flow and looping facilities of JOVIAL.
- Every statement is terminated by a \$.

The rules governing the usage of JOVIAL data forms, functions, and functional modifiers are very broad and flexible. Thus it is possible to code powerful and complex statements. The

complexity of a single statement may vary depending upon the types of operations used. But in all instances, the compiler will accept a statement of 100 symbols maximum.

STATEMENT LABELS

A statement can be identified by a programmer-supplied name followed by a period. Spaces preceding the period are allowed. This identifier is called a statement label. Its format is:

name.

where

name = programmer-supplied identifier for the statement

STATEMENT LABEL LIST

A named statement can have more than one label in the form of a statement label list:

name₁. [(name₂ name_n.)]

where

name₁., name₂., name_n. = programmer-supplied identifiers, each of which is followed by a period

Examples:

- PROG'END. STOP\$
- BRANCH. JUMP. TRANSFER. ETC.
- AA. AA=10\$

Whenever [name.] or statement-label-list is used in this section, one or more statement names may be used. A statement may be referenced by any of the names in the statement-label-list preceding that statement. The statement following the name or statement-label-list is referred to as a named statement.

ASSIGNMENT STATEMENTS

The assignment statement is used to assign a variable. The format is:

[name.] variable = formula\$

The formula is evaluated and the resulting value is assigned to the variable. If the variable appears within the formula; then the old value of the variable is used in computing the formula.

Assignment statements may be numeric, literal, status, Boolean, or entry. The variable and the formula must agree in type; that is, a numeric formula should be specified when the variable is numeric.

NUMERIC

A numeric assignment statement assigns the value of a numeric formula to a numeric variable. The format is:

```
[name.] numeric-variable = numeric-formula$
```

When the receiving variable and the final value of the formula are both integer or both fixed-point but of differing precision, the precision of the final value of the formula will be adjusted to that of the receiving variables.

When the receiving variable and the final value of the formula being assigned are of differing arithmetic modes, the final value of the formula is converted to conform to the arithmetic mode of the receiving variable. For example, if the final value of a formula is an integer value but the mode of the receiving variable is floating-point, the integer value will be converted to a floating-point value prior to being stored in the floating-point variable. These processes may result in the loss of precision due to truncation. Care should be taken in specifying mixed-precision or mixed-mode assignment operations to insure that the results are predictable. The assignment process is accomplished as follows:

- The binary points of the receiving variable and the formula value are aligned.
- The bits representing the value of the formula to the right side of the binary point are truncated to fit within the defined boundary of the receiving variable, if necessary. If truncation occurs in this case, least significant bits of the formula value will be lost.
- The bits representing the value of the formula to the left side of the decimal point are treated in the same manner. In a simple item, array, or table item with no packing, the integer bits will be truncated only if the bits extend beyond the boundary of the word containing the item. This occurs when more integer bits are set in the word containing the item than specified in the declaration. Any further computations involving this variable would risk having to use the stored number with undefined results. In dense-packed and defined entry tables, the integer portion will be truncated at the boundary of the item. In either situation the sign bit will be used for storage if required. In this situation, the most significant bits will be lost.

Both integer and fractional bit truncation can occur in a single assignment statement. For example, consider a formula whose final value (in binary notation) is:

10111001.1011011

The value is being assigned to a variable defined as having only three bits of precision on both sides of the binary point; that is, its picture is:

000.000

The result of the assignment is:

10111001.1011011	formula value
000.000	variable size
001.101	value stored
10111 1011	truncated bits

The above example refers primarily to the assignment of formula values to fixed-point and integer variables.

When nonmatching values are being assigned to floating-point variables, conversion is made and truncation can occur. Since the internal representation of a floating-point value cannot be programmer-defined, the results of assignment into a floating-point variable are more predictable. The programmer need only be aware of the fact that the conversion to floating-point notation may result in only an approximation of the original fixed-point or integer value. That is, some fixed-point and integer values may assume precision characteristics that cannot be precisely expressed in floating-point notation. As long as the values involved do not exceed hardware limitations, truncations that occur as a result of assigning the fixed-point or integer values to floating-point variables will always result in the loss of least significant bits.

Examples:

- Floating-point assignment statements:

FLOAT = INTEGER \$	The value of the integer item, INTEGER, will be assigned to the floating-point item FLOAT.
FLOAT = FIXD \$	The value of the fixed-point item, FIXD, will be assigned to the floating-point item FLOAT.
FLOAT = O(124) \$	The decimal equivalent of the octal constant, O(124), will be assigned to the floating-point item FLOAT. The decimal equivalent is 84.

The results of the floating-point assignments may not be exact due to the nature of floating-point representation.

- Fixed-point assignment statements:

```
ITEM FLT F P 13.137 $
ITEM FIX A 11 S 4 $
ITEM LFIX A 10 U 5 P
  15.40265 $
ITEM BFIX A 7 U 3 P 11.125 $
ARRAY AFIX 10 A 7 U 4 $

TABLE V 5 1 $
BEGIN
ITEM FIL1 I 17 S 0 0 $
ITEM DFIX A 7 U 4 0 17 $
ITEM FIL2 I 18 S 0 24 $
END
```

These data descriptions apply to the following examples.

FIX = FLT \$

The value of the floating-point variable (FLT = 13.137) will be assigned to the fixed-point variable FIX. Due to the nature of floating-point representation, FIX will be set to 13.125 because of the inability to represent the decimal number .137 in the four fractional bits.

FIX = O(27)

FIX is set to the decimal equivalent of the octal constant. Octals always set the integer portion; fractional bits cannot be set by an octal constant. The decimal equivalent is 23.

AFIX (\$0\$) = BFIX \$

The value of the simple fixed-point item, BFIX = 11.125, with four integer bits and three fractional bits will be assigned to an array item (AFIX) which has three integer bits and four fractional bits. Since AFIX has more fractional bits than BFIX, there is no problem, and since AFIX has fewer integer bits than BFIX, the additional bits will be set. That is, more bits will be set than specified in the ARRAY declaration. The extra bit will be used for any further computations involving AFIX(\$0\$).

<u>Item</u>	<u>Decimal</u>	<u>Octal</u>	<u>Bit Pattern</u>
BFIX	11.125	131	0001011.001
AFIX(\$0\$)	11.125	262	0001011.0010

DFIX (\$0\$) = LFIX \$

The value of the simple fixed-point item, LFIX = 15.40265, with 10 integer bits and five fractional bits will be assigned to a dense-packed table fixed-point item, DFIX, which has only seven integer and four fractional bits. In this case, both left and right truncation will occur, as detailed below:

	<u>Item</u>	<u>Decimal</u>	<u>Octal</u>	<u>Bit Pattern</u>
Before	{Entry ₀	15.40265	00000 00000 00000 00755	001111.01101
	{LFIX		755	
After	{DFIX (\$0\$)	7.375	00000 00077 75400 00000	00111.0110
	{Entry ₀		166	

• Integer assignment statements:

ITEM INT I 7 S \$
ITEM FLOT F P 123.456 \$

These data descriptions apply to the examples below.

INT = O(30) \$

INT is set to the decimal equivalent of the octal constant. The decimal value is 24.

NINT (\$0\$) = INT \$

The value of the integer item, INT = 127, with eight integer bits will be assigned to a non-packed table item NINT, which has seven integer bits specified.

TINT (\$0\$) = FLOT \$

The value of the floating-point item, FLOT = 123.456, will be assigned to a defined table item TINT, which has seven integer bits. In this case, right truncation of the fractional bits, left truncation of the integer bits, and the setting of the sign bit causing it to become negative will occur. The word containing the item TINT (\$0\$) will be set to -4.

LITERAL

A literal assignment statement assigns the value of a literal formula to a literal variable.

The format is:

[name.] literal-variable = literal-formula\$

No conversion is performed if the variable and the formula are of different codes, i. e., if one is in transmission code and the other in Hollerith code. If the sizes differ and the

formula is longer than the variable, the formula is truncated on the left; if the formula is too short, it is prefaced on the left with blanks of the type of the formula, H or T.

Examples:

- The variable DISP is defined as display code (H) with a size of four bytes.
 - DISP = 4H(ABCD) \$ DISP is set to an octal value of 01020304, the equivalent of Hollerith characters ABCD.
 - DISP = 5H(ABCDE) \$ DISP is set to an octal value of 02030405, the equivalent of Hollerith characters BCDE. Left truncation will occur.
 - DISP = 3H(ABC) \$ DISP is set to an octal value of 55010203, the equivalent of Hollerith characters ΔABC. Left padding will occur.
 - DISP = O(01020304) \$ DISP is set to an octal value of 01020304, the equivalent of 4H(ABCD).
 - DISP = 4T(WXYZ) \$ DISP will be set to an octal value of 34353637, the equivalent of STC characters WXYZ.
- The variable TRANS is transmission code (T) with a size of four bytes.
 - TRANS = 3T(XYZ) \$ TRANS is set to an octal value of 05353637, the equivalent of STC characters ΔXYZ. Left padding will occur.
 - TRANS = 5T(VWXYZ) \$ TRANS is set to an octal value of 34353637, the equivalent of STC characters WXYZ. Left truncation will occur.
 - TRANS = O(34353637) \$ TRANS is set to an octal value of 34353637, the equivalent of 4T(WXYZ).
 - TRANS = 3H(ONE) \$ TRANS is set to an octal value of 55171605, the equivalent of Hollerith characters ΔONE. Note that since the formula was shorter than the receiving literal, it was right-justified in the literal and padded on the left with a Hollerith blank. Hollerith values are used even though TRANS was defined as STC.
- The variable HOLV is defined as Hollerith with a size of 17 bytes, and the variable HOLD is defined as Hollerith with a size of 13 bytes.

HOLD = HOLV \$

Since the formula HOLV is longer than the variable in the assignment HOLD, it is right-justified and left-truncated. That is, this move has the same result as moving BYTE (\$4, 13\$) of HOLV to HOLD.

HOLV = HOLD \$

Since the formula HOLD is shorter than the variable HOLV, it is right-justified and the remaining four bytes on the left are padded with Hollerith blanks.

- The variable TTT is defined as a transmission code with a size of seven bytes, and the variable HHH is defined as Hollerith with a size of 12 bytes.

HHH = TTT \$

Since TTT is shorter than HHH, it will be right-justified in HHH and the five remaining bytes on the left will be padded with transmission code blanks. The rule for padding specifies that blanks of the type of the formulas will be used, regardless of the type of the assignment variable.

STATUS

A status assignment statement assigns the value of a status formula to a status variable. The format is:

[name.] status-variable = status-formula\$

If the status formula is a status constant, it must appear in the declaration of the variable. If the formula is a status variable, it should have a value from zero to 1 less than the number of constants in the status list of the status variable.

Examples:

ASTATUS is declared as ITEM ASTATUS S V(A) V(B) V(C) \$

BSTATUS is declared as ITEM BSTATUS S V(X) V(Y) V(Z) P V(Y) \$

- ASTATUS = V(B) \$ The item is set to the status constant V(B).
The integer value would equal 1.
- ASTATUS = BSTATUS \$ ASTATUS is set to V(B) since BSTATUS is
preset to the second constant in its status
constant list. • The integer value would equal 1.

BOOLEAN

A Boolean statement assigns the value of a Boolean formula to a Boolean variable. The format:

[name.] Boolean-variable = Boolean-formula\$

The Boolean formula can be one of the Boolean constants, 1 or zero; 1 signifying if the formula is true, zero if the formula is false.

Examples:

- `BOOL = 1$`
- `BOOL1 = 0$`
- `BOOL = AA GQ 18 AND AA LQ 26 AND BB EQ V(TRUE) $`
- `BOOL = BOOL1$`
- `BOOL = ODD(INT)$` – where INT is an integer item.

ENTRY

An entry assignment statement assigns the value of an entry formula to an entry variable. The format is:

[name.] entry-variable = entry-formula\$

The entry formula can be only another entry variable or zero. If the formula is zero, every bit in the variable is set to zero. When the sizes differ, the formula is either right-justified and left-truncated, or zero-filled on the left to the entry variable size.

Examples:

- `ENT(TABL(0)) = 0$` The first entry of TABL is set to zero.
- `ENT(TABL(1)) = ENT(TABA(1))$` The second entry of TABL is set to the value of the second entry of the TABA.

EXCHANGE STATEMENTS

An exchange statement is a shorthand method of exchanging the value of two variables of the same type. The variable types may be numeric, literal, status, Boolean, or entry. The format is:

[name.] variable₁ == variable₂ \$

No spaces are allowed between the equal signs. The compiler generates a temporary storage area for the move.

The exchange statement VAR1==VAR2\$ has the same effect as:

```
TEMP = VAR1 $
VAR1 = VAR2 $
VAR2 = TEMP $
```

In most instances, the left and right sides of the exchange statements are interchangeable, that is, AA == BB \$ produces the same result as BB == AA \$. The exception to this is where the specification of one variable is dependent upon the variable on the other side of the exchange statement. The possible conditions are:

- If the two variables overlay each other.
- If the value of one variable is used in computing the positioning of the other. For example, in the statement AA == BB(\$AA\$), the value of one variable is used to compute the subscript of the second variable; or in the statement (BIT(\$BB, 20\$)(AA) == BB \$, one variable is used to specify the bit positions used in a second variable.

In these instances it would be advisable to use the expanded form, and specify the moves and the exact method of evaluation desired.

NUMERIC

A numeric exchange statement exchanges the values of two numeric variables. The format is:

```
[name.] numeric-variable1==numeric-variable2 $
```

When the variables are both integer or both fixed-point but of differing precision, the precision of the sending variable will be adjusted to that of the receiving variable. When the variables are of differing arithmetic modes, the sending variable will be converted to the mode of the receiving variable. This may result in a loss of precision to both integer or fractional bits on both variables. If one of the variables is floating-point and the other is integer or fixed-point, the results of exchange may not be exact, due to the nature of floating-point representation. Rules for mode or precision matching are listed under Assignment Statements. Examples of numeric exchange statements are listed in Appendix K.

LITERAL

A literal exchange statement exchanges the values of two literal variables. It has the same effect as two simultaneous assignment statements. The format is:

```
[name.] literal-variable1 == literal-variable2 $
```

No conversion of codes takes place when Hollerith or STC literals are exchanged. As with literal assignments, if the lengths of the variables are not the same, the sending variable will be right-justified in the receiving variable.

Example:

VAR1 = transmission code

VAR2 = Hollerith code

- VAR1 == VAR2 \$ VAR1 is set to Hollerith code value. VAR2 is set to transmission code value.

STATUS

A status exchange statement exchanges the values of two status variables. The format is:

[name.] status-variable₁ == status-variable₂ \$

If the value of the status variable on either side of the exchange statement exceeds the range of the status constant list for the other status variable, the results of the exchange are unpredictable.

Examples:

- ITEM ASTATUS S V(A0) V(A1) V(A2) V(A3) V(A4) V(A5) V(A6)
P V(A3) \$ Since ASTATUS was preset to V(A3), the fourth in the list, it originally had the integer value 3. Likewise, since V(B0) was the first status value in the list for BSTATUS, it will be preset to integer value zero. The exchange statement will exchange the values. ASTATUS will be set to integer value zero indicating status value V(A0), and BSTATUS will be set to integer 3, the value for V(B3).
- ITEM BSTATUS S V(B0) V(B1) V(B2) V(B3) V(B4) V(B5) V(B6)
P V(B0) \$
- ASTATUS == BSTATUS \$

BOOLEAN

A Boolean exchange statement exchanges the values of two Boolean variables. The format is:

[name.] Boolean-variable₁ == Boolean-variable₂ \$

Example:

- `BOOL1 == BOOL2 $` The current value of BOOL1 is exchanged with the current value of BOOL2. Both values are either 1 or zero.

ENTRY

An entry exchange statement switches the values of two entry variables. The format is:

`[name.] entry-variable1 == entry-variable2 $`

Examples:

- `ENTRY(AA(3)) ==
ENTRY(AA(6)) $` The bit patterns in entries four and seven of table AA, or the table containing item AA, are exchanged. This exchange is in the same table; therefore, the entries are the same length and no padding or truncation will be required.
- `ENTRY(TABLEB(7)) ==
ENTRY(TABLA(4)) $` In this example the eighth entry of table TABLEB (five words per entry) will be exchanged with the fifth entry of table TABLA (four words per entry). Truncation and padding will occur because the two tables have different length entries: the five words of TABLEB will be right-justified and the left word truncated in the four-word entry; the four words of TABLA will be right-justified and zero-filled on the left in the five-word entry.

CONTROL STATEMENTS

The control statements GOTO, IF, IFEITH, and ORIF enable the user to branch out of the normal serial sequence of execution. Each of these is detailed in the following pages.

GOTO

The GOTO statement transfers control to a statement designated by a sequential formula. A sequential formula is either a statement label, a switch name, or the name of a CLOSE routine. The format is:

`[name.] GOTO sequential-formula $`

)
)
)
)
When the sequential formula is a statement-label, the program branches to the statement designated by the label.

)
)
)
)
When the formula is a CLOSE routine name, the CLOSE routine is executed. Control then returns automatically to the statement following the GOTO, unless a GOTO statement within the CLOSE routine transfers control to another point in the program.

)
)
)
)
)
)
)
)
When the sequential formula is a switch name, it is either a simple name or a name followed by an index. If it is a simple name, it refers to a switch declaration containing a list of statement labels or another switch name, or CLOSE name to which control is transferred. If the switch name is indexed, the index is an integer value which refers to one of the names specified in the switch declaration. If there is no meaningful value associated with this index, the next statement in the regular sequence is executed.

Examples:

- GOTO AA \$

If AA is a statement label, the statement named by AA is executed and the program continues from that point.

If AA is a CLOSE name, the CLOSE routine AA is executed and then control returns to the statement following GOTO AA, unless a different transfer is specified in the CLOSE.

If AA is a switch name, it must be an item switch which has been set, prior to the switch call, with a simple item whose value determines where control will transfer.

- GOTO SW(\$1\$) \$

An indexed sequential formula always designates a switch name. The index either specifies the second formula in an index switch, the second occurrence of an item in a table, or a one-dimensional array specified in an item switch.

- GOTO WHICH(\$A, B, C\$)\$

WHICH names an item switch specifying a three-dimensional array. The value of the particular item in the array determines the sequential formula to which control transfers.

CONDITIONAL

A conditional statement is an IF clause followed by an independent statement which may be either simple or compound. In the event that the IF clause is true, the independent statement

is executed; when the IF clause is false, the following independent statement is skipped and the next sequential independent statement is executed. Conditional statements may not be immediately followed by another conditional statement. Thus, the following sequence is illegal;

```
IF AA GR 10 $
IF BB LS 15 $
```

The following form may be used to accomplish the stated testing:

```
IF AA GR 10 $
  BEGIN
  IF BB LS 15 $
  •
  •
  END
```

Placing BEGIN-END brackets around the conditional statements makes it a compound statement. The number of nested IF clauses is limited to 50. The format is:

[name.] IF Boolean-formula \$ [name.] simple or compound-statement \$

Examples:

- IF DIST LQ 500 \$
GOTO RUN \$
DIST = DIST - 1 \$

If the Boolean formula DIST LQ 500 is true, control transfers to the statement or CLOSE named RUN; otherwise, the variable DIST is decremented by 1.
- IF 10 - AA(\$K\$) EQ 0 \$
AA(\$K\$)=AA(\$K\$) + 1 \$

If the item AA(\$K\$) is equal to 10, it is incremented by 1.
- IF WIN \$ GOTO PROFIT \$
GOTO LOSS \$

WIN must have been declared as a Boolean variable. Its current value determines whether PROFIT or LOSS is executed.
- IF GRADES LS 75 \$
BEGIN
WARNING = V(ON) \$
CLASS = V(FAIL) \$
GOTO OUT \$
END
GOTO TEST \$

If GRADES is less than 75, the compound statement within the BEGIN-END brackets is executed; it includes a transfer to avoid TEST. Otherwise, transfer to TEST.

IFEITH AND ORIF

The IFEITH and ORIF statements are variants of a simple IF clause, providing a means of selecting one of a series of possible statements. The choice depends on the evaluation of Boolean formulas. The formats are:

```

[name.]   IFEITH Boolean-formula1 $
[name.]   statement1
[name.]   ORIF Boolean-formula2 $
[name.]   statement2
[name.]   ORIF Boolean-formulan $
[name.]   statementn
          END

```

where

statement 1, . . . , n may be either simple or compound, but not complex.

An IFEITH statement acts as an implied BEGIN. Therefore, since each BEGIN must have a matching END, the last ORIF statement must be followed by an END to match with the IFEITH statement. An IFEITH must be followed by at least one ORIF. Each Boolean formula is evaluated in turn until a true one is found, which then causes execution of its statement. If the statement being executed does not contain a GOTO, control is given to the first statement after the matching END statement. If none of the Boolean formulas is true, none of the statements is executed and control drops through to the first statement after the matching END statement. If control is passed to a labeled ORIF statement, a search begins for alternatives at that point, as if all of the preceding alternatives had been false.

IFEITH and ORIF statements can also be compound statements. An ORIF 1 \$ statement specified before the matching END statement will cause its statement to be executed if no previous statement was true. This is a convenient way of terminating the IFEITH and ORIF statements. The maximum nesting level for IFEITH statements is 50.

Examples:

- IFEITH MM EQ V(SEP) OR
MM EQ V(APR) OR MM EQ
V(JUN) OR MM EQ V(NOV)\$
DD = 30 \$
ORIF NN EQ V(FEB) \$
BEGIN
IFEITH LPTR
DD = 29 \$
ORIF 1 \$
DD = 28 \$
END "FEB"
END

The use of the Boolean constant 1 in the last ORIF clause assures that all possible alternatives are considered.

```

ORIF 1 $
  DD = 31 $
END 'MONTH' '
GOTO DATE'SET $

```

- IFEITH AA LS BB \$


```

AA = BB $
T1. ORIF AA + BB GR 10 $
  BEGIN
    CG = (AA+BB)/2 $
T2. AA = CG + 1 $
    BB = AA + 1 $

    GOTO REMAIN $
  END
  ORIF 1 $
  GOTO LEAVE $
END 'IFEITH' '

```

Entry at the beginning tests all the alternatives. Entry at T1 tests the second and third alternatives only. If the second alternative is true, all four following statements are executed and control transfers to REMAIN. Entry T2 does not test any alternative; AA and BB are set and control leaves the IFEITH-ORIF statement.

The comment following END is used here to identify the END as the terminator of the IFEITH clause.

LOOP STATEMENTS

The loop statement provides for the repeated execution of an independent statement and the activation of a loop variable for use by the independent statement. It may also provide for incrementing or decrementing the loop variable, testing it, and branching according to the results. The FOR clause activates the loop variable and begins the iterative process; the TEST statement can be used within a FOR clause to transfer control to the implicit modify-test-branch section of a FOR clause.

FOR CLAUSE

The complete FOR clause defines a loop variable, establishes its initial value, specifies increment or decrement value, and declares the terminal value. The loop variable is only active for the independent statement following the FOR clause. It is an integer variable and may be used by the independent statement. The values in the FOR clause may be specified explicitly as integer constants, or may be numeric formulas. The number of times the statement is to be iterated is specified by the formulas in the FOR clause, unless altered by the independent statement. The format is:

```

name. FOR-letter = formula1 [, formula2 [, formula3]]$

```

where

FOR-letter = a single letter which is set to formula₁

formula₁ = specifies the initial value

formula₂ = specifies the increment or decrement

formula₃ = specifies the terminal value

FOR STATEMENT

A FOR statement is a complex statement defined as a FOR clause, followed by a simple or compound statement.

The loop variable is the only instance in which a single letter variable may be used. The formats are:

[name.] FOR loop-variable = formula₁ [, formula₂ [, formula₃]]\$

[name.] FOR loop-variable = ALL(name)\$

where

loop variable = a single letter which is set to formula₁.

formula₁ = the initial value.

formula₂ = an increment or decrement.

formula₃ = the terminal value of the loop variable.

ALL(name) = name is table or table item name. This is equivalent to specifying NENT (name)-1 for formula₁, -1 for formula₂, and 0 for formula₃. It is a special shorthand form for processing active entries, based on the value of a table NENT.

The formulas may be any numeric formulas yielding a positive or negative value between $-2^{17} - 1$ and $+2^{17} - 1$. If the result of the formula is not integral, it is truncated to obtain an integer. The FOR statement must be immediately followed by the loop statement; data declarations are not permissible between the FOR statement and its loop statement.

When a FOR statement is executed, the sequence of steps is as follows:

1. Initialize Assign the initial value to the loop variable.
2. Execute Execute the loop statement.
3. Modify Increment or decrement the value of the loop variable by increment or decrement value.
4. Test Compare the value of the loop variable by the value of the terminal value.

5. Iterate If increment or decrement value is greater than or equal to zero, and if the loop variable is equal to or less than the terminal value, return to step 2 above. Similarly, if the increment or decrement value is negative, and the loop variable is greater than or equal to the terminal value, there is an automatic return to step 2.
6. Exit When the increment or decrement value is greater than or equal to zero, and the loop variable is greater than the terminal value, control passes to the next statement after the FOR clause. Similarly, when the increment or decrement value is less than zero, and the loop variable becomes less than the terminal value, control passes to the next statement.

The loop variable is defined as soon as the initial value is assigned to it. It remains defined within the FOR loop until control is transferred to a point outside of the loop. The loop variable is undefined when the program is executing outside of the loop. It can be used in the increment or decrement value and in the terminal value of the FOR clause, and can also be included in other FOR clauses within the original FOR loop.

Other FOR clauses, procedures, functions, and subprograms can be called from within a FOR loop without affecting the active status of the loop variable. Direct reference to the loop variable is not allowed within the procedure, function, or subprogram called, but it can be used as an actual parameter (input or output) in the calling statement.

ONE-COMPONENT FOR STATEMENT

This FOR statement does not control the program, it merely defines a loop variable for a specified period. No modification, testing, or iteration is performed. The format is:

FOR loop-variable = formula \$

Example:

- FOR Z = 0 \$
 - BEGIN
 - AA. IF ITEMZ(\$Z\$) GR AVERAGE \$
 - GOTO EXEC \$
 - Z = Z + 1 \$
 - IF Z LQ 999 \$
 - GOTO AA \$
 - END ' 'Z' '

The BEGIN-END brackets define the limits of the variable Z. The FOR statement could have been written by using the three-component FOR statement:

```
FOR Z = 0, 1, 999 $
  BEGIN IF ITEMZ($Z$) GR AVERAGE $ GOTO EXEC $ END
```

The IF statement following a one-component FOR statement is allowed; but if the FOR statement had more than one component and thus implied iteration, the IF statement would have to be enclosed in BEGIN-END brackets.

A one-component FOR statement may be used where it is convenient to define a loop variable to represent the integer value of an expression in several independent statements. This may be accomplished by enclosing the independent statement in BEGIN-END brackets to form a compound statement following the one factor FOR clause, which defines the loop variable. While this may save coding effort, it will not necessarily result in more efficient code. The computation of expressions and their storage in the index registers is decided by the scheduling algorithm in the local optimizer.

Example:

```
• FOR A = XX + YY($5$) $
  BEGIN 'A' '
    ZZ = A + 5 $
    •
    •
    •
    INT = ARRY($A$) $
    •
    •
    •
    SUB = A $
  END 'A' '
```

The use of a loop variable eliminates the need to write out the formula each time it is used.

TWO-COMPONENT FOR STATEMENT

A FOR statement with two components defines the initial value of the loop variable and its increment or decrement. The format is:

```
FOR loop-variable = formula1, formula2 $
```

This sets up an endless loop because the formula to provide a terminal value with the implicit test to end this loop is omitted. Therefore, it is necessary to provide an exit from the loop after a specific number of increments. Consider the example given below:

```

FOR A = 0, 2 $
BEGIN
  TABLA ($A$) = 0 $
END

```

After the statement within BEGIN-END is executed, the loop variable A is incremented by 2 and the statement is executed again. A statement to test A and exit must be inserted in order to avoid an infinite loop. Thus:

```

FOR A = 0, 2 $
BEGIN
  TABLA ($A$) = 0 $
  IF A GR 100 $ GO TO FIN $
END

```

THREE-COMPONENT FOR STATEMENT

This is the complete FOR statement; an initial value is set, an increment or decrement is specified, and a terminating value is set to the loop variable. The format is:

```

FOR loop-variable = formula1, formula2, formula3 $

```

After each execution of the loop statement, the following occurs:

- The loop variable is incremented or decremented by the increment or decrement value (formula₂).
- The loop variable is compared with the terminal value (formula₃).
- If the increment or decrement value is positive, the direction is assumed to be from low to high, so the loop variable is tested to see if it is greater than the value of the terminal formula. If the test is successful, the iteration is complete, the loop variable is deactivated and the next independent statement is executed. If the test fails, the independent statement is iterated again.
- If formula₂ is negative, the direction is assumed to be from higher to lower, and the variable is tested to see if it is less than the value of formula₃. This test is identical to the test described when formula₂ is positive.

Examples:

- Given a table, MANPOWR, find the age of the oldest male employee with a salary less than 201.35. (Assume the existence of a valued table with the characteristics defined below.)


```

TABLE MANPOWR V 10000 P 2 $
  BEGIN
    ITEM NUMBER H 800 $
    ITEM SALRY A 40 U 7 1 0 $
    ITEM SEX S 1 V(MALE) V(FEMALE) 1 40 $
    ITEM AGE I 7 U 1 41 $
  END
  ITEM OLDST I 60 S P 0 $
  FOR A = 0, 1, 9999 $
  BEGIN 'A' '
    IF AGE ($A$) GR OLDST AND SEX EQ V(MALE) AND SALRY LS
      201.35 $
    OLDST = AGE($A$) $
  END 'A' '

```

- The FOR statement can be expressed as an IF statement by setting the value of an index item and testing it as shown below.

```

ITEM AA I 60 U P 0 $
A1. IF AGE ($AA$) GR OLDST AND SEX EQ V(MALE) AND SALRY LS
      201.35 $
  OLDST = AGE ($AA$) $
  AA = AA + 1 $
  IF AA LQ 999 $
  GOTO A1 $

```

- The loop variable activated by a FOR statement does not need to be used in the loop statement. Rather, it may be used to determine the number of times the loop statement is iterated. Given a file OUT assigned to OUTPUT on which 10 lines of space are desired before printing the next output:

```

FOR Z = 0, 1, 9 $
  OUTPUT OUT 1H( ) $

```

These statements will cause OUTPUT OUT 1H() \$ to be iterated 10 times, printing one blank line each time.

PARALLEL FOR STATEMENTS

A parallel FOR statement consists of a string of FOR statements preceding and governing the execution of a single loop statement. The iteration is controlled by the first three-

component FOR statement, and the controlling statement must not be preceded by a two-component FOR statement. The three- and two-component FOR statements are modified from the last loop variable to the first.

Example:

- FOR A = 1 \$
 - FOR B = 0, 1, 9 \$
 - FOR C = 10, -2 \$
 - BEGIN
 - L1. XX(\$B\$) = C * 4 \$ This statement will be executed 10 times.
 - IF C LS A \$ TEST B \$
 - L2. YY = B \$ This statement will be executed 5 times.
 - END

NESTED FOR STATEMENTS

FOR statements can be nested, forming loops within loops, by enclosing FOR statements within BEGIN-END so that each FOR statement is separated from the next by a BEGIN. The limit for nested FOR statements is the maximum character set, 26.

Example:

- FOR A = 1 \$
 - BEGIN 'A' '
 - FOR B = 0, 1, 9 \$
 - BEGIN 'B' '
 - FOR C = 10, -2 \$
 - BEGIN 'C' '
 - L1. XX (\$B\$) = C * 4 \$ This statement will be executed 60 times.
 - IF C LS A \$ TEST B \$
 - L2. YY = B \$ This statement will be executed 50 times.
 - END 'C' '
 - END 'B' '
 - END 'A' '

A common use for nested loops is iteration through the elements of an array.

Example:

- Given array MATRX, set each element to zero where the current value is greater than 50 (the array elements are all integers).

```

ARRAY MATRX 3 4 3 I 48 U $
FOR A = 2, -1, 0 $
BEGIN 'A' '
  FOR B = 3, -1, 0 $
  BEGIN 'B' '
    FOR C = 2, -1, 0 $
    BEGIN 'C' '
      IF MATRX ($A, B, C$) GR 50 $
      MATRX ($A, B, C$) = 0 $
    END 'C' '
  END 'B' '
END 'A' '

```

This statement loops through columns in the array.

This statement loops through rows in the array.

This statement loops through planes in the array.

FOR ALL CLAUSE

The FOR ALL clause is a shorthand method of setting up a FOR clause based on the value of a table NENT. The format of the FOR ALL clause is:

```
FOR loop-variable = ALL ( (table-name
                          {table-item-name} ) ) $
```

While this clause is normally used to process all active entries in a table, it may also be used for other purposes. It has the same effect as the following:

- FOR loop variable = NENT ((table-name
{table-item-name})) -1, -1, 0 \$
- For rigid tables, the NENT is a constant, and the FOR ALL clause will iterate through all entries.
- For variable tables, the NENT is a variable and should be set to the number of active entries prior to a FOR ALL clause. For variable tables, a FOR ALL clause will iterate from one less than the number of active entries indicated by the NENT down through zero.

Example:

- Given the table MANPOWR defined on page 4-20, count the number of male employees and place the count in the item MEN.

```

MEN = 0 $
FOR A = ALL(MANPOWR) $
BEGIN
  IF SEX EQ V(MALE) $ MEN = MEN + 1 $
END

```

TEST STATEMENT

In a FOR statement, the compiler automatically supplies the steps that modify and test the loop variable and branches accordingly. The TEST statement provides a means to branch directly to the modify-test-branch process of a FOR statement. The TEST statement must appear within the loop statement following the FOR clause. The format is:

```
[name.] TEST [letter] $
```

where

letter (when used) = a particular loop variable specified in the FOR clause preceding the loop statement in which TEST appears

- If the letter is specified, control transfers to the modify-test-branch for that particular loop variable. All inner loops will be reinitialized because of this.
- If there is no letter specified, control is given to the modify-test-branch process of the innermost loop variable.

The TEST statement is applicable to parallel FOR statements as well as nested FOR statements.

Examples:

- ```
FOR A = 0, 1, NUM - 1 $
FOR B = 0, 1 $
BEGIN
 IF ACCT (A) NQ ACC'T
 (B) $
 TEST A $

 SHRT = SHRT + PAYMT
 (A) $

END
```

If the two items are unequal, test the outer loop variable by setting A = A+1 going to the statement after END if A is greater than NUM -1, returning to BEGIN if A is less than or equal to NUM -1.

Otherwise, increment SHRT and then test A by setting B=B+1, A = A +1 and returning to BEGIN if A is less than or equal to NUM -1, or exiting if A is greater than NUM -1.

- This example shows the programmer-supplied JOVIAL code and the implicit code generated by the JOVIAL compiler.

| <u>JOVIAL Code</u>    | <u>Implicit Code</u>       |
|-----------------------|----------------------------|
| FOR A = 0, 2, 100 \$  | A = 0 \$                   |
| FOR B = 50, 1 \$      | B = 50 \$                  |
| FOR C = 10 \$         | C = 10 \$                  |
| BEGIN 'ABC' '         | ABC.                       |
| FOR E = 100, -2, 0 \$ | E = 100 \$                 |
| FOR F = 5, 1 \$       | F = 5 \$                   |
| BEGIN 'EF' '          | EF.                        |
| •                     | •                          |
| •                     | •                          |
| •                     | •                          |
| TEST A \$             | GOTO ATEST \$              |
| •                     | •                          |
| •                     | •                          |
| •                     | •                          |
| TEST B \$             | GOTO BTEST \$              |
| TEST E \$             | GOTO ETEST \$              |
| TEST F \$             | GOTO FTEST \$              |
| TEST \$               | GOTO FTEST \$              |
| END 'EF' '            | FTEST. F = F+1 \$          |
|                       | ETEST. E = E-2 \$          |
|                       | IF E GQ 0 \$ GOTO EF \$    |
|                       | BTEST. B = B+1 \$          |
|                       | ATEST. A = A+2 \$          |
|                       | IF A LQ 100 \$ GOTO ABC \$ |

## PROGRAM CONTROL STATEMENT

The JOVIAL statement STOP provides a means of halting program execution and returning control to the operating system, or of pausing temporarily, then continuing execution of the program.

### STOP

The STOP statement halts program execution and transfers control to the operating system. The format is:

```
[name.] STOP [statement-label] $
```

If no STOP statement is included in the program, execution will be terminated and control returned to the operating system when control flows out at the end of the program.

If no statement label has been specified, the program will be halted at the STOP card. Control will be returned to the operating system. If a statement label has been specified, execution of the program will be suspended and the control point at which the job is executing will display PAUSE. This will continue until the control point is given a GO by the operator; the program will continue at the statement label specified in the STOP statement. The statement label is not displayed; thus, if more than one STOP statement and/or STOP statement label is included, it may not be possible to determine which caused the PAUSE or where execution will commence after a restart. Because this involves operator intervention, it should only be used for hands-on debugging.

Examples:

- AA1. STOP \$ Terminates the program execution.
- STOP LABX \$ The program is suspended; the control point at which the job is executing will display PAUSE. After a restart, the program will continue at LABX.

## DIRECT-JOVIAL STATEMENTS

Symbolic machine-language code can be inserted within a JOVIAL program if the code is placed within the DIRECT-JOVIAL brackets. The form is:

```
DIRECT
 •
 •
 •
 (assembler language code)
 •
 •
 •
JOVIAL
```

The assembler language code is a proper subset of the COMPASS assembly language. The allowable forms are given in Appendix L. The DIRECT-JOVIAL brackets provide a language facility for coding machine operations that cannot be expressed directly in JOVIAL. The facility is not intended to provide an unlimited escape into machine language, but rather to provide computational efficiency. Consequently, communication is restricted between a direct

code block and other parts of a JOVIAL program. In particular, no branches into or out of a direct code block are permitted; entrance and exit to the block are only at the top and bottom, respectively.

Three additional COMPASS pseudo-instructions are available with the assembly-type statements. These are the DIRECT, JOVIAL, and ASSIGN pseudo-instructions. The first two direct the transition from JOVIAL to assembly code and back again. ASSIGN allows direct code to communicate with JOVIAL variables.

### DIRECT-JOVIAL PSEUDO-INSTRUCTIONS

The DIRECT pseudo-instruction informs the JOVIAL compiler that all lines up to and including the JOVIAL pseudo-instruction line consist of direct code. The JOVIAL pseudo-instruction informs the direct-code processor of the compiler that this is the end of the direct-code sequence.

JOVIAL code resumes in column 1 of the next line, thus the remainder of a card containing the pseudo-instruction should be blank. The contents will not be used by the compiler, but printed in the listing as comments. The pseudo-instructions DIRECT and JOVIAL must appear in the operation field; that is, they must begin in a card column between 3 and 35.

JOVIAL statements cannot appear on the same line as COMPASS statements, therefore, the following form is illegal:

```
SB4 X3 JOVIAL
```

The JOVIAL statement must appear on the line following the last COMPASS statement.

### ASSIGN PSEUDO-INSTRUCTION

The only reference to a JOVIAL variable from inside the block of direct code is through the ASSIGN pseudo-instruction. ASSIGN allows loading from or storing into a simple nonindexed JOVIAL variable. The forms of ASSIGN are:

$$\text{ASSIGN variable} = A \left( \begin{array}{l} \text{X register} \\ \text{integer-constant} \end{array} \right) \$$$

or

$$\text{ASSIGN A} \left( \begin{array}{l} \text{X register} \\ \text{integer-constant} \end{array} \right) \$$$

The variable must be simple and not subscripted. If the X register or the integer constant are omitted, the assignment is to or from a floating-point register. Even if empty, the pair of parentheses following A must be specified. The use of ASSIGN is further described in Appendix L.

Various types of data, in the form of numbers, characters, or simply a string of contiguous bits, can be described and manipulated in the JOVIAL language. The data can be organized into tables of items, multidimensional arrays, or simple items. The size as well as the type of data forms may vary. This section presents a detailed description of these data forms.

### ITEM DECLARATIONS

An item is declared using the general form:

```
ITEM item-name item-description $
```

The item-name is supplied by the user and must conform to the general rules governing names. The item-description depends on the type of item. It can be one of six types:

- Integer
- Fixed-point
- Floating-point
- Literal (Hollerith or transmission code)
- Status
- Boolean

### INTEGER

The form of the integer item declaration is:

```
I n1 {U|S}{R}[n2 ... n3] [P n4] $
```

where

- I = an integer of arbitrary precision
- n1 = the number of bits required to contain the item including the sign bit if signed
- U = unsigned



- S = signed
- [R] = rounding specified
- [n2 ... n3] = range of integer value magnitudes the item contains from minimum (n2) to maximum (n3); n2 and n3 must be integer numbers.
- [P n4] = the item is preset to a numeric value of n4 (numeric constant).

Examples:

- ITEM INT I 60 S \$ Full word signed integer.
- ITEM UNS I 59 U \$ Largest unsigned integer which can be declared.
- ITEM SUM I 36 S P 400 \$ 36-bit signed integer (sign bit and 35 data bits). The variable is preset to 400.
- ITEM FIN I 18 S P 40.32 \$ An 18-bit signed item. The preset value is converted to integer; since the fractional part is truncated in the conversion to integer, the preset will be the same as if FIN had been preset to 40.
- ITEM ROUND I 48 S R \$ A 48-bit signed integer item. If the variable is set to a fixed-point or floating formula, it is rounded to an integer value instead of truncated.
- ITEM ROUND I 18 S 100 ... 100000 \$ ROUND is an 18-bit signed item. The range description indicates that a value less than 100 will not be held. This may enable the compiler to generate more efficient code for statements using the item.
- ITEM OCT I 28 S P O(5134) \$ Item OCT has a sign bit and 27 integer bits. The octal preset has a decimal value of 2652.

**FIXED-POINT**

The form of a fixed-point item declaration is:

A n1  $\begin{Bmatrix} U \\ S \end{Bmatrix}$  n2 [R] [n3 ... n4] [P n5] \$

where

A = a fixed-point value of the precision specified in n2.

- n1 = the number of bits required to contain the item including the sign bit if signed.
- U = unsigned
- S = signed
- n2 = the number indicating the position of the binary point from the right boundary of the item. If n2 is positive, it is the number of fractional bits; if negative, it is the number of missing bits. If n2 is omitted, the compiler treats the variable as an integer.
- [R] = rounding specified
- [n3 ... n4] = range of value magnitudes the item contains from minimum (n3) to maximum (n4).
- [P n5] = the item is preset to a numeric value of n5 (numeric constant).

Examples:

- ITEM FIX A 36 S 18 \$      FIX has one sign bit, 17 integer bits, and 18 fractional bits.
- ITEM FRACT A  
48 S 10 P 11.37 \$      A 48-bit signed item with a sign bit, 10 fractional bits, and 37 integer bits. FRACT is preset to the value 11.37.
- ITEM WORK A 24 U 9 \$      WORK is a 24-bit item with 15 integer bits and nine fractional bits. It is specified as unsigned; therefore, it has no sign bit.
- ITEM ALTITUDE A 10 U -6 \$      Item ALTITUDE has 10 integer bits. The negative fractional bit specification indicates that six bits are missing; that is, the decimal point is actually six bits to the right of the 10 bits. The least bit has a value of 64 ( $2^6$ ), the second 128 ( $2^7$ ), etc. Thus, the values which take 16 bits when fully represented can be represented in only 10 bits; however, the precision is now to the nearest multiple of 64 instead of the nearest multiple of one.
- ITEM RANGE A 8 U -2 \$      The item RANGE has eight integer bits. The -2 fractional bit indicates that two bits are missing. Thus, data values up to 1023 may be

- ITEM FIX'INT A 24 S 0 \$

represented in eight bits as the nearest multiple of 4; that is, the least bit of the variable has the value 4, the second 8, etc.

This 24-bit fixed-point item has no fractional bits specified; therefore, the resulting variable is the same as if integer 24 bits signed had been specified.

- ITEM FIXR A 48 S 3 R P  
234.5665 \$

The item FIXR has a sign bit, 44 integer bits, and 3 fractional bits. The R indicates rounding. If a fixed-point formula of greater precision than three fractional bits is assigned, the formula is rounded instead of truncated. In this example, the preset will result in FIXR having an initial setting of  $3525_8$  or  $234.625_{10}$ . The third fractional bit, which accounts for the difference, has a value of  $0.125_{10}$ . If rounding had not been specified, the preset would have been truncated to a value of  $3524_8$  or  $234.5_{10}$ .

- ITEM FRONT A 37 S 8 P  
O(1234) \$

Item FRONT has 37 bits, a sign bit, 28 integer bits, and eight fractional bits. The octal preset will be to the integer bits giving a value of  $668_{10}$  to the variable.

- ITEM LIMIT A 24 S 10  
-8000 ... -0.001 \$

The variable will have a sign bit, 13 integer bits, and 10 fractional bits. The information of the range, that the variable will not be set to 0 or a positive value, may enable more efficient code to be generated.

## FLOATING-POINT

The item description of a floating-point item has the form:

F [R] [P nl] \$

where

F = a floating-point item

$\left[ R \right]$  = rounding specified  
 $\left[ P \ n1 \right]$  = item is preset to initial value n1, a numeric constant

Examples:

- ITEM AMNT F R P 6.025\$      The floating-point item AMNT is rounded and preset to the floating value 6.025.
- ITEM XX F\$      The floating-point item XX is not rounded and has no preset value.

### LITERAL

The form of a literal item description is:

$\left\{ \begin{array}{l} H \\ T \end{array} \right\} \ n1 \ \left[ P \ n2 \right] \$$

where

- H = item containing display code characters
- T = item containing transmission code characters
- n1 = number of bytes allocated to item (six bits to a byte)
- $\left[ P \ n2 \right]$  = Item is preset to value n2, an octal, display, or transmission code constant

If a preset is not specified, literal items are preset to blanks of the item type, i. e. , Hollerith or STC blanks.

Examples:

- ITEM HOL H 15 \$      A 15-byte Hollerith item. Since no preset is specified, it will be set to Hollerith blanks.
- ITEM STC T 3 \$      A three-byte transmission code item. Since no preset is specified, it will be preset to transmission code blanks.
- ITEM HEADER H 10 P 10H(DATA TABLE) \$      The 10-byte Hollerith item HEADER is preset to DATA TABLE.
- ITEM NUMBER H 10 P O(171605) \$      The 10-byte Hollerith variable is preset to the Hollerith characters ONE. The result is right-justified with blank filled.

- ITEM LABEL  
T 5 P 5T(OPEN) \$      The five-byte variable is preset to the transmission code value for OPEN.
- ITEM MIXED H 3 P 3T(EFG) \$      MIXED is a three-byte Hollerith item. The transmission code preset is accepted but not converted to Hollerith; thus, the code generated is the same as if MIXED had been preset to 3H(JKL).

## STATUS

A status item defines symbolic values, which are essentially mnemonic labels that the item can assume. The format is:

S [n1] status-list [P n2] \$

where

- S = the item is a status item.
- [n1] = size indication (optional). If omitted, the size is determined by the number of values in the status list. The size is the number of bits required to contain the largest value the item may assume.
- status-list = the list of status constants whose value the status item can take; the value is composed of any JOVIAL letter or name. Status constants are separated by blanks. Status constant have the form V(status-constant).
- [P n2] = preset value. This value must be one of the status constants declared in the status list of the item.

Examples:

- ITEM GRADE S 6 V(PASS)  
V(FAIL) V(HONORS)\$      GRADE lists the possible states of the item. The size of the item is six bits; if the size specification is omitted, only two bits are required to contain the largest value, the number two.
- ITEM JUMP'CODE S  
V(AJM) V(EJM) V(EXN)  
V(FJM) V(IJM) V(LJM)  
V(MJN) V(NJM) V(PJN)  
V(RJM) V(UJN) V(ZJN)\$      This status item (JUMP'CODE) defines 12 states. The compiler assigns four bits to the item. The value assigned to V(AJM) is 0; to V(EJM), 1; and so forth incrementing by one through V(ZJN), which has the value of 11.

- ITEM VOWEL S V(A)  
 V(E) V(I) V(O) V(U) P V(A) \$
 

This status item defines five states. The compiler will assign three bits to the item. Since V(A) is the first status constant in the status-list, the initial value will be 0.
- ITEM DELIM S V(BEGIN) V(END)  
 V(START) V(TERM) V(DIRECT)  
 V(JOVIAL) V(IFFEITH)\$
 

In this example, the compiler assigns three bits to the item; it can then be used to represent the JOVIAL reserved words for delimiters.

## BOOLEAN

A Boolean item is used to express two alternatives. It is always a one-bit item whose value is either 0 or 1. When the value is equal to 0, it is false; when equal to 1, true.

The form of a Boolean item description is:

B [P n1]

where

B = identifies the item as a one-bit Boolean item  
 [P n1] = preset value (optional); n1 can be either 0 or 1.

Examples:

- ITEM PASS B\$
 

The item PASS is either true (1) or false (0).
- ITEM FORMULAB B P 0 \$
 

This item is preset to the value 0 (false). The value can change during execution.

## IMPLICIT ITEM DECLARATIONS

All of the preceding items (except Boolean and status) can be declared implicitly by substituting a constant of the item type for the item description. The size and precision of the item type will be determined by the constant. Integer and fixed-point items have only the minimum number of bits required to contain the preset constant. A sign bit is allocated only if the constant is signed. Implicit definition may not be used during execution of the program; integer or fixed-point items will be set to formulas requiring greater precision than the preset.

The form of an implicitly defined item is:

ITEM name constant \$

Examples:

- ITEM TITLE 14H(MANPOWER TABLE)\$ TITLE is a Hollerith literal constant requiring 14 six-bit bytes.
- ITEM NUMBER 10\$ NUMBER is an integer item whose size is the number of bits required to contain the unsigned integer number 10 (four bits).
- ITEM AAA 2. 5A25\$ The size of the fixed-point item AAA is determined by the value and must include 25 fractional bits.
- ITEM AMNT 6. 025\$ The floating value defines item AMNT as a floating-point item.

## MODE DECLARATIONS

Simple unsubscripted items that are not defined by an item declaration are automatically defined by a default mode of definition as signed integers occupying a full word, (1 60 S).

The user can change this default mode by the mode declaration. The mode declaration causes the compiler to assume a new default mode in accordance with the item description in the declaration. The format is:

MODE item-description [P constant] \$

The item description can be any of the forms shown in Table 5-1. If P and a constant are specified, all subsequent mode-defined items are preset to the constant value.

The mode declaration may occur anywhere among the statements or declarations of a program, within the main program, or in a procedure or function declaration.

A mode declaration takes effect when encountered and remains in effect until the next mode declaration or the end of the program.

The compiler does not resolve undefined items until all source cards have been read. Thus, if an item-declaration does not occur in the program until after it is used in a statement, it receives the description specified in the item-description, not the current mode value at the time it is first used.

Mode definition may not take place in an overlay statement. The variables used must either be declared in the program or mode-defined previously. Mode definition may not be used for the formal parameters in procedures.

TABLE 5-1. ITEM DESCRIPTIONS

| Item Type                                       | Item Description <sup>†</sup> |                                            |                |                        |         |                                       |                                                       | Preset Descriptor                            |            |
|-------------------------------------------------|-------------------------------|--------------------------------------------|----------------|------------------------|---------|---------------------------------------|-------------------------------------------------------|----------------------------------------------|------------|
|                                                 | Type Code                     | Size                                       | Signed         | Fractional Bits        | Rounded | Specific Values                       | Range <sup>††</sup>                                   | Preset <sup>†††</sup> Values                 | Terminator |
| Integer                                         | { I }<br>{ A }                | # bits, <sup>††††</sup><br>integer<br>≤ 60 | { U }<br>{ S } | -                      | [ R ]   | -                                     | [ integer<br>constant<br>...<br>integer<br>constant ] | [ P<br>Numeric<br>constant ] <sup>††††</sup> | \$         |
| Fixed-Point<br>(arithmetic)                     | A                             | # bits, <sup>††††</sup><br>integer<br>≤ 60 | { U }<br>{ S } | [+]<br>integer<br>≤ 59 | [ R ]   | -                                     | [ numeric<br>constant<br>...<br>numeric<br>constant ] | [ P<br>Numeric<br>constant ] <sup>††††</sup> | \$         |
| Floating,<br>single<br>precision                | F                             | -                                          | -              | -                      | [ R ]   | -                                     | -                                                     | [ P<br>Numeric<br>constant ]                 | \$         |
| Literal<br>(display or<br>transmission<br>code) | { H }<br>{ T }                | # chars,<br>integer<br>≤ 250               | -              | -                      | -       | -                                     | -                                                     | [ P literal<br>constant ]                    | \$         |
| Status                                          | S                             | [ # bits,<br>integer<br>≤ 59 ]             | -              | -                      | -       | one or<br>more<br>status<br>constants | -                                                     | [ P status<br>constant ]                     | \$         |
| Boolean                                         | B                             | -                                          | -              | -                      | -       | -                                     | -                                                     | [ P Boolean<br>constant ]                    | \$         |

<sup>†</sup> A dash indicates that the entry is not applicable.

<sup>††</sup> The range states the estimated minimum through maximum (absolute value) that is likely to be assigned to the variable. The constants must be positive or zero, and the smaller must come first. The compiler uses the range estimation for scaling computations in numeric formulas.

<sup>†††</sup> The specification of a preset value gives the item an initial value.

<sup>††††</sup> The number of magnitude bits, excluding sign, must be at least 1 and not more than 59. A signed item size must be at least 2; an unsigned item size must be less than 60.

Example:

```
START$
ITEM AA I 18 S $
ITEM BB B P 0 $
ITEM FF F $
```

```
•
•
AA=25 $
CC=AA + 15 $
•
•
```

Start of program.

Items declared in normal manner.

Since CC is not declared in the program and no COMPOOL has been used, it is mode defined as I 60 S.



|                                 |                                                                                                                                                                                                                                                          |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MODE F \$                       | Effective with this line of code, the mode definition is now floating-point.                                                                                                                                                                             |
| •                               |                                                                                                                                                                                                                                                          |
| SUM=FF - CC \$                  | Since SUM is not defined in the program, it is given the mode definition of a floating-point variable.                                                                                                                                                   |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| TIME=2*SUM \$                   | SUM has been previously mode defined as floating-point. Although TIME has not been defined previously in the program, a declaration occurs further on in the program. Thus the integer definition, not the current mode, will be used in this statement. |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| MODE I 14 S P -30\$             | Effective with this statement, the mode definition changes to a 14-bit signed integer with a preset value of -30.                                                                                                                                        |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| AA = CC/CONST \$                | Since CONST has not been defined in the program, the mode value of a 14-bit signed integer with value -30 is used.                                                                                                                                       |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| MODE H 15 \$                    | Effective with this statement, the mode definition changes to 15 Hollerith.                                                                                                                                                                              |
| •                               |                                                                                                                                                                                                                                                          |
| ITEM TIME I 60 S \$             | Normal item-declaration which occurs after it is used in the program.                                                                                                                                                                                    |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| TITLE=14H<br>(JOVIAL PROGRAM)\$ | Since TITLE has not been declared in the program, it is mode defined as 15 Hollerith bytes. This statement justifies the 14-byte header in it.                                                                                                           |
| •                               |                                                                                                                                                                                                                                                          |
| •                               |                                                                                                                                                                                                                                                          |
| TERM \$                         |                                                                                                                                                                                                                                                          |

## ARRAYS

An array is an arrangement of item-like elements in one or more dimensions. A particular element of an array is designated by an index having as many components as there are dimensions in the array. An array can have a maximum of seven dimensions.

A simple type of array is the two-dimension array or matrix in which each element is located by row and column.

| Row | Column |   |    |    |
|-----|--------|---|----|----|
|     | 0      | 1 | 2  | 3  |
| 0   | 1      | 3 | -7 | 0  |
| 1   | -4     | 6 | 36 | 19 |
| 2   | 12     | 5 | 10 | 11 |

In this array the value 36 is in Row 1, Column 2. The number of dimensions of the array is two, and the dimensions are three by four.

Each element of an array is referenced by the array name followed by an index which has as many components as there are dimensions in the array. For instance, the two dimensional array described above has a two-component index. The number 36 in the array is designated by the index 1, 2. The total size of an array is limited to  $2^{17}-1$  words.

### ARRAY DECLARATION

The format of an ARRAY declaration is:

```
ARRAY array-name dimension-list item-description $
 [constant-list]
```

where:

- array-name = User-supplied name. The name by itself is used to refer to the entire array. To specify a particular element off the array, the array-name must be followed by indexes. The number of indexes must be equal to the number of constants in the dimension list.
- dimension-list = One or more unsigned integer constants which specify the number of dimensions and the size of each dimension. The list can be up to seven integers; the first is the number of rows; the second the number of columns; the third the number of planes; the fourth is the number of volumes, and so forth.
- item-description = The array type, which may be any of the legal descriptions of a single item shown in Table 5-1. Each element of the array has the properties of the array type; the declaration may not contain a preset.
- [constant-list] = An optional constant list following the array declaration is used to preset all or part of the array with initial values. The constant list has the same number of dimensions as the array to which the list applies. The constants in the list must agree in type with the item description of the array. The valid presets are the same as for simple items (see Table 5-1).

Examples:

- ARRAY ONE 60 F \$ This array has one dimension. Each element is a floating-point item.
- ARRAY MATRIX 3 4 I 60 S \$ This array has two dimensions of three rows and four columns. Each element is a 60 bit signed integer.
- ARRAY NOTE  
4 4 4 A 60 S 15 R\$ Array NOTE has three dimensions with four rows, four columns, and four planes. Each element is a 60-bit fixed-point signed item with 15 fractional bits. The values in the array will be rounded.
- ARRAY BOOL 100 100 B\$ This Boolean array has two dimensions of 100 rows and 100 columns. Each element will be one bit; the array is packed by bit, not by word.
- ARRAY LONG'LIT  
20 20 H 25\$ This literal array has two dimensions of 20 rows and 20 columns. Each element will be three consecutive words; thus, the array will take up  $20 \times 20 \times 3$  or  $1200_{10}$  ( $2260_8$ ) words of storage.
- ARRAY SEVEN  
2 2 2 3 4 5 2 I 18 S \$ This integer array has the maximum number of dimensions, seven. Each element is an integer 18-bit signed item.

### CONSTANT LIST

The constant list is used to preset all or part of the array with initial values. The constant list for each dimension is enclosed in separate levels of BEGIN-END brackets. The number of BEGIN-END brackets must equal the number of dimensions specified in the dimension list. The following rules apply to the use of brackets in the constant list:

- If fewer constants are specified, then only the first elements in the dimension are set. Null presets are not defined; therefore, all elements in the level being preset up to the one desired, must be presets.
- The innermost BEGIN-END brackets enclose rows.
- The second level of BEGIN-END brackets encompass columns of a plane. The third and subsequent levels of BEGIN-END brackets include planes of a volume.

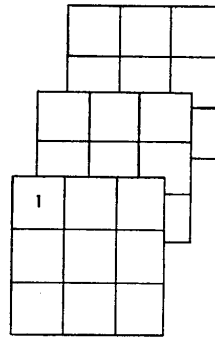
- No more than the first 5800 words of an array can be preset.
- There may be no more than 25 preset arrays in one program.

The order of the two innermost BEGIN-END brackets is a reversal of the allocation by column, row, plane. It is a compromise that allows the declaration of constant lists to reflect the visual order of an array. The items in an array are indexed in allocation order.

Examples:

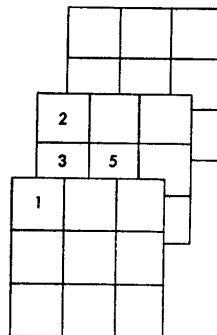
- ```
BEGIN
  BEGIN BEGIN 1 END END
  END
```

The constant list is preset to a value of 1. The first element of the first column in the three by three by three dimensional array with preset values is shown below.



- ```
BEGIN
 BEGIN BEGIN 1 END END
 BEGIN
 BEGIN 2 END
 BEGIN 3 5 END
 END
 END
```

To specify the middle element as five, it is necessary to specify other elements as shown in this example.

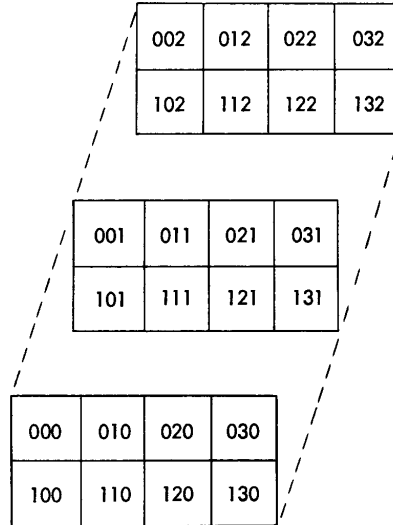


- `ARRAY XX 2 4 3 1 15 U$`      The preset in this example sets each element to a value composed of its concatenated index components. Only the innermost two dimensions are inverted (see below).
 

```

 BEGIN
 BEGIN
 BEGIN 000 010 020 030 END
 BEGIN 100 110 120 130 END
 END
 BEGIN
 BEGIN 001 011 021 031 END
 BEGIN 101 111 121 131 END
 END
 BEGIN
 BEGIN 002 012 022 032 END
 BEGIN 102 112 122 132 END
 END
 END
 END

```

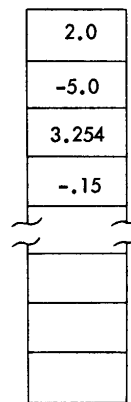


- `ARRAY FLOAT 10 F $`      A one-dimensional floating-point array has the first four elements set to the constants in the constant list; the remaining six elements are not preset (see below).
 

```

 BEGIN 2. -5 3.254-.15 END

```



- ARRAY COMPASS 3 3 H 2 \$  
 BEGIN  
 BEGIN 2H(NW) 2H(N ) 2H(NE) END  
 BEGIN 2H(W ) 2H( ) 2H(E ) END  
 BEGIN 2H(SW) 2H(S ) 2H(SE) END  
 END

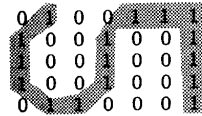
This two-dimensional array, COMPASS, is set to display code values. Each value consists of two characters (see below).

|        |        |        |
|--------|--------|--------|
| 2H(NW) | 2H(N ) | 2H(NE) |
| 2H(W ) | 2H( )  | 2H(E ) |
| 2H(SW) | 2H(S ) | 2H(SE) |

- ARRAY ONE 4 8 2 I 1 U \$  
 BEGIN  
 BEGIN 1 0 0 1 0 0 0 1 END  
 BEGIN 0 1 1 0 1 1 1 0 END  
 BEGIN 1 0 0 1 1 1 0 0 END  
 BEGIN 1 1 0 0 1 1 0 0 END  
 END  
 BEGIN  
 BEGIN 0 0 1 1 0 0 1 1 END  
 BEGIN 0 0 0 0 0 1 0 0 END  
 BEGIN 1 1 1 1 1 1 1 1 END  
 BEGIN 1 1 0 0 0 0 0 1 END  
 END  
 END

This array has a three dimensional constant list which sets each element to either one or zero.

- ARRAY TWO 5 7 B \$  
 BEGIN  
 BEGIN 0 0 0 1 1 1 END  
 BEGIN 0 0 0 0 0 1 END  
 BEGIN 0 0 1 0 0 1 END  
 BEGIN 0 0 1 0 0 1 END  
 BEGIN 0 0 0 0 0 1 END  
 BEGIN 0 0 0 0 0 1 END  
 END



This array is preset to represent the character 2. The preset specification is not in allocation order; the inverted picture of the numeral two is obtained when the ones are connected in the declaration. The indexes in allocation order are:

|                     |                     |
|---------------------|---------------------|
| TWO(\$ 0, 0 \$) = 0 | TWO(\$ 2, 2 \$) = 0 |
| TWO(\$ 1, 0 \$) = 1 | TWO(\$ 3, 2 \$) = 0 |
| TWO(\$ 2, 0 \$) = 1 | TWO(\$ 4, 2 \$) = 1 |
| TWO(\$ 3, 0 \$) = 1 | TWO(\$ 0, 3 \$) = 0 |
| TWO(\$ 4, 0 \$) = 0 | TWO(\$ 1, 3 \$) = 1 |
| TWO(\$ 0, 1 \$) = 1 | TWO(\$ 2, 3 \$) = 1 |
| TWO(\$ 1, 1 \$) = 0 | TWO(\$ 3, 3 \$) = 1 |
| TWO(\$ 2, 1 \$) = 0 | TWO(\$ 4, 3 \$) = 0 |
| TWO(\$ 3, 1 \$) = 0 | TWO(\$ 0, 4 \$) = 1 |
| TWO(\$ 4, 1 \$) = 1 | TWO(\$ 1, 4 \$) = 0 |
| TWO(\$ 0, 2 \$) = 0 | TWO(\$ 2, 4 \$) = 0 |
| TWO(\$ 1, 2 \$) = 0 |                     |

|                     |  |                     |
|---------------------|--|---------------------|
| TWO(\$ 3, 4 \$) = 0 |  | TWO(\$ 0, 6 \$) = 1 |
| TWO(\$ 4, 4 \$) = 0 |  | TWO(\$ 1, 6 \$) = 1 |
| TWO(\$ 0, 5 \$) = 1 |  | TWO(\$ 2, 6 \$) = 1 |
| TWO(\$ 1, 5 \$) = 0 |  | TWO(\$ 3, 6 \$) = 1 |
| TWO(\$ 2, 5 \$) = 0 |  | TWO(\$ 4, 6 \$) = 1 |
| TWO(\$ 3, 5 \$) = 0 |  |                     |
| TWO(\$ 4, 5 \$) = 0 |  |                     |

- ARRAY SEVEN 3 5 2 4 5 4 3 I 18 S \$ The constant list presets the first row of the first plane of the seven dimensional array. This array example takes up 7200 words of storage; the last 1400 words cannot be preset since the 5800 word limit applies.
 

```

 BEGIN
 BEGIN
 BEGIN
 BEGIN
 BEGIN
 BEGIN 3 4 50 80 -4500 END
 END
 END
 END
 END
 END
 END
 END

```

### REFERENCING ARRAYS

An array may be referenced in two ways:

- By the array name only. The name only is used to reference an array that is used as a parameter to a procedure or function or used in an INPUT or OUTPUT statement to transfer an entire array from or to a binary file.
- By the array name followed by an index. One component for each dimension of the array is used in the index when an element of the array is specified.

The components which may be used as indexes are:

- Any numeric formula. If the formula does not yield an integer value, the formula is truncated to integer. The numeric formula may be a subscripted value or several subscripted values. The maximum subscript nesting level is 20.
- A formula that begins with 0 and runs to a maximum of n-1 for a dimension with n elements. If the formula yields a result which does not lie between 0 and n-1, it will be performed with unspecified results.

Examples:

- INPUT FILE'A MATRIX \$ Reads a record from binary file FILE'A into array MATRIX.

- `MAT'MULT (ARRY'A, ARRY'B  
= ARRY'C) $` Two arrays are passed to procedure  
MAT'MULT and ARRY'C is an output param-  
eter. The procedure MAT'MULT requires two  
arrays as input parameters and one array as  
an output parameter. The actual array param-  
eters in this statement must agree with the  
type defined as formal parameters.
- `FLOAT($2$)` Refers to the third element of array FLOAT.
- `XX($0, 1, 2$)` Refers to the element in the first row, second  
column, and third plane of array XX.
- `XX($VBL, FLOAT($4$),  
FUNCT(30) $)` In this reference to array XX, the element  
has the value of the numeric variable VBL for  
its first index, the value of the fifth element of  
array FLOAT for its second index, and the  
result of the numeric function FUNCT acting  
on 30 as its third index. As with the evaluation  
of `FLOAT ($4$)` for the second index, the for-  
mula will be truncated to integer if an integer  
value is not given.
- `FLOAT($ XX($1, 1$)  
+ 3*ZZZ - 20 $)` The element referenced by this formula is  
determined by the results of the evaluation of  
the expression given as the index. The formula  
includes another array item and the numeric  
variable ZZZ.

## TABLES

Data which is logically related but not identical in structure can be grouped in tables. Tables also allow packing of data for maximum utilization of storage. A table might contain the following fields:

- Twelve bytes for the employee name
- Nine bytes for his social security number



- Twelve integer bits, unsigned, for salary
- Fifteen integer bits, unsigned, for earnings to date
- Thirteen integer bits, unsigned, for income tax to date
- Thirteen bits in fixed-point for social security to date
- One-bit, Boolean, for marital status

Although the items contained in the fields listed above differ in size and structure, they can be logically grouped into one table entry. The table will contain one entry per employee, each entry having an identical structure. If the table is loosely-packed, eight words per entry (the same as if each item were a one-dimensional array) are required; by selected packing, each entry could be reduced to three words. The significant saving in storage may more than justify the slight increase in access time caused by the packing of the data. The ENTRY functional modifier moves the logically related data items in a table entry by a single statement.

#### **TYPES OF TABLES**

Tables can be either fixed-length (with a set number of active entries) or variable-length (with only the maximum number of entries specified).

The three main types of tables and table declarations are:

- Ordinary – Table whose allocation is set by the compiler
- Defined – Table with allocation specified by the user
- Like – Table which is a copy of a similarly structured table

A table declaration can be rigid or variable in the number of active entries and serial or parallel in structure.

#### **TABLE STRUCTURE**

Tables are similar in structure to one-dimensional arrays; however, tables may contain entries composed of several different data items whereas arrays can contain only elements of like type. In all tables, the word prior to the first data word is reserved by the compiler to contain the number of active entries in the table (LOC(TAB)-1). This control word is known as the NENT of the table.

A table entry may contain one or more items with identical substructures. Both entries and items can be referenced. An entry is referenced by applying the ENTRY modifier to the table name with an entry index; the items are referenced by indexing the item-name with an entry index. Because a table is a one-dimensional arrangement of entries, the entry and the table item indexes have only one component.

In a serial table, storage is allocated one entry at a time so that each item in an entry is allocated consecutive storage. In a parallel table, storage is allocated with the first word of each entry forming the first block of storage, the second word of each forming the second block of storage, etc.

The programmer can specify that the arrangement of a table be either parallel (P) or serial (S). If neither P nor S are specified, P is the default structure.

Example:

```
• TABLE AA0 R 4 P D $
 BEGIN
 ITEM A1 B $
 ITEM B1 I 10 U $
 ITEM C1 A 49 S 10 $
 ITEM D1 H 20 $
 ITEM E1 F $
 END
```

In this TABLE declaration, parallel structure is specified. See Figure 5-1 for the parallel data structure created in the example. If the declaration had specified serial structure the structure shown in Figure 5-2 would have been created.

If a table is to be read or written in blocks or is referenced by entry, a serial table is recommended because the entire entry is adjacent in storage. Otherwise, a parallel table is preferred because any single item can be referenced by a single index; thus, the compiler need not calculate relative subscripts for each item referenced. For tables in which there is only one word per entry, either type of table can be used with equal effect.

#### TABLE SIZE

The space allocated for a table is fixed at compilation time according to the number of entries specified in the user's table declaration. In an ordinary table, the compiler determines the size of entries for a table; in a defined table, the user must specify the size. The total storage required for a given table equals the number of words required by an entry multiplied by the number of entries plus one word for the NENT. Two methods of keeping track of the space are provided: R for rigid-length tables and V for variable-length tables.

- For rigid tables, all entries are assumed to be active and the NENT field is an integer constant which is generated at compilation time.

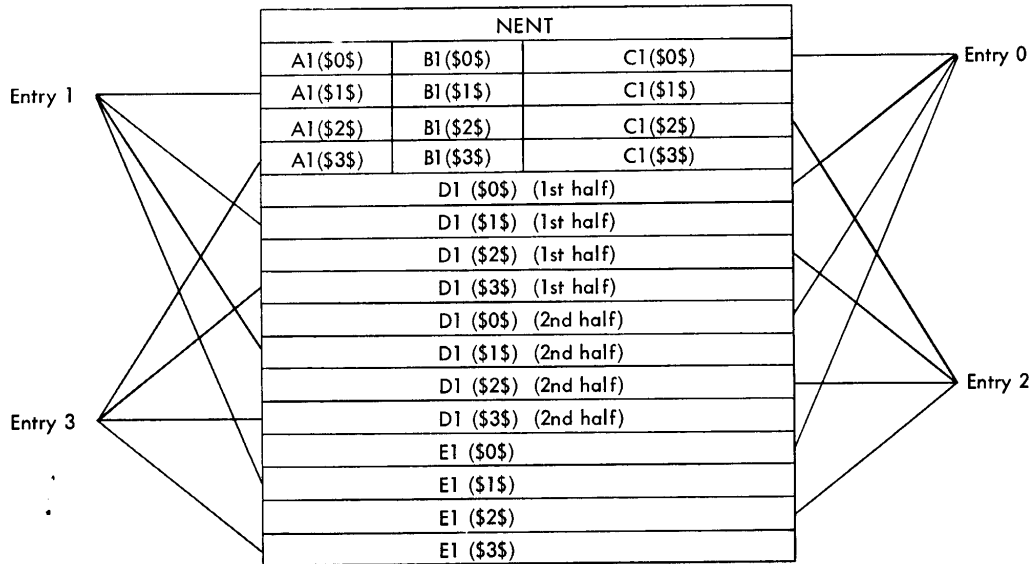


Figure 5-1. Parallel Table Structure

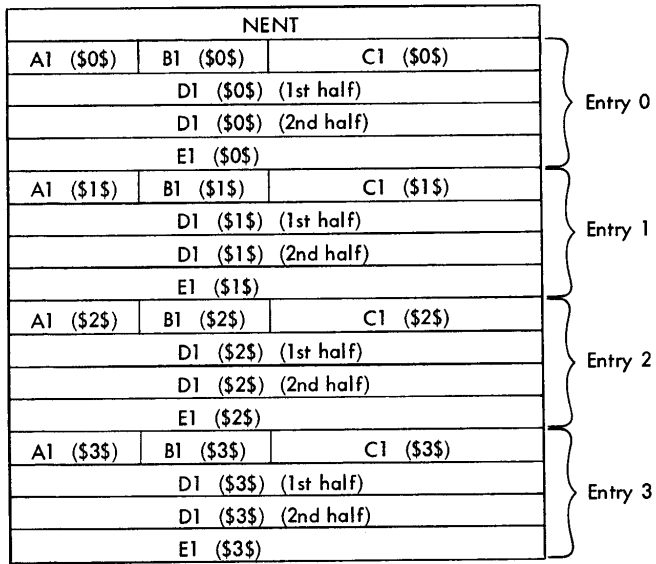


Figure 5-2. Serial Table Structure

- For a variable table, the number of active entries is assumed to vary from 0 to the maximum specified in the table declaration. The NENT of a variable table is an integer variable and is an undefined value until set by the user during execution of the program; the range of NENT is from entry 0 to the maximum number of active entries specified in the table declaration.

### TABLE PACKING

Table packing is the allocation of data items within an entry. Two of the three methods available for packing are performed automatically by the compiler and are influenced by the use of subordinate overlay statements. The third method requires the programmer to specify the placement of each item within the entry. The methods are:

- No packing – This is the default option in an ordinary table or may be specified by using N for the packing descriptor.
- Dense packing – This is specified by use of the M<sup>†</sup> or D packing descriptor in the table declaration.
- Programmer packing – The programmer must specify the number of words per entry in the table declaration for use in a defined table.

No packing provides faster access to table items, but requires more storage than dense packing. Storage for table entries with less than full word items can be greatly reduced by use of dense-packing or programmer packing methods in a defined table, but only at the expense of access time.

### NO PACKING

No packing means that the compiler will allocate space the same as would be allocated for a simple item. That is, each item takes up the least number of full words required to contain it. The amount of data in a table with no packing is the same as the storage required when each table item is declared as a one-dimensional array.

Example:

- The table described on page 5-17 specified with no packing, will have eight words per entry or a total of 800 words for the entire table. The allocation has at least one item per word. Items less than one word in length are right-justified, and literals larger than one word are left-justified.

---

<sup>†</sup>Medium packing (M) has not been implemented in the 6000 computer. If M is specified, it will default to dense packing.

```

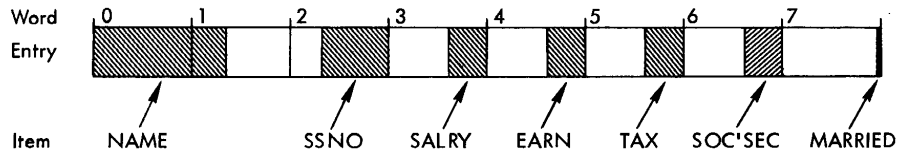
START $
TABLE PERSON R 100 S $
BEGIN
 ITEM NAME H 12 $
 ITEM SSNO H 9 $
 ITEM SALRY I 12 U $
 ITEM EARN I 15 U $
 ITEM TAX I 13 U $
 ITEM SOC'SEC A 13 U 7 $
 ITEM MARRIED B $
END
TERM $

```

The storage map generated by the compiler allocates the following entry:

| <u>Item</u> | <u>Word</u> | <u>Bit</u> | <u>Length</u> |
|-------------|-------------|------------|---------------|
| NAME        | 0           | 0          | 12 bytes      |
| SSNO        | 2           | 6          | 9 bytes       |
| SALRY       | 3           | 48         | 12 bits       |
| EARN        | 4           | 45         | 15 bits       |
| TAX         | 5           | 47         | 13 bits       |
| SOC'SEC     | 6           | 47         | 13 bits       |
| MARRIED     | 7           | 59         | 1 bit         |

This allocation uses eight words (480 bits) to contain a total of 180 bits of data, leaving 62.5% of each entry unused. The data structure appears as follows:



### DENSE PACKING

In dense packing, the compiler will attempt to allocate the entry in the minimum number of words. However, only a literal item longer than 10 bytes may cross a word boundary and, if it does, it must start on the byte boundary.

Example:

The table entry described on page 5-17 specified with dense packing will have four words per entry or a total of 400 words for the table. This requires only half the space used in no packing. Dense packing places several items in the same word and requires additional shifting to access the items.

```

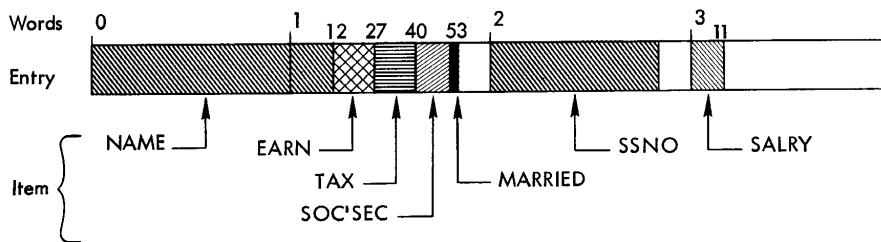
START $
TABLE PERSON R 100 S D $
BEGIN
 ITEM NAME H 12 $
 ITEM SSNO H 9 $
 ITEM SALRY I 12 U $
 ITEM EARN I 15 U $
 ITEM TAX I 13 U $
 ITEM SOC'SEC A 13 U 7 $
 ITEM MARRIED B $
END
TERM $

```

The storage map generated by the compiler allocates the following entry:

| Item    | Word | Bit | Length   |
|---------|------|-----|----------|
| NAME    | 0    | 0   | 12 bytes |
| EARN    | 1    | 12  | 15 bits  |
| TAX     | 1    | 27  | 13 bits  |
| SOC'SEC | 1    | 40  | 13 bits  |
| MARRIED | 1    | 53  | 1 bit    |
| SSNO    | 2    | 0   | 9 bytes  |
| SALRY   | 3    | 0   | 12 bits  |

This allocation uses four words (240 bits) to contain a total of 180 bits of data, leaving only 25% of the entry unused. The data structure appears as follows:



### PROGRAMMER PACKING

A table with programmer packing (also known as defined packing) will have the number of words per entry specified by the programmer. Each item declaration within the table must specify the word and bit position at which the entry will begin. Only literal items may cross word boundaries; if they do, they must start at a byte boundary. There is no restriction on the minimum length of literals. In other words, an option to cross word boundaries with literals of less than one word is available. However, the use of partial word literals is very expensive; two words must be fetched, shifted, and combined to form the desired variable.

Example:

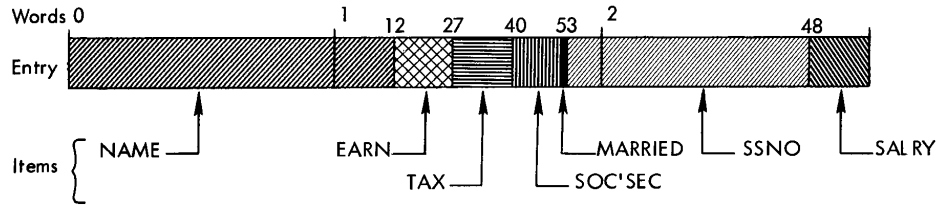
The table described on page 5-17 specified with defined packing will have three words per entry or a total of 300 words for the entire table. As a result, the items given in the example can be packed in three-eighths the space of a no-packed table and three-fourths the space of a dense-packed table.

```
START$
TABLE PERSON R 100 S 3 $
 BEGIN
 ITEM SSNO H 9 1 54 $
 ITEM NAME H 12 0 00 $
 ITEM SALRY I 12 U 2 48 $
 ITEM EARN I 15 U 1 12 $
 ITEM TAX I 13 U 1 27 $
 ITEM SOC'SEC A 13 U 7 1 40 $
 ITEM MARRIED B 1 53 $
 END
TERM $
```

The storage map generated by the compiler allocates the following entry:

| <u>Item</u> | <u>Word</u> | <u>Bit</u> | <u>Length</u> |
|-------------|-------------|------------|---------------|
| NAME        | 0           | 0          | 12 bytes      |
| EARN        | 1           | 12         | 15 bits       |
| TAX         | 1           | 27         | 13 bits       |
| SOC'SEC     | 1           | 40         | 13 bits       |
| MARRIED     | 1           | 53         | 1 bit         |
| SSNO        | 1           | 54         | 9 bytes       |
| SALRY       | 2           | 48         | 12 bits       |

In this example, the data structure is identical to the one for dense packing except for the allocation of items SSNO and SALRY. The entire entry is used for data storage. The data structure appears as follows:



### OVERLAY UTILIZATION

In addition to packing, the user can overlay items within the same table. An overlay within a table declaration is called a subordinate overlay. The following restrictions apply to subordinate overlays:

- Only items declared in the table prior to the overlay statement can be referenced.
- Subject to the previous restriction, overlay declarations may be placed anywhere an item declaration can be placed.
- The overlay can be used to allocate different items to the same storage space or to specify the order of placement within an entry for the items in the overlay statement.
- For tables with no packing, the items are overlaid on a full word basis.
- For dense-packed tables, the overlay specifies the order of dense packing and the overlay is based on the actual bit length of the specified items.
- The subordinate overlay cannot be used with defined tables because they specify all item positions explicitly.

Except for these restrictions, subordinate overlays are the same as detailed in the general OVERLAY Declaration described on page 5-41.

### TABLE DECLARATIONS

Tables are declared in two parts:

1. A declaration of the table as a whole in a table header declaration
2. A set of item descriptions enclosed in BEGIN-END brackets forming a table entry declaration.



## ABBREVIATED TABLE DECLARATIONS

The format for an abbreviated TABLE declaration is:

```
TABLE [name] {R} number $
 {V}
 BEGIN item-descriptions END
```

where:

name = programmer-supplied name  
R = rigid table length  
V = variable table length, the number of active entries is a variable  
number = maximum number of entries, an integer constant  
item-descriptions = descriptions for the items comprising an entry

The abbreviated table declaration described above is a minimal declaration in that all optional fields except for the table name has been omitted.

The three types of table declarations are:

- Ordinary table declarations in which the arrangement of data is left to the compiler.
- Defined table declaration in which the programmer specifies how the data of an entry is to be arranged.
- Like table declaration which is a shorthand method of declaring a table to have an entry structure similar to another ordinary or defined table.

For ordinary and defined tables, naming is optional; like tables must be named. Packing may be specified only for ordinary and like tables. Following the table declarations are the item declarations for the items within a table entry. The types of table item declarations correspond to the table declarations. Every table must have at least one item.

## TABLE HEADER DECLARATIONS

Headers for the three types of TABLE declarations are:

- Ordinary table header:

```
TABLE [name] {V} number1 [P] [N] $
 {R} [S] [M]
 [D]
```

- Defined table header:

$$\text{TABLE } [\text{name}] \left\{ \begin{array}{c} \text{V} \\ \text{R} \end{array} \right\} \text{ number}_1 \left[ \begin{array}{c} \text{P} \\ \text{S} \end{array} \right] \text{ number}_2 \$$$

- Like table header, that is, like table declaration:

$$\text{TABLE name } \left\{ \begin{array}{c} \text{V} \\ \text{R} \end{array} \right\} \text{ number}_1 \left[ \begin{array}{c} \text{P} \\ \text{S} \end{array} \right] \left[ \begin{array}{c} \text{N} \\ \text{M} \\ \text{D} \end{array} \right] \text{L} \$$$

### ORDINARY TABLE DECLARATIONS

In an ordinary table, the compiler allocates the table items one to a word for items of one word or less; literal items longer than 10 bytes are allocated to the next full word. The bit starting position within a word is also allocated by the compiler. The user must provide a size specification of the maximum number of entries; naming and packing are optional. The user must specify whether the table structure is variable or rigid. The format of an ordinary table declaration is:

$$\text{TABLE } [\text{name}] \left\{ \begin{array}{c} \text{V} \\ \text{R} \end{array} \right\} \text{ number}_1 \left[ \begin{array}{c} \text{P} \\ \text{S} \end{array} \right] \left[ \begin{array}{c} \text{N} \\ \text{M} \\ \text{D} \end{array} \right] \$$$

BEGIN

(one or more) ITEM item-name item-description \$ [one-dimensional-constant-list] [OVERLAY declaration(s)]

END

where:

- |                     |                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| name                | = table identifier, optional except when used as the pattern table for a like declaration.                                                                                                                                                                                                                                                                   |
| V or R              | = variable or rigid                                                                                                                                                                                                                                                                                                                                          |
| number <sub>1</sub> | = an unsigned integer specifying either the maximum number of entries in a variable-length table or the exact number in a rigid-length table. For a rigid-length table, the NENT is preset by the compiler to the specified number of entries. For a variable-length table, the programmer sets NENT to the number of entries currently active in the table. |
| P or S              | = parallel or serial structure. If both letters are omitted, parallel structure is assumed.                                                                                                                                                                                                                                                                  |

- N or M or D = no packing, medium, or dense packing. If an option is not selected, the default is no packing. The compiler assigns each item to one or more words so that no two items share the same word. The compiler keeps track of the location of the items and the number of words per entry.
- item-name = item identifier, as with a simple item
- item-description = characteristics of item, as with a simple item except that no preset may be present.
- one-dimensional-constant-list = table items may be set to initial values with a one-dimensional constant list, as in a one-dimensional array. The first constant in the list presets the item in the first entry, the second constant presets the item in the second entry, and so forth. The presets must agree in type and size. There is no provision for skipping items in the middle of a preset list.
- OVERLAY declarations = the rules governing the use of overlay statements in tables are described on page 5-25

Examples:

- TABLE AA0 R 4 P D \$  
 BEGIN  
 ITEM A1 I 10 U \$  
 ITEM B1 B \$  
 ITEM C1 A 49 S 10 \$  
 ITEM D1 H 20 \$  
 END

The table structure shown in the illustration (page 5-20) of parallel structure is defined. Note that the packing is dense, the length is rigid consisting of four entries, and P is specified though it could be omitted and the structure would still be parallel.
- TABLE AA0 R 4 S D \$

(Item declarations are the same as in the previous example.)

The table shown in the illustration (page 5-20) of serial structure has the same set of item descriptions, but S must be specified in the table declaration.
- TABLE AA0 R 4 P D \$  
 BEGIN  
 ITEM A1 I 10 U \$ BEGIN 100 200  
                                   300 400 END  
 ITEM B1 B \$ BEGIN 1 1 1 1 END  
 ITEM C1 A 49 S 10 \$ BEGIN  
                                   1000.00 2000.00  
                                   3000.00 4000.00  
                                   END  
 ITEM D1 H 20 \$  
 END

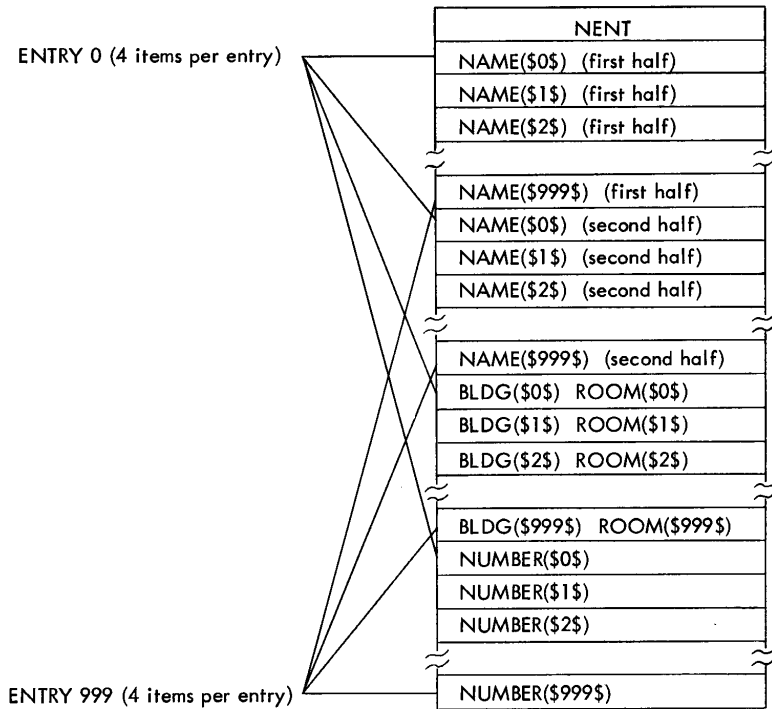
The table in this example is preset to initial values (by a constant list) except for the display code item.

- TABLE DIRECTORY V 1000 M \$ The following table contains phone directory data. The table is parallel by default with dense packing specified. It is variable-length with a maximum of 1000 entries, each entry containing four items. The data structure created is shown below.
 

```

 BEGIN
 ITEM NAME H 20 $
 ITEM BLDG H 5 $
 ITEM ROOM H 5 $
 ITEM NUMBER I 60 $
 END

```



- TABLE DATA R 3 S \$ An ordinary table with no packing and including a subordinate overlay declaration. Since no packing was specified, the overlay will be done on a full word basis. The word containing II will overlay the word containing HH, which will be followed by the word containing AA. This example has the following data structure.
 

```

 BEGIN
 ITEM AA A 20 S 5 $
 ITEM HH H 6 $
 ITEM II I 36 $
 OVERLAY HH = II, AA $
 END

```

| NENT (3)               | Word | Entry |
|------------------------|------|-------|
| HH(\$0\$)<br>II(\$0\$) | 0    | } 0   |
| AA(\$0\$)              | 1    |       |
| HH(\$1\$)<br>II(\$1\$) | 2    | } 1   |
| AA(\$1\$)              | 3    |       |
| HH(\$2\$)<br>II(\$2\$) | 4    | } 2   |
| AA(\$2\$)              | 5    |       |

- TABLE PACK V 30 S D \$  
 BEGIN  
 ITEM LIT0 H 60 \$  
 ITEM LIT1 H 42 \$  
 ITEM LIT2 H 11 \$  
 ITEM LIT3 H 11 \$  
 ITEM LIT4 H 13 \$  
 ITEM LIT5 H 9 \$  
 ITEM PCK1 H 7 \$  
 ITEM PCK2 H 7 \$  
 ITEM PCK3 H 7 \$  
 ITEM PCK4 H 26 \$  
 OVERLAY LIT0 = PCK1, LIT1 \$  
 OVERLAY LIT0 = PCK2, LIT2 \$  
 OVERLAY LIT0 = PCK3, LIT4,  
                   PCK4, LIT3 \$  
 OVERLAY LIT3 = LIT5 \$  
 END

An example of the use of a subordinate overlay in conjunction with dense packing. This combination will cause the table to result in an allocation structure identical to that of the defined table as shown on page 5-32.

#### DEFINED TABLE DECLARATIONS

In a defined table, the user directly controls the allocation of his table. He specifies the number of words per entry and assigns the table items to a particular word and bit position in the entry. The following rules apply to defined table declarations:

- The programmer must completely describe the positions of items within entries of defined table declarations. Packing specifications are not allowed in the defined table header.
- Size is determined by multiplying number<sub>1</sub> by number<sub>2</sub> + 1 for the NENT.
- Variable-length entries may be defined by specifying items to exist in a word of entry greater than number<sub>2</sub>. While this is a useful table structure, the table must be serial if variable-length entries are utilized. Otherwise, referencing those items beyond the words per entry specified would set or use information beyond the table allocation. (The location of an entry is found by multiplying the subscript by number<sub>2</sub>).

- Only literal items may cross word boundaries; if they do, the literal must start on a byte boundary. (There are six bits between byte boundaries.)

The complete definition of a defined table requires a TABLE declaration followed by a defined entry declaration consisting of one or more ITEM declarations and/or one or more STRING declarations. OVERLAY declarations cannot be included in a defined entry description. Any overlaying of items can be explicitly defined in the declaration. The format is:

```
TABLE [table-name] {RV} number1 [SP] number2 $
 BEGIN
 {
 [one or more defined item descriptions (see page 5-32)]
 [one or more string item descriptions (see page 5-33)]
 }
 END
```

where

table-name = identifies the name of the table

V = variable table

R = rigid table

number<sub>1</sub> = the maximum number of active entries for a variable-length table or the exact number of entries for a rigid-length table.

P = parallel table

S = serial table

number<sub>2</sub> = number of words in an entry.

Examples:

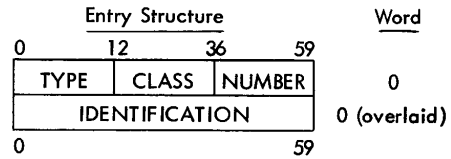
- TABLE FLIGHT V 1000 S 3 \$  
 BEGIN  
 ITEM IDENT H 5 0 0 \$  
 ITEM ALTITUDE I 30 U 0 30 \$  
 ITEM LONGITUDE I 30 U 1 0 \$  
 ITEM LATITUDE I 30 S 1 30 \$  
 ITEM SPEED F 2 0 \$  
 END

Each entry in the table has the following defined structure.

| Entry Structure |          | Word |
|-----------------|----------|------|
| 0               | 30       | 59   |
| IDENT           | ALTITUDE | 0    |
| LONGITUDE       | LATITUDE | 1    |
| SPEED           |          | 2    |

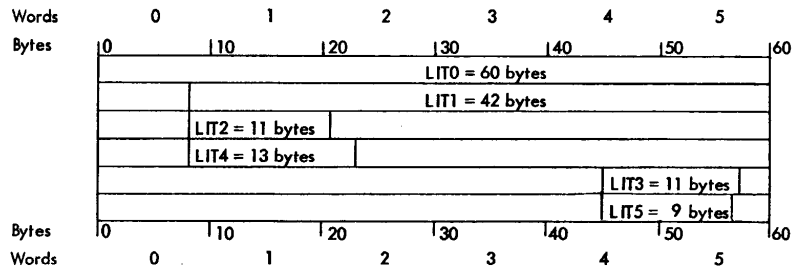
- TABLE PARTS V 500 S 1 \$  
 BEGIN  
 ITEM TYPE T 2 0 0 \$  
 ITEM CLASS T 4 0 12 \$  
 ITEM NUMBER T 4 0 36 \$  
 ITEM IDENTIFICATION T 10 0 0 \$  
 END

Each entry in this table consists of one word.



- TABLE TST V 30 S 6 \$  
 BEGIN  
 ITEM LIT0 H 60 0 0 \$  
 ITEM LIT1 H 42 0 42 \$  
 ITEM LIT2 H 11 0 42 \$  
 ITEM LIT3 H 11 4 36 \$  
 ITEM LIT4 H 13 0 42 \$  
 ITEM LIT5 H 9 4 36 \$  
 END

The items overlay each other and permit the same data to be accessed as a part of several different Hollerith items. Item LIT0 utilizes the entire entry. Items LIT1, LIT2, and LIT4 all start at the seventh byte of the entry and extend for 42, 11, and 13 bytes, respectively. Items LIT3 and LIT5 both start at byte 46 of the entry and extend for 11 and 9 bytes, respectively, as shown in the data structure below:



#### DEFINED ITEM DECLARATION

The defined item declaration allows the programmer to specify the values of one or more items in a table. The format is:

$$\left[ \text{one or more ITEM item-name item-description number}_3 \text{ number}_4 \begin{matrix} N \\ M \\ D \end{matrix} \$ \right]$$

[one-dimensional constant-list]

where

item-name = item identifier

item-description = defines the characteristics of the item

- number<sub>3</sub> = declares which word in the entry contains the item or in which word the item begins for long literal items. Words are numbered from 0 to number<sub>2</sub>-1.
- number<sub>4</sub> = the bit within the word in which the item begins. Bits are numbered from the left beginning with 0.
- N, M, D = packing specification; this does not direct the packing, but describes the packing that results from number<sub>3</sub>, number<sub>4</sub>, item-declaration, and the conditions of adjacent items in the entry. The one-dimensional constant list performs the same function as in an ordinary entry description.

#### STRING ITEM DECLARATION

The STRING item declaration allows the programmer to place more than one value for an item in each entry of a table. Each occurrence of a string item within an entry is called a bead. Since STRING item declarations require bit positioning to be specified by the programmer, they may be used only in defined tables. The format is:

$$\left[ \begin{array}{l} \text{one or more STRING string-name item-description number}_3 \text{ number}_4 \begin{bmatrix} N \\ M \\ D \end{bmatrix} \\ \text{number}_5 \text{ number}_6 \$ [\text{two-dimensional constant-list}] \end{array} \right]$$

where

- string-name = identifies the string item
- item-description = defines the characteristics of the string item
- number<sub>3</sub> = word within the entry where the item starts; can be from zero to number<sub>2</sub>-1.
- number<sub>4</sub> = bit position within the designated word where the item starts; can be from 0-59. A literal item that crosses word boundary must start on a byte boundary.
- number<sub>5</sub> = frequency of occurrence of string items in the words of the entry. Beads of the string are in every number<sub>5</sub> word of the entry.
- number<sub>6</sub> = the number of beads in each word of the entry, since it must be an integer, the beads cannot be declared longer than one word.
- N, M, or D = no packing or dense packing. If no packing is selected, there must not be more than one bead per word. If dense packing is specified, number<sub>6</sub> may be as large as the number of beads which will fit in a word.



Examples:

- TABLE ALPHA R 100 S 3 \$  
 BEGIN  
 STRING AA H 2 0 0 D 1 5 \$  
 END

The structure of this table consists exclusively of the display coded two-byte beads AA. The first bead starts in word 0, bit 0; each word of the entry contains the beads defined in the string declaration with five beads per word. If dense packing is omitted, the packing would be the same as being specified.

| Entry Structure |    |    |    |    | Word | Entry |
|-----------------|----|----|----|----|------|-------|
| 0               | 12 | 24 | 36 | 48 | 59   |       |
| AA              | AA | AA | AA | AA |      | 0     |
| AA              | AA | AA | AA | AA |      | 1     |
| AA              | AA | AA | AA | AA |      | 2     |

- TABLE V 100 S 4 \$  
 BEGIN  
 STRING PARTS I 15 U 1 0 2 4 \$  
 ITEM NAME1 H 10 0 0 \$  
 ITEM CODE B 2 0 \$  
 ITEM NAME2 H 8 2 6 \$  
 END

This example shows a mixture of items and string declarations. It has the following data structure.

| Entry Structure |       |       |       | Word  |    |
|-----------------|-------|-------|-------|-------|----|
| 0               | 8     | 15    | 30    | 45 56 | 59 |
| NAME1           |       |       |       |       | 0  |
| PARTS           | PARTS | PARTS | PARTS |       | 1  |
| ← CODE          | NAME2 |       |       |       | 2  |
| PARTS           | PARTS | PARTS | PARTS |       | 3  |

- TABLE TA V 3 P 6 \$  
 BEGIN  
 STRING TAA I 30 S 0 0 D 2 2 \$  
 STRING TAB I 60 S 1 0 D 2 1 \$  
 END

Each entry of TABLE TA contains two string declarations. The strings are specified such that each occupies alternate words of the entry. Thus, TAA is in words 0, 2, and 4. TAB is in words 1, 3, and 5 as shown below.

| Words | 0   | 1   | 2   | 3   | 4   | 5   |
|-------|-----|-----|-----|-----|-----|-----|
| Entry | TAA | TAA | TAB | TAA | TAA | TAB |

LIKE TABLE DECLARATION

A like table declaration is used to describe a new table which is similar in construction to a pattern table previously declared with a name. The like table must be named and the name

must be formed by suffixing a letter or numeral to the name of the pattern table. The following rules apply:

- The entry of a like table need not be described because the order and structure of the table is automatically generated following the order and structure of the pattern table.
- The items are named by suffixing the same letter or numeral used in the table name to the pattern table items.
- A like table declaration is identified by the letter L at the end of the declaration.
- The optional fields may be included; if they are, they will override the parameters of the pattern table. For example, the structure may be changed from serial to parallel or from dense to no packing.
- If no change is desired, this information can be obtained from the declaration of the pattern table.
- The packing specification must not be included if the pattern table is a defined table.

The format is:

TABLE name  $\begin{bmatrix} V \\ R \end{bmatrix}$  [number of entries]  $\begin{bmatrix} P \\ S \end{bmatrix}$   $\begin{bmatrix} N \\ M \\ D \end{bmatrix}$  L \$

Examples:

- TABLE DIRECTORY2 L \$      Similar to the ordinary table DIRECTORY on page 5-29, the items are named: NAME2, BLDG2, ROOM2, and NUMBER2.
- TABLE FLIGHTX P L \$      Describes a table that is like the pattern table FLIGHT, except this table is specified as parallel in structure.

### CONSTANT LIST

The table item-declaration may be followed by a one-dimensional constant list which is like the constant list that sets values in a one-dimensional array. Each constant presets the value of the item in the successive entries. The first constant presets the value of the entry 0, the second presets entry 1, and so forth. No more than the first 5800 words of a table can be preset.

The string declaration may be followed by a two-dimensional constant list. This list declares values for some of the beads of the string. The two-dimensional list is composed of a set of one-dimensional constant lists between BEGIN-END brackets. The first one-dimensional constant list provides values for the beads in entry 0, the second for beads in entry 1, and so forth. Within each list is the set of values for each bead in an entry with the first constant setting the value of bead 0, the second of bead 1, and so forth.

Examples:

- ```

TABLE PARTS V 500 S 1 $
  BEGIN
    ITEM TYPE T 2 0 0 $
      BEGIN
        2T(AA) 2T(BB) 2T(CC) 2T(DD)
        2T(DD) 2T(EF) 2T(FG)
      END
    ITEM CLASS T 4 0 12 $
      BEGIN
        4T(0101) 4T(0001) 4T(0111)
        4T(0110) 4T(0102) 4T(0100)
      END
    ITEM NUMBER T 4 0 36 $
      BEGIN
        4T(0001) 4T(0002) 4T(0003)
        4T(0004) 4T(0005) 4T(0006)
      END
    ITEM IDENTIFICATION T 10 0 0 $
  END

```

The PARTS table described on page 5-32 could be preset in this way. The storage allocation and presets are shown below. The compiler will not set NENT because this table is variable.

	0	12	36	59	
	NENT				
2T(AA)	4T(0101)	4T(0001)			ENTRY(\$0\$)
2T(BB)	4T(0001)	4T(0002)			ENTRY(\$1\$)
2T(CC)	4T(0111)	4T(0003)			ENTRY(\$2\$)
2T(DD)	4T(0110)	4T(0004)			ENTRY(\$3\$)
2T(EF)	4T(0102)	4T(0005)			ENTRY(\$4\$)
2T(FG)	4T(0100)	4T(0006)			ENTRY(\$5\$)
TYPE	CLASS	NUMBER			
IDENTIFICATION					

- ```

TABLE ALPHA R 6 S 3 $
 BEGIN STRING AA H 2 0 0
 D 1 5 $ END
 BEGIN
 BEGIN 2H(00) 2H(01) 2H(02) 2H(03)
 2H(04) 2H(05) 2H(06) 2H(07)
 2H(08) 2H(09) END
 BEGIN 2H(10) 2H(11) 2H(12) 2H(13)
 2H(14) 2H(15) 2H(16) 2H(17)
 2H(18) 2H(19) 2H(1A) END
 BEGIN 2H(20) 2H(21) 2H(22) 2H(23)
 2H(24) 2H(25) END
 BEGIN 2H(30) 2H(31) 2H(32) 2H(33)
 2H(34) 2H(35) 2H(36) 2H(37)
 END
 BEGIN 2H(40) 2H(41) 2H(42) 2H(43)
 2H(44) 2H(45) 2H(46) 2H(47)
 2H(48) 2H(49) 2H(4A) 2H(4B)
 2H(4C) 2H(4D) 2H(4E) END
 BEGIN 2H(50) 2H(51) 2H(52) 2H(53)
 2H(54) 2H(55) 2H(56) END
 END

```

This table presets certain beads in the table to initial values. The table has the structure shown below. There is no preassignment of data in the shaded areas of the diagram; these areas contain whatever was in memory when the program was loaded.

| 0      | 12     | 24     | 36     | 48     | 59 | Word | Entry |
|--------|--------|--------|--------|--------|----|------|-------|
| 2H(00) | 2H(01) | 2H(02) | 2H(03) | 2H(04) |    | 0    | } 0   |
| 2H(05) | 2H(06) | 2H(07) | 2H(08) | 2H(09) |    | 1    |       |
|        |        |        |        |        |    | 2    |       |
| 2H(10) | 2H(11) | 2H(12) | 2H(13) | 2H(14) |    | 0    | } 1   |
| 2H(15) | 2H(16) | 2H(17) | 2H(18) | 2H(19) |    | 1    |       |
| 2H(1A) |        |        |        |        |    | 2    |       |
| 2H(20) | 2H(21) | 2H(22) | 2H(23) | 2H(24) |    | 0    | } 2   |
| 2H(25) |        |        |        |        |    | 1    |       |
|        |        |        |        |        |    | 2    |       |
| 2H(30) | 2H(31) | 2H(32) | 2H(33) | 2H(34) |    | 0    | } 3   |
| 2H(35) | 2H(36) | 2H(37) |        |        |    | 1    |       |
|        |        |        |        |        |    | 2    |       |
| 2H(40) | 2H(41) | 2H(42) | 2H(43) | 2H(44) |    | 0    | } 4   |
| 2H(45) | 2H(46) | 2H(47) | 2H(48) | 2H(49) |    | 1    |       |
| 2H(4A) | 2H(4B) | 2H(4C) | 2H(4D) | 2H(4E) |    | 2    |       |
| 2H(50) | 2H(51) | 2H(52) | 2H(53) | 2H(54) |    | 0    | } 5   |
| 2H(55) | 2H(56) |        |        |        |    | 1    |       |
|        |        |        |        |        |    | 2    |       |

```

• TABLE TA V 3 P 6 $
 BEGIN
 STRING TAA I 30 U 0 0 D 2 2 $
 BEGIN BEGIN 1 2 3 4 5
 6 END
 BEGIN 7 8 9 10 11
 12 END
 BEGIN 13 14 15 16 17
 18 END
 END
 STRING TAB I 60 U 0 0 D 2 2 $
 BEGIN BEGIN 18 1 16 END
 BEGIN 3 14 5 END
 BEGIN 12 7 10 END
 END
 END
 END

```

TA is declared as parallel and preset with a constant list. The storage allocation and preset values are shown below. The compiler will not set NENT as this table is variable.

| NENT         |              | Entry |    |   |          |
|--------------|--------------|-------|----|---|----------|
| TAA(\$0,0\$) | TAA(\$1,0\$) | 1     | 2  | 0 | } Word 0 |
| TAA(\$0,1\$) | TAA(\$1,1\$) | 7     | 8  | 1 |          |
| TAA(\$0,2\$) | TAA(\$1,2\$) | 13    | 14 | 2 |          |
| TAB(\$0,0\$) |              | 18    |    | 0 | } Word 1 |
| TAB(\$0,1\$) |              | 3     |    | 1 |          |
| TAB(\$0,2\$) |              | 12    |    | 2 |          |
| TAA(\$2,0\$) | TAA(\$3,0\$) | 3     | 4  | 0 | } Word 2 |
| TAA(\$2,1\$) | TAA(\$3,1\$) | 9     | 10 | 1 |          |
| TAA(\$2,2\$) | TAA(\$3,2\$) | 15    | 16 | 2 |          |
| TAB(\$1,0\$) |              | 1     |    | 0 | } Word 3 |
| TAB(\$1,1\$) |              | 14    |    | 1 |          |
| TAB(\$1,2\$) |              | 7     |    | 2 |          |
| TAA(\$4,0\$) | TAA(\$5,0\$) | 5     | 6  | 0 | } Word 4 |
| TAA(\$4,1\$) | TAA(\$5,1\$) | 11    | 12 | 1 |          |
| TAA(\$4,2\$) | TAA(\$5,2\$) | 17    | 18 | 2 |          |
| TAB(\$2,0\$) |              | 16    |    | 0 | } Word 5 |
| TAB(\$2,1\$) |              | 5     |    | 1 |          |
| TAB(\$2,2\$) |              | 10    |    | 2 |          |

If the table is declared serial, the data allocation and preset values will be those shown below.

| NENT         |              | Word |    |   |           |
|--------------|--------------|------|----|---|-----------|
| TAA(\$0,0\$) | TAA(\$1,0\$) | 1    | 2  | 0 | } Entry 0 |
| TAB(\$0,0\$) |              | 18   |    | 1 |           |
| TAA(\$2,0\$) | TAA(\$3,0\$) | 3    | 4  | 2 |           |
| TAB(\$1,0\$) |              | 1    |    | 3 |           |
| TAA(\$4,0\$) | TAA(\$5,0\$) | 5    | 6  | 4 |           |
| TAB(\$2,0\$) |              | 16   |    | 5 |           |
| TAA(\$0,1\$) | TAA(\$1,1\$) | 7    | 8  | 0 | } Entry 1 |
| TAB(\$0,1\$) |              | 3    |    | 1 |           |
| TAA(\$2,1\$) | TAA(\$3,1\$) | 9    | 10 | 2 |           |
| TAB(\$1,1\$) |              | 14   |    | 3 |           |
| TAA(\$4,1\$) | TAA(\$5,1\$) | 11   | 12 | 4 |           |
| TAB(\$2,1\$) |              | 5    |    | 5 |           |
| TAA(\$0,2\$) | TAA(\$1,2\$) | 13   | 14 | 0 | } Entry 2 |
| TAB(\$0,2\$) |              | 12   |    | 1 |           |
| TAA(\$2,2\$) | TAA(\$3,2\$) | 15   | 16 | 2 |           |
| TAB(\$1,2\$) |              | 7    |    | 3 |           |
| TAA(\$4,2\$) | TAA(\$5,2\$) | 17   | 18 | 4 |           |
| TAB(\$2,2\$) |              | 10   |    | 5 |           |

## REFERENCING TABLES

To make full use of the power and variety of declarations possible for table structures, several means of referencing tables and data in tables are available:

- The table name itself refers to the entire table structure, including both data items and the NENT. This is used to pass the table as an actual procedure input or output parameter, or in an INPUT or OUTPUT statement to transfer the table to or from a binary file.
- A table name followed by a two-component index is used to separate the input or output entries from the first component through second component.
- Table items are referenced by the item-name followed by a one-component index.
- String items are referenced by the string item name following by a two-component index. The first component is for the position in the entry, the second is for the entry in the table.
- The functional modifiers ENTRY or ENT, NENT, and NWDSSEN.

Any numeric formula may be used as a subscript; if the formula does not yield an integer value, the formula is truncated to integer. The numeric formula may be a subscripted value or several subscripted values. The maximum subscript nesting level is 20.

A formula that begins with 0 and runs to a maximum of n-1 for a dimension with n elements. If the formula yields a result which does not lie between 0 and n-1, it will be performed with unspecified results.

Examples:

- OUTPUT FILEA TA \$                      Transfers the entire table TA to FILEA. The NENT of the table will be transferred also. This should only be used with a binary file.
- SEARCH (PARTS = NBR) \$                Table PARTS is passed as an input parameter to procedure SEARCH, which examines the table and returns a value to the simple variable NBR.
- OUTPUT FILEA FLIGHT (\$15...25\$) \$    Starting with entry 15 and continuing through entry 25, this example transfers the 11 consecutive entries of table FLIGHT to FILEA.
- TYPE(\$3\$)                                Refers to the fourth element of table item TYPE.

- TAA(\$4,1\$)

Refers to the fifth bead in the second entry of table TA. This was preset to the value 11 in the preset examples.

## TABLE MODIFIERS

### ENTRY OR ENT TABLE REFERENCE

Entire entries can be referenced using the functional modifiers ENTRY or ENT. The format is:

$$\left\{ \begin{array}{l} \text{ENTRY} \\ \text{ENT} \end{array} \right\} \left( \left\{ \begin{array}{l} \text{table-name} \\ \text{item-name} \end{array} \right\} (\$index\$) \right) \$$$

If the table has no name, the name of an item in the table can be used. The index indicates the particular entry referenced. An entry variable can be tested for equality only with the value 0 or with another entry variable. (An entry equals 0 if every bit in the entry is equal to 0 and equal to another entry if corresponding bits are the same.) Furthermore, an entry may be assigned only to another entry or to 0. If one entry variable is assigned to another, every bit in the entry is set equal to the corresponding bit in the other; if the value 0 is assigned to an entry, every bit is set to 0.

### NENT TABLE REFERENCE

The NENT functional modifier can be used to refer to the number of entries in a table. It is a variable if the table has a variable number of active entries and a constant if rigid was specified. NENT can be used in formulas wherever an integer variable or a constant is used. NENT is the first word prior to the first data word of any table; it is either the current number of active entries for a variable-length table (if the programmer has set NENT) or the maximum number for a rigid-length table. The format is:

$$\text{NENT} \left( \left\{ \begin{array}{l} \text{table-name} \\ \text{item-name} \end{array} \right\} \right) \$$$

### NWDSSEN TABLE REFERENCE

The number of words in a table entry can be referenced by the NWDSSEN modifier. It is a constant for any table and can be used anywhere that an integer constant is legal. The format is:

$$\text{NWDSSEN} \left( \left\{ \begin{array}{l} \text{table-name} \\ \text{item-name} \end{array} \right\} \right) \$$$

The number of table entries is limited to  $2^{17}-1$ . The number of words for an entry may not exceed 31; the number of beads per word in a string may not be more than 31.

## DATA ALLOCATION

The allocation of storage is completely controlled by the compiler except when the user specifies a particular order with the overlay declaration or specifies the packing within a table by selecting dense packing, defined table declaration, or the subordinate overlay declaration. For simple items, arrays, and tables with no packing, items are allocated one to a word; literals longer than one word are assigned to the least number of full words necessary to contain the required bytes. Items of less than one word in length are right-justified. Literals longer than one word are left-justified.

## OVERLAY DECLARATION

The OVERLAY declaration is used to position in storage previously declared or previously mode defined data items, tables, and arrays. By this declaration, the user can assign the same starting address and determine the order of allocation for tables, arrays, and simple items. He can also access the same data in two or more ways by overlaying items with the desired types. OVERLAY furnishes the user with equivalence of locations, but not the values stored in the locations. The format is:

$$\text{OVERLAY name}_1 \left[ \left\{ \begin{array}{l} ' \\ = \end{array} \right\} \text{name}_2 \right] \left[ \dots \left\{ \begin{array}{l} ' \\ = \end{array} \right\} \text{name}_n \right] \$$$

The following rules apply to OVERLAY declarations:

- The names must be previously declared or previously mode defined items, arrays, or tables.
- If only one name is specified, it merely provides early allocation of the named data.
- Names separated by commas are assigned sequential locations in storage.
- Names separated by equal signs are overlaid; that is, the data structures identified by the names following the equal signs are allocated to the same starting location.
- Commas and equal signs can be intermixed in one OVERLAY declaration.
- The NENT word is used in computing the allocation of overlaid tables, not the first data word.
- A COMPOOL-defined name may be used only if it immediately follows the word OVERLAY.



- The amount of storage allocated to overlaid data is that required by the longest piece of data in the OVERLAY declaration.
- Only the relative location of data structures may be controlled; allocation to a fixed address within the program is not permitted.
- An item name may appear only once in any single OVERLAY declaration; a name may be specified in more than one OVERLAY only if the name immediately follows the word OVERLAY in the second or succeeding declarations.

Items within a table can be overlaid specifically by the subordinate OVERLAY declaration (see page 5-25) following the item declarations in an ordinary table, or by assigning the same location in a defined table.

The number of names that can appear in OVERLAY statements in one source program is 2000 minus the number of OVERLAY statements in the program.

Examples:

- OVERLAY AA, BB = DD, EE \$      This is a legal example of specifying a name  
OVERLAY DD = FF, GG \$      in more than one overlay declaration.  
OVERLAY GG = HH \$
- OVERLAY OO, KK = LL, MM \$      Illegal example; LL and KK do not follow the  
OVERLAY NN = LL \$      word OVERLAY in the second and succeeding  
OVERLAY LL, KK = XX \$      declaration of the same name.
- OVERLAY TABLA, TABLB,      Previously defined tables, TABLA, TABLB,  
  TABLC \$      and TABLC, are put in consecutive locations.  
The storage space is allocated according to the  
length of a rigid table or the maximum length  
of a variable table.
- OVERLAY TABLA = TABLB =      The three tables are assigned storage with  
  TABLC \$      the same origin as shown below:

|              |        |        |  |
|--------------|--------|--------|--|
| NENT(TABLA)  | TABLA  | TABLA  |  |
| NENT(TABL B) | TABL B | TABL B |  |
| NENT(TABL C) | TABL C | TABL C |  |
| Location 0   | 1      | 2      |  |

- `OVERLAY TAB1, TAB2 = TAB3, TAB4$` TAB2 will be stored immediately following TAB1, and TAB4 immediately after TAB3. TAB1 and TAB3 will have a common origin.

|            |      |            |      |
|------------|------|------------|------|
| NENT(TAB1) | TAB1 | NENT(TAB2) | TAB2 |
| NENT(TAB3) | TAB3 | NENT(TAB4) | TAB4 |

- `OVERLAY TAB1, TAB2 = ITEM1, ARRAY1$`

If a simple item is given a common origin with a table, it will share the location of the NENT word of the table. Thus, TAB2 is placed immediately following TAB1; ITEM1 overlays the NENT word of TAB1 and is followed by ARRAY1.

|            |        |            |      |
|------------|--------|------------|------|
| NENT(TAB1) | TAB1   | NENT(TAB2) | TAB2 |
| ITEM1      | ARRAY1 |            |      |

- `ITEM HOL H 30 $`  
`ITEM KK 3 $`  
`TABLE TAB R 4 $`  
`BEGIN`  
`ITEM TAB1 I 60 S $`  
`END`  
`ARRAY ARR 5 I 60 S $`

This example defines two items, one integer, and one literal.

- `OVERLAY TAB = KK, HOL $`

This OVERLAY statement has the relative storage assignments shown.

|           |     |  |
|-----------|-----|--|
| KK        | HOL |  |
| NENT(TAB) | TAB |  |

- `OVERLAY HOL = KK, TAB $`

This OVERLAY statement has the relative storage assignments shown

|     |           |     |
|-----|-----------|-----|
| HOL |           |     |
| KK  | NENT(TAB) | TAB |

- OVERLAY ARR = TAB \$

This OVERLAY statement has the relative storage assignments shown

|           |     |  |
|-----------|-----|--|
| NENT(TAB) | TAB |  |
| ARR       |     |  |

- ITEM INT I 18 S \$
- ITEM HOL H 3 \$
- ITEM HOLD H 40 \$
- ITEM LIT H 30 \$
- ARRAY HHH 3 H 10 \$

Data declarations

- OVERLAY HOLD = HOL, LIT \$
- OVERLAY HOL = INT \$
- OVERLAY LIT = HHH \$

Overlay declarations which produce the data structure shown below:

|      |            |            |            |
|------|------------|------------|------------|
| INT  | HHH(\$0\$) | HHH(\$1\$) | HHH(\$2\$) |
| HOL  | LIT        |            |            |
| HOLD |            |            |            |

The result of this structure is that location 0 may be used to contain either a 3-byte Holerith item or an 18-bit signed integer. Locations 1, 2, and 3 may be accessed as a 30-byte item, or separately as 10-byte array items. The entire structure may be moved within core by moving the 40-byte item HOLD, or may be output as a binary record.

---

**GENERAL**

Data is input from or output to files residing on some external storage device, such as tape, cards, printer, or typewriter. The structure of data in external storage is defined by a file declaration. Input/output statements are used to open or close files when a read, write, or transfer operation is to be performed. The routines for FORTRAN-formatted output are also discussed in this section.

**JOVIAL FILES**

JOVIAL files are a sequence of logical records followed by an end-of-file indicator. A file is a sequential string of bits residing on an external storage device. A file is divided into one or more logical records which may be in either binary or BCD mode.

Binary records are written and read as SCOPE binary records. Each SCOPE logical record corresponds exactly to one JOVIAL logical record. Binary files are terminated by an end-of-file indicator.

BCD records are written and read as SCOPE coded records. Each BCD record is terminated by the display code end-of-line mark (12-bit binary zero byte). SCOPE logical end-of-record marks are ignored when reading coded files unless the file was declared with device name INPUT. For a file with device name INPUT, a SCOPE logical end-of-record mark acts as an end-of-file. The SCOPE logical end-of-record mark is not entered on a JOVIAL BCD file that is being written; when records are written on a BCD file, the records are separated by SCOPE end-of-line marks.

Data structures which could possibly contain a 12-bit binary zero byte end-of-line mark should be output only to binary files. If they were output to BCD files, the data between each end-of-line mark would be input as if it were a separate record. Thus, it could not be read back, nor would it be desirable to use it on a device which used the first character of each line for carriage control. When table entries, entire tables, or entire arrays contain unused space or data in binary format, i. e., nonliteral items, they fall into this category.

## CARRIAGE CONTROL

BCD files which are to be used on display or printing devices must satisfy the requirements of the particular device in regard to carriage control and maximum record length. Since these are features of a particular installation or device and not of the JOVIAL language, documentation for the particular system and device should be consulted to determine these parameters.

The line printer uses the first character of a BCD record for carriage control. This character is not printed. The second character in the line appears in the first position; therefore, a maximum number of 137 characters can be specified for a print line, but 136 is the maximum number of characters that can be printed. The carriage control characters available for both the 501 and 512 Line Printers (except as noted) under SCOPE 3.3 are:

| <u>Character</u> | <u>Action Before Printing</u> | <u>Action After Printing</u>                        |
|------------------|-------------------------------|-----------------------------------------------------|
| A                | Space 1                       | Eject to top of next page <sup>†</sup>              |
| B                | Space 1                       | Skip to last line of page <sup>†</sup>              |
| C                | Space 1                       | Skip to channel 6                                   |
| D                | Space 1                       | Skip to channel 5                                   |
| E                | Space 1                       | Skip to channel 4                                   |
| F                | Space 1                       | Skip to channel 3                                   |
| G                | Space 1                       | Skip to channel 2                                   |
| H                | Space 1                       | Skip to channel 1 (501)<br>Skip to channel 11 (512) |
| I                | Space 1                       | Skip to channel 7 (512)                             |
| J                | Space 1                       | Skip to channel 8 (512)                             |
| K                | Space 1                       | Skip to channel 9 (512)                             |
| L                | Space 1                       | Skip to channel 10 (512)                            |
| 1                | Eject to top of next page     | No space <sup>†</sup>                               |
| 2                | Skip to last line on page     | No space <sup>†</sup>                               |
| 3                | Skip to channel 6             | No space                                            |
| 4                | Skip to channel 5             | No space                                            |
| 5                | Skip to channel 4             | No space                                            |
| 6                | Skip to channel 3             | No space                                            |
| 7                | Skip to channel 2             | No space                                            |

<sup>†</sup>The top of a page is indicated by a punch in channel 8 of the carriage control tape for the 501 printer and channel 1 for the 512 printer. The bottom of a page is channel 7 in the 501 and 12 in the 512.

| <u>Character</u> | <u>Action Before Printing</u>                       | <u>Action After Printing</u> |
|------------------|-----------------------------------------------------|------------------------------|
| 8                | Skip to channel 1 (501)<br>Skip to channel 11 (512) | No space<br>No space         |
| 9                | Skip to channel 7 (512)                             | No space                     |
| X                | Skip to channel 8 (512)                             | No space                     |
| Y                | Skip to channel 9 (512)                             | No space                     |
| Z                | Skip to channel 10 (512)                            | No space                     |
| +                | No space                                            | No space                     |
| 0 (zero)         | Space 2                                             | No space                     |
| - (minus)        | Space 3                                             | No space                     |
| blank            | Space 1                                             | No space                     |

When the following characters are used for carriage control, no printing takes place. The remainder of the line will not be printed.

|                         |                                                                                                                                                                                                                                            |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Q                       | Clear auto page eject                                                                                                                                                                                                                      |
| R                       | Select auto page eject                                                                                                                                                                                                                     |
| S                       | Clear 8 vertical lines per inch (512)                                                                                                                                                                                                      |
| T                       | Select 8 vertical lines per inch (512)                                                                                                                                                                                                     |
| PM<br>(columns 1 and 2) | Output remainder of line (up to 30 characters) on the B display and the dayfile and wait for the JANUS typein /OKuu. For files assigned to a printer, n.GO. must be typed to allow the operator to change forms or carriage control tapes. |
| any other               | Acts as a blank                                                                                                                                                                                                                            |

Any preprint skip operation of 1, 2, or 3 lines that follows a post skip operation is reduced to 0, 1, or 2 lines.

The functions S and T should be given at the top of a page. In other positions, S and T can cause spacing to be different from the stated spacing. Q and R need not be given at the top of a page as each causes a page eject before performing its function.

#### **FILE INPUT/OUTPUT**

The user does not normally have direct access to files on physical devices other than the system device. The SCOPE file handling concept should be thoroughly understood prior to manipulating files in JOVIAL. (Refer to the SCOPE Operating System Reference Manual.)

Both binary and BCD JOVIAL files use FORTRAN Extended input/output routines. The interface between these routines and an executing JOVIAL program is accomplished by the routine JOVIO which transforms JOVIAL data structures to a form acceptable to the FORTRAN Extended I/O routines.

## FILE DECLARATION

Every JOVIAL file must be declared in a file declaration. The logical records contained in a file may not be manipulated directly by a JOVIAL program, but may be read into a JOVIAL-defined data structure in central memory. The format is:

FILE name  $\left\{ \begin{array}{c} B \\ H \end{array} \right\}$  number<sub>1</sub>  $\left\{ \begin{array}{c} V \\ R \end{array} \right\}$  number<sub>2</sub> status-list device-name \$

where:

name = A JOVIAL name in the program supplied by the user by which the file is known to the JOVIAL program.

- B or H** = Specifies the mode in which data is recorded in the file. Numeric, Boolean, and status constants or variables may be used as operands in binary file statements. Only literal constants and variables may be specified in operations involving Hollerith file statements.
- B = Binary  
H = Hollerith (BCD)
- number<sub>1</sub>** = Indicates the number of logical records in the file. This number has no meaning in a 6000 JOVIAL program. It may be any positive number, including zero.
- V or R** = Indicates the type of records to be used in the file. This has no meaning in a 6000 JOVIAL program.
- V = variable length records  
R = rigid length records
- number<sub>2</sub>** = This integer determines the size of the buffer for the file (see discussion under Buffer Size).
- status list** = At least one valid status constant must be declared, but the first four constants declared will correspond to the four format status constant values discussed under File Status. If more than four status constants are declared, only the first four are accepted; the rest are ignored unless an attempt is made to reference them in the program in which case a fatal compiler diagnostic error message will result.
- device name** = The device name is the name by which the file is known to SCOPE (see discussion under SCOPE File Name).

### SCOPE FILE NAME

A file declaration contains both a file name and a device name. The file name is used to reference the file within the JOVIAL program. The device name is used to logically associate JOVIAL files with SCOPE files. The device name is the name by which the file is known to SCOPE and may be used only in the file declaration or SCOPE control cards. The device name in the file declaration may refer to an existing SCOPE file or may be used to create a new one.



A file name can be used in three contexts:

- As a file identifier in an INPUT, OUTPUT, SHUT, or OPEN statement;
- As functionally modified by POS to obtain the current logical record position, or to alter the logical record position;
- In Boolean formulas where the file name is related to a status constant in the status list of file states.

The device name must conform to JOVIAL name rules, with the exception that the primitives INPUT and OUTPUT can be used according to the standard SCOPE file meaning of INPUT and OUTPUT: INPUT refers to the file where input from the card reader may be read; OUTPUT refers to the file where coded output to be printed may be written.

The device names TAPE1 through TAPE99 correspond to the FORTRAN tape units 1-99; the names PUNCH and PUNCHB correspond to the standard SCOPE card punch files.

- A file name may not be used in more than one file declaration in any compilation, whether it is the main program or a subprogram.
- The device name consists of up to seven characters. The first six must be unique because entry points are used within programs to refer to the FET file. If the name is greater than six characters, the symbol ≡ is added to the end as the seventh character.
- A file used only for monitored output in a program or subprogram need not be explicitly declared. If the declaration for a MONITOR file is omitted, the computer will create one using as the device name the name following the M option on the JOVIAL control card.

#### **BUFFER SIZE**

The size of the buffer for a particular JOVIAL file must be specified in the file declaration; in the file declaration format, the buffer size is given in number<sub>2</sub> as an integer. The compiler adds one to this integer to create the buffer. The integer should be at least equal to the size of the physical record unit (PRU) of the device to which the file is assigned. (Refer to the SCOPE Reference Manual listed in the Preface.) The PRU size for the standard SCOPE devices are:

- Disk - 64 words
- Coded tape - 128 words
- Binary tape - 512 words

I/O is utilized more efficiently by declaring a buffer size greater than the PRU size of the device. No check is made by the compiler to verify that the number is sufficient for the type of device because the device can be changed from disk to tape at run time via a SCOPE REQUEST card. If, at run time, the I/O routines determine that the buffer is less than one PRU of the device on which a file is residing, an I/O error abort will result. In some compilers, this number is used to denote the number of bits or bytes in an output record.

## FILE STATUS

A JOVIAL file may have a series of status constants to enable the program to check if the file is open, and ready for use, if an end-of-file was encountered on the last read, if a length error occurred on the last read, or if a parity error occurred on the last read (tape file only). These refer to logical records. The four format status constant values are: V(OPEN), V(EOF), V(LENGTH'ERR), V(PARITY). The user must specify at least one or up to four status constants. Any valid status constant may be used. Each will have the value of its respective status constant in the formal list, i. e., the first is V(OPEN), the second is V(EOF), the third is V(LENGTH'ERROR), and the fourth is V(PARITY). If only one constant of the four is to be used, the status constants that precede it in the list must also be declared, even though they will not be used. For example, if the third constant in the list is desired, constant values for V(OPEN) and V(EOF) must also be declared.

The condition under which the four formal constant values are set are:

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| V(OPEN)       | is set if the file is currently open and ready for use.                                                                                                                                                                                                                                                                                                                                                                           |
| V(EOF)        | is set if an end-of-file or in the case of a coded file assigned to INPUT, an end-of-record was detected during the last read.                                                                                                                                                                                                                                                                                                    |
| V(LENGTH'ERR) | is set if more words are requested in binary mode input than were available on the read, or if a record being read into a variable-length table did not contain an even multiple of the number of words per entry (NWDSSEN). The NENT of table is set to the number to indicate the entry that was partially read. Any I/O operation, OPEN, SHUT, INPUT, OUTPUT, or POS clears V(LENGTH'ERR), enabling further reads on the file. |
| V(PARITY)     | is set if a parity error occurred on the last read. A disk parity error will cause SCOPE to abort the job. If V(PARITY) is tested, it is cleared and further reads may be performed on the tape. This enables the user to discard bad records or to perform special processing on the records. If V(PARITY) is not tested, the job is aborted on the next read.                                                                   |

Examples:

- FILE HOL H 0 R 128 V(OK)  
V(EN'INPUT) INPUT \$  
File HOL, recorded in Hollerith mode, has variable length records specified in the file with a 128 word buffer size creating a 129-word circular buffer holding two disk PRUs, which enables processing to overlap the reading. The status constant, V(OK) determines if the file is open. The status constant V(EN'INPUT) is checked for end-of-file input. INPUT is the SCOPE file name.
- FILE BIN B 200 R 2048 V(OK)  
TAPE50 \$  
File BIN is a binary tape when TAPE50 is requested on a SCOPE REQUEST card prior to program execution. The file has 200 logical records recorded in binary mode. The buffer size (2048) creates a 2049 word buffer, which permits the use of a tape file with I/O processing overlapping. TAPE50 corresponds to FORTRAN unit 50 to be used by a FORTRAN subroutine called by the JOVIAL program.
- FILE AA H 1000 R 64 V(OPEN)  
PUNCH \$  
File AA is used to punch 1000 Hollerith mode cards. The buffer size (64) creates a circular buffer of 65 words, which is the minimum allowed for a disk file. The status constant, V(OPEN), will be set when the file is open and ready for use. PUNCH corresponds to the standard card punch file.
- FILE BCD H 0 V 128 V(O) V(E)  
V(L) BCD \$  
The JOVIAL file BCD has been assigned to a SCOPE file of the same name.
- FILE WORK B R 6040 V(READY)  
SCRATCH \$  
File WORK has a buffer size of 6041 words which would be adequate for a program having several I/O operations followed by processing with little I/O.

## FILE POSITIONING

The POS functional modifier can be used to obtain the current logical record position of a file or to position the file to a particular record position. The POS modifier may assume any value from zero through the number of logical records that the file can hold. A POS value of zero indicates that a file is rewound and positioned to read or write the first record of the file.

- There is no fixed relation between a JOVIAL logical record and physical records. The size of the physical record is determined by the type of storage device. One physical record may contain several logical records of the same or different lengths or several physical records may be used to hold one logical record. The programmer has no control over physical record positioning and need not concern himself with it.

POS can be used to position any file to a logical record or to obtain its logical record position.

- If POS is used for a file assigned to device name INPUT, OUTPUT, PUNCH, or PUNCHB, the file is positioned within the current SCOPE logical record.
- When an OPEN or SHUT is executed, the file is positioned to logical record location zero (that is, it is rewound), unless the file is for device name INPUT, OUTPUT, PUNCH, or PUNCHB. This decision is made at run time by the I/O interface routine (i. e., if file replacement is performed at run time the rewind or no rewind on OPEN or SHUT is based upon the actual file device in use and not that of the file declared in the file declaration).

### NOTE

It is the user's responsibility to consider this when doing run time file replacement.

- If POS is used on the right side of an assignment statement the current logical record position will be obtained, that is, the next record which would be read or written.

## FILE POSITIONING STATEMENT

The file-positioning statements are:

POS (file-name) = numeric-formula \$  
numeric-variable = POS (file-name) \$

where

file-name = File identifier corresponding to the file name used in the FILE declaration.

numeric-formula = Any value from zero to the maximum logical records of the file.

numeric-variable = User-defined variable.

The first statement above positions the file to the logical record specified by the numeric formula. The second statement records the current logical record position of the file by assigning the file position integer to the specified numeric variable.

The following statements have a particular meaning:

POS (file-name) = 0 \$

This statement rewinds the file, or, if the file has a device name INPUT, OUTPUT, PUNCH, or PUNCHB, positions the file to the beginning of the current SCOPE logical record.

POS (file-name) = POS (file-name) ± numeric-formula \$

This statement positions the file forward (+) or backward (-) the number of logical records corresponding to the value of the numeric formula.

Each time a record is read or written, the file position is advanced one record, so an INPUT or OUTPUT statement has an implied POS (file-name) = POS (file-name)+1 \$.

## ORGANIZATION OF DATA FOR TRANSFER

The names and structure of data in central memory is declared with a TABLE, ITEM, or ARRAY declaration; data on external storage devices must be organized in sequential files. The transfer of data is done in terms of full word or full word multiples. The transfer of data is exact; that is, no transformation of data takes place. If conversions are required, they must be accomplished by the program prior to output or after input.

- Records are input and output as full words or multiples of full words.
- If a bit or byte output is less than a full word, it is right-justified within a full word on output. If more than 10 bytes are output, they are left-justified within the maximum number of words required to contain the specific number of bytes.
- If an operand being output is less than a full word or multiple of a full word, the record is filled with blanks if it is a BCD record or zeros if it is a binary record.

- For BCD files, if the data record input is not long enough to fill the operand, the record is left-justified in the operand and blank-filled on the right.
- If the operand of a BCD file is not long enough for the record input, the record is left-justified and right-truncated.
- For binary files, if the logical record input is longer than the operand, the record is left-justified in the operand and right-truncated.
- For binary files, if the record input is shorter than the operand, it is left-justified within the record and the remainder of the operand is unchanged with V(LENGTH'ERR) set for the file status.
- Data is transferred exactly as it is on the file or in core; any conversion needed such as binary to Hollerith or from transmission code (STC) to Hollerith, would have to be effected by the program prior to transference of the data or after the transference. No conversion of data is performed during transmittal of data. The user must, therefore, perform any necessary conversion himself prior to reading or writing data.
- Numeric functional modifiers are right-justified in full words on output and the appropriate number of high order bits are stripped off and used on input.

All data forms generate compatible logical records on output and use the same method on input. Any legal assignment may be accomplished by outputting the formula and reading it back into the desired variable, for example:

```
data-form-b = data-form-a $
```

This could also be accomplished by:

```
OPEN OUTPUT SCRCH $
OUTPUT SCRCH data-form-a $
SHUT OUTPUT SCRCH $
OPEN INPUT SCRCH $
INPUT SCRCH data-form-b $
```

## DATA FORMS

The effects of using the various forms of data organization when data is transferred are described below.

### SIMPLE ITEMS

In the case of simple items, the entire word or words containing the item in core are output. Partial word items are right-justified within the word; literals longer than a word are left-justified within the minimum number of words necessary to contain the literal. Anything occupying the remainder of the word is output also.

For input to a simple item, the logical record input is read into the full word of the simple item.

### ARRAYS

An entire array is input or output as a single record.

Single array items are input and output the same as simple items, except for the required subscript.

### TABLE ITEMS

Packed table items are moved to a temporary storage area where they are arranged in a format as if they had been declared as simple items. The minimum number of full words needed to contain the item are then written out. Conversely input is done to a temporary storage area which is then moved to the proper position within the table, stripping off any portion of the full word multiple that is not required for storage. The maximum size of a literal which can be output or input from a packed table is 150 words.

Unpacked table items are input and output the same as a single array item.

### PARALLEL TABLES

All table or entry output is done in terms of serial table entries. If a parallel table or table entry is output, it is converted to serial entry format in a temporary storage area before being output. Conversely, when entries are input to a parallel table, they are moved to a temporary storage area and then converted to parallel format. A parallel table is limited to 150 words per entry. Thus, a parallel table can be written out and read back into a compatible serial table or a serial table can be output and read back into a compatible parallel table.

### VARIABLE TABLES

Only the number of entries specified by the NENT are output for variable tables so it is necessary that the NENT be set to the proper value before an output.

- On output entries, Zero to NENT -1 are output.

- Parallel to serial conversion is done on the output if required.
- On input, the NENT of a variable table is set to the number of entries read in. If the record read in does not contain a number of words which is a multiple of the number of words per entry, the NENT of a variable table is set to the entry which was only partially read in. As with output, the serial to parallel conversion is effected if necessary.

### TABLE ENTRIES

Table entries are output as one record for each entry and may be output as one entry or several, depending upon the number of entries or range of NENT in the table. The range of NENT-1 is from entry 0 to the maximum number of entries specified in the declaration.

### TABLES

Tables output the record NENT first, followed by the number of entries specified in NENT for rigid-length tables or 0 through NENT -1 for variable-length tables.

On input of rigid-length tables, the NENT of the table is read followed by the number of specified entries. For variable-length tables, NENT and the entries that follow are read into entries 0 through NENT-1.

The status constant, V(LENGTH'ERR) (see File Declaration), will be set if a record being read into a variable-length table did not contain an even multiple number of words per entry. The NENT of the table is set to the number of the entries only partially read.

## **INPUT/OUTPUT STATEMENTS**

Transmission of data between central memory and external files is accomplished by the JOVIAL input and output statements. A file must be opened before read or write operations can be performed; it can be opened for input or output, but not for both simultaneously. If a file is opened, it must be closed before it may be used in the opposite way. The transfer of logical records is accomplished by the INPUT or OUTPUT statements. It is advisable to shut all files before terminating execution of the program.

The statements that control input data are OPEN INPUT to open the file for input; INPUT to transfer data to central memory from the operand files, and SHUT INPUT to close the file when all the data is transferred. External files are named by the FILE declaration, and all I/O statements must include the file name. The statements that control output data are OPEN OUTPUT to open the file to receive output; OUTPUT to write the data from main storage onto an operand output file, and SHUT OUTPUT to close the output file when output is complete.



## **OPENING FILES**

Prior to transfer of data, JOVIAL files must be opened by an OPEN INPUT or an OPEN OUTPUT statement.

### OPEN INPUT STATEMENT

This statement activates the file for input and positions the file to logical record zero. The form is:

OPEN INPUT file-name \$

where:

file-name = Name designated for the file by the user in the file declaration.

### OPEN OUTPUT STATEMENT

This statement activates the file for output and positions the file to logical record zero.

OPEN OUTPUT file-name \$

where

file-name = File identifier name designated by the user.

## **TRANSFERRING DATA**

The opening of a file for input or output does not in itself transfer data; after the file has been opened, an INPUT or OUTPUT statement must be issued in order to transfer data.

### INPUT STATEMENT

This statement transfers data to the central memory. The form is:

INPUT file-name operand \$

where

file-name = Name designated for the file by the user in the file declaration.

operand = Defines the data structure in central memory (see Table 6-1).

The INPUT statement reads into the specified operand the logical record at which the file is positioned. After the record is read, the file is positioned at the start of the next logical record and the POS operator is incremented by one.

TABLE 6-1. OPERAND FORMS FOR INPUT STATEMENTS

| Operand Type <sup>†</sup>                                                                                                                                                                                                    | Meaning                                                                                                                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Single item name                                                                                                                                                                                                             | Transfer data to a single simple item                                                                                                                                                                                                      |
| Table item name (\$ index \$)                                                                                                                                                                                                | Transfer data to a single table item (including packed)                                                                                                                                                                                    |
| Array name (\$ index \$)                                                                                                                                                                                                     | Transfer data to a single element of an array                                                                                                                                                                                              |
| Array name                                                                                                                                                                                                                   | Transfer data to all elements of an array                                                                                                                                                                                                  |
| Table name                                                                                                                                                                                                                   | Transfer data to all entries of a table. The data is assumed in serial table order so it is shuffled by the compiler-generated code for parallel tables with NWDSSEN>1. After input, NENT is set to number of words transferred/NWDSSEN-1. |
| Table name (\$ index \$)                                                                                                                                                                                                     | Transfer data to a single entry of the table                                                                                                                                                                                               |
| Table name<br>(\$ index <sub>1</sub> ... index <sub>2</sub> \$)                                                                                                                                                              | Transfer data to consecutive entries of the table; the first entry being defined by index <sub>1</sub> , the last by index <sub>2</sub> , inclusive.                                                                                       |
| BIT (\$ index \$) (variable)                                                                                                                                                                                                 | Transfer fullword integer value to substring                                                                                                                                                                                               |
| BYTE (\$ index \$) (variable)                                                                                                                                                                                                | Transfer necessary number of bytes <sup>††</sup> to substring                                                                                                                                                                              |
| POS (file-name)                                                                                                                                                                                                              | Transfer fullword integer to reset the value of the file position.                                                                                                                                                                         |
| NENT<br>(variable-length-table-name)<br>(variable-length-item-name)                                                                                                                                                          | Transfer fullword integer to reset NENT of specified table.                                                                                                                                                                                |
| {ENTRY}<br>{ENT }<br>( {table-name} (\$ index \$) )                                                                                                                                                                          | Transfer number of words that is NWDSSEN for this table into a single entry of the table                                                                                                                                                   |
| {CHAR}<br>{MANT}<br>{ODD } (variable)                                                                                                                                                                                        | Transfer value into variable functional modifier                                                                                                                                                                                           |
| <sup>†</sup> Only literal variables are legal operands for Hollerith file operations.<br><sup>††</sup> Number of bytes determined by second index component; for example, INPUT XX BYTE(\$1, J\$)(XYZ) \$ transfers J bytes. |                                                                                                                                                                                                                                            |

### OUTPUT STATEMENT

This statement transfers data from central memory to the file specified by this statement.

OUTPUT file-name operand \$

where:

file-name = File identifier designated by the user

operand = Valid forms of operands are the same as for INPUT with the addition of constant and NENT of a rigid length table (see Table 6-2).

OUTPUT writes the contents of the operand as a logical record in the named file. The record is written at the position where the file is at the time OUTPUT is specified; the POS of the file is incremented by one after the output.

### **CLOSING FILES**

At the conclusion of the file usage, the file must be closed by a SHUT INPUT or SHUT OUTPUT statement. SHUT statements are also used to reverse the usage of a file. That is, a file that is opened for input must be closed before it can be used for output; the sequence would be: OPEN INPUT, INPUT, SHUT INPUT, OPEN OUTPUT, OUTPUT, SHUT OUTPUT.

A file that is opened for output must be closed before it can be used for input; the sequence would be: OPEN OUTPUT, OUTPUT, SHUT OUTPUT, OPEN INPUT, INPUT, SHUT INPUT.

The only legal references to a file that has been shut are OPEN INPUT, OPEN OUTPUT or the test, IF FILE EQ V(OPEN).

### SHUT INPUT STATEMENT

This statement closes the file; any subsequent references to it, except another OPEN statement or test case V(OPEN), is illegal. The form is:

SHUT INPUT file-name \$

where:

file-name = Name of an active file designated by the user.

TABLE 6-2. OPERAND FORMS FOR OUTPUT STATEMENTS

| Operand Type †                                                                                                                                                                                                                                                                                                    | Meaning                                                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Single item name                                                                                                                                                                                                                                                                                                  | Transfer data from a single simple item                                                                                                               |
| Table item name (\$ index \$)                                                                                                                                                                                                                                                                                     | Transfer data from a single table (including packed) item                                                                                             |
| Array name (\$ index \$)                                                                                                                                                                                                                                                                                          | Transfer data from a single element of an array                                                                                                       |
| Array name                                                                                                                                                                                                                                                                                                        | Transfer data from all elements of an array                                                                                                           |
| Table name                                                                                                                                                                                                                                                                                                        | Transfer data from all entries of the table using NENT. The data is output in serial table order.                                                     |
| Table name (\$ index \$)                                                                                                                                                                                                                                                                                          | Transfer data from a single entry of the table                                                                                                        |
| Table name (\$ index <sub>1</sub> ... index <sub>2</sub> \$)                                                                                                                                                                                                                                                      | Transfer data from consecutive entries of the table; the first entry being defined by index <sub>1</sub> , the last by index <sub>2</sub> , inclusive |
| BIT (\$ index \$) (variable)                                                                                                                                                                                                                                                                                      | Transfer fullword integer value from a substring <sup>††</sup>                                                                                        |
| BYTE (\$ index \$) (variable)                                                                                                                                                                                                                                                                                     | Transfer necessary number of bytes <sup>†††</sup> from a substring <sup>††</sup>                                                                      |
| POS (file-name)                                                                                                                                                                                                                                                                                                   | Transfer fullword integer from the value of the file position                                                                                         |
| NENT { table-name<br>{ table-item-name } }                                                                                                                                                                                                                                                                        | Transfer fullword integer from NENT of specified table                                                                                                |
| { ENTRY }<br>{ ENT }                                                                                                                                                                                                                                                                                              | Transfer number of words (NWDSSEN) for this table from a single entry of the table                                                                    |
| ( { table-name } (\$ index \$) )                                                                                                                                                                                                                                                                                  |                                                                                                                                                       |
| { NWDSSEN ( { table-name<br>{ table-item-name } } ) }                                                                                                                                                                                                                                                             | Transfer constant to file                                                                                                                             |
| { CHAR }<br>{ MANT } (variable) <sup>†</sup><br>{ ODD }                                                                                                                                                                                                                                                           | Transfer fullword value                                                                                                                               |
| <p>†Only literal variables and constants are legal operands for Hollerith file operations.</p> <p>††if the number of bits or bytes is equal to zero, write zero-length record.</p> <p>†††Number of bytes determined by second index component; for example, OUTPUT XX BYTE(\$I, J\$(XYZ)\$ transfers J bytes.</p> |                                                                                                                                                       |

### SHUT OUTPUT STATEMENT

This statement writes an end-of-file on the output file.

```
SHUT OUTPUT file-name $
```

where:

```
file-name = Programmer-supplied file identifier
```

### **SHORT FORMS**

There are short forms for two of the INPUT statements and two of the OUTPUT statements. They are:

- To open a file and transfer the first record, the form is:

```
OPEN INPUT file-name operand $
```

- To read the last record and close the file, the form is:

```
SHUT INPUT file-name operand $
```

- To open a file and write the first record, the form is:

```
OPEN OUTPUT file-name operand $
```

- To write the last record and close the file, the form is:

```
SHUT OUTPUT file-name operand $
```

### **EXAMPLES OF INPUT/OUTPUT DATA FORMS**

The following program is intended to give examples of the various JOVIAL input/output operands, it is not an actual working program. The JOVIAL compiler provides compatibility of items of the same type, regardless if they are simple items, table or array items, or functional modifiers. JOVIAL also provides compatibility between tables and table entries for serial and parallel tables.

```
''
FILE SCRATCH B O R 65 V(OK) JWF $
''
''DECLARE SIMPLE ITEMS''
ITEM SIMP'LIT H 4 $
ITEM SIMP'INT I 18 S $
ITEM SIMP'FIX A 18 S 7 $
ITEM SIMP'FLT F $
ITEM SIMP'BOOL B $
ITEM SIMP'STAT S V(AA) V(BB) V(CC) V(DD) V(EE) V(FF) V(GG) $
```

```

ITEM SIMP'LNG'LIT H 17 $
''
''DECLARE ARRAY ITEMS''
ARRAY ARY'LIT 10 H 4 $
ARRAY ARY'INT 10 I 18 S $
ARRAY ARY'FIX 10 A 18 S 7 $
ARRAY ARY'FLT 10 F $
ARRAY ARY'BOOL 10 B $
ARRAY ARY'STAT 10 S V(XA) V(XB) V(XC) V(XD) V(XE) V(XF) V(XG) $
ARRAY ARY'LGN'LIT 10 H 17 $
ARRAY ARY'LIT'TWO 10 H 4 $
''
''DECLARE TABLE ITEMS''
TABLE TAB V 75 S 4 $
BEGIN
ITEM TAB'LIT H 4 00 0 $
ITEM TAB'INT I 18 S 0 24 $
ITEM TAB'FIX A 18 S 7 0 42 $
ITEM TAB'FLT F 2 0 $
ITEM TAB'STAT S V(A) V(B) V(C) V(D) V(E) V(F) V(G) 3 0 $
ITEM TAB'BOOL B 3 13 $
ITEM TAB'LNG'LIT H 17 3 18 $
END
TABLE TABP P L $
''
''ONE WORD LITERAL I/O OPERANDS''
LABEL.
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH TAB'LIT(5) $
OUTPUT SCRATCH 3H(OUT) $
OUTPUT SCRATCH ARY'LIT (7) $
OUTPUT SCRATCH BYTE(12.5(SIMP'LNG'LIT) $
OUTPUT SCRATCH SIMP'LIT $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH ARY'LIT(2) $
INPUT SCRATCH BYTE(11.3(TAB'LNG'LIT(4)) $
INPUT SCRATCH SIMP'LIT $
SHUT INPUT SCRATCH $
''
''INTEGER I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH SIMP'INT $
OUTPUT SCRATCH TAB'INT(3) $
OUTPUT SCRATCH 50 $
OUTPUT SCRATCH BIT(5.15(TAB'LNG'LIT(3)) $
OUTPUT SCRATCH NWDSSEN(TAB) $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH ARY'INT(5) $
INPUT SCRATCH SIMP'INT $
INPUT SCRATCH POS(SCRATCH) $
INPUT SCRATCH CHAR(SIMP'FLT) $
SHUT INPUT SCRATCH $
''
''FIXED POINT I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH MANT(SIMP'FLT) $
OUTPUT SCRATCH SIMP'FIX $
OUTPUT SCRATCH ARY'FIX(9) $

```

```

SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH ARY'FIX(3) $
INPUT SCRATCH TAB'FIX(8) $
INPUT SCRATCH TAB'FIXP(8) $
SHUT INPUT SCRATCH $
''
'' FLOATING POINT I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH 32.30 $
OUTPUT SCRATCH SIMP'FLT $
OUTPUT SCRATCH TAB'FLT(7) $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH ARY'FLT(1) $
SHUT INPUT SCRATCH $
''
'' BOOLEAN I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH TAB'BOOL(7) $
OUTPUT SCRATCH ARY'BOOL(5) $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH SIMP'BOOL $
INPUT SCRATCH ODD(SIMP'INT) $
SHUT INPUT SCRATCH $
''
'' STATUS I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH ARY'STAT(5) $
OUTPUT SCRATCH SIMP'STAT $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH TAB'STAT(8) $
SHUT INPUT SCRATCH $
''
'' LONGER THAN ONE WORD LITERAL I/O OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH SIMP'LNG'LIT $
OUTPUT SCRATCH TAB'LNG'LIT(0) $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH TAB'LNG'LIT(0) $
SHUT INPUT SCRATCH $
''
'' TABLE AND ARRAY OPERANDS''
OPEN OUTPUT SCRATCH $
OUTPUT SCRATCH TAB $
OUTPUT SCRATCH ENT(TABP(3)) $
OUTPUT SCRATCH ARY'LIT $
OUTPUT SCRATCH TAB($3...7$) $
SHUT OUTPUT SCRATCH $
OPEN INPUT SCRATCH $
INPUT SCRATCH TABP $
INPUT SCRATCH ENTRY(TAB(4)) $
INPUT SCRATCH ARY'LIT'TWO $
INPUT SCRATCH TABP($1...5$) $
SHUT INPUT SCRATCH $
TERM $

```

## OBJECT-TIME EXECUTION WITH I/O

The user can change the main program file-device names at object-time by specifying the new names as parameters on the execution control card. The new names may be any SCOPE file of up to seven characters. Since the compiler uses only the first six characters of a device-name, file replacement is the only way in which a seven character file name may be used by a JOVIAL program.

- Comma separators should be used to logically space the parameter list on the execution control card.
- Device-name not to be replaced must be indicated by a null parameter
- The rules for file usage previously declared apply to the file actually used. Any special treatment of files, such as INPUT, is determined at run time. It is the user's responsibility to see that replacing a file at run time does not cause a conflict with the way in which the file is used by the program.

The effect of file replacement is the same regardless of the file from which the program is loaded.

## EXECUTION CONTROL CARD

The loader call

```
LGO (, ,TAPE1, MASTER2)
```

causes the relocatable object program on file LGO to be link-edited. The first two files will remain the same as specified at compile time. The name in the file environment table (FET) of the third file will be replaced with TAPE1 and the fourth file will be replaced with MASTER2. Execution will begin with the last transfer address specified in LGO.

## FORTRAN-FORMATTED OUTPUT

The JOVIAL user can output FORTRAN formatted data by means of four FORTRAN library routines. The data may be output to any SCOPE file specified as a device name (not a file name) in a JOVIAL file declaration or the system file OUTPUT. The routines are PRINT, PRINTF, LIST, and ENDL (see the test program examples at the end of this section). The following rules apply to the use of these routines:

- At least two of these routines must be called to perform output with formatted conversion.



- The first call may be PRINT (which passes the format to be used) or PRINTF (which passes the format to be used and the FET of the file to be written on).
- Constants or variables to be converted are then passed by calling the routine LIST once for each variable or constant to be converted.
- If the output is entirely Hollerith, the output can be specified in the format passed to PRINT or PRINTF so LIST need not be called.
- The output process is terminated by a call to ENDL.

The four routines are described in Table 6-3.

The library routine PRINT does not automatically create a local file which will be assigned to the system file OUTPUT at the termination of the job. It is the responsibility of the programmer to insure that a file is available on which PRINT will write. This can be done by a file declaration statement indicating the name OUTPUT for a device name or by the M, MONITOR, option on the JOVIAL control card. Execution of a call to PRINT without providing a file will result in an abort.

If the conversion format is specified in a literal variable, the same format may be used for several PRINT or PRINTF calls, saving coding effort. The use of a variable to hold the format also permits it to be changed at run time. This may be done by an assignment statement or by reading in a new format from an external file.

TABLE 6-3. FORMATTED OUTPUT ROUTINES

| Routine | Definition                                                                                                                 | Parameters                                                                                                                                                                                                  |
|---------|----------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PRINT   | Initialize the output by supplying the format to the FORTRAN library output routine for writing on the system file OUTPUT. | Hollerith constant or the name of a Hollerith variable containing the legal FORTRAN format.                                                                                                                 |
| PRINTF  | The same as PRINT except the output file is the one specified in the second parameter.                                     | The first parameter is the same as for PRINT. The second is LOC(FILE) or variable containing the LOC of the file to be used. The name must be a file name specified in a JOVIAL file declaration statement. |
| LIST    | Conveys data value to FORTRAN output routines.                                                                             | Any legal constant or variable contained in one word and suitable for FORTRAN conversion.                                                                                                                   |
| ENDL    | Terminate output.                                                                                                          | No parameters.                                                                                                                                                                                              |

Since the LOC of the file is passed as a parameter in the call to PRINTF, it may be changed at run time by using a variable and changing the value of the variable to the LOC of the desired file prior to calling PRINTF.

Output of FORTRAN formatted data is subject to the following restrictions:

- Variables to be output must be subject to FORTRAN conversion routines and must be contained in one word.
- Integer items can be any size and signed or unsigned.
- Fixed-point items can not be output.
- Floating-point items are acceptable.
- Hollerith variables must be ten characters or less.
- JOVIAL stores partial word literals right-justified; FORTRAN partial words are stored left-justified, so the R format should be used to output Hollerith items of less than 10 characters.
- To output literals longer than 10 characters, overlay the literal with an array of one dimension and 10 characters. Call LIST once for each word to be output.
- Table items and BIT or BYTE modifications will be properly removed to a temporary storage space and output, provided they meet the length restrictions given for simple items.
- The maximum length possible for FORTRAN output is 150 characters.

Refer to the FORTRAN Extended Reference Manual for details on legal formats.

#### FORMATTED OUTPUT EXAMPLES

The following is a listing of several test examples of the four FORTRAN output routines. The following SCOPE control cards were used to generate these examples:

```

 JOVIAL(F,M)
 LGO.
 REWIND(F1,F2)
 COPYSBF(F1,OUTPUT)
 COPYSBF(F2,OUTPUT)
 START $ 'DEMONSTRATE PRINT PRINTF LIST ENDL'
1 'USE OF FORMAT IN A LITERAL VARIABLE'
 ITEM FORM 7H((3X,I3)) $
 PRINT (FORM) $
 LIST (50) $
 ENDL $

```

} SCOPE Control Cards

↑ Program Reference Numbers

```

'' ''
2 ''PRINT 5 CHARACTER LIT VAR THROUGH A10 FORMAT''
 ITEM LIT H 5 $
 ITEM HLL H 10 $
 OVERLAY LIT = HLL $
 LIT = 5H(56789) $
 BYTE($0,2$(HLL)) = 2H(HI) $
 PRINT (7H((X,A10))) $
 LIST (LIT) $
 ENDL$

'' ''
3 ''10 CHARACTER LITERAL THROUGH 5 CHARACTER FORMAT''
 ITEM HOLL 10H(0123456789) $
 PRINT (6H((X,A5))) $
 LIST (HOLL) $
 ENDL$

'' ''
4 ''PRINT 5 CHARACTER LITERAL THROUGH R10 FORMAT''
 PRINT (6H((X,R5))) $
 LIST (HOL) $
 ENDL $

'' ''
5 ''LESS THAN FULL WORD INTEGER''
 ITEM FMT 8H((3X,I10)) $
 ITEM INT1 I 18 S $
 ITEM INT2 I 37 S $
 ITEM INT3 I 37 U $
 ITEM DDD I 59 S P 0 $
 OVERLAY DDD = INT3 $
 INT1 = 100 $
 INT2 = 500 $
 PRINT(FMT) $
 LIST (INT1) $
 LIST (INT2) $
 ENDL $

'' ''
6 ''UNSIGNED INTEGER''
 ITEM USI I 5 U $
 USI = 31 $ '' SET ALL FIVE BITS''
 PRINT(FMT) $
 LIST (USI) $
 ENDL $

'' ''
7 ''PRINT OUT 5 WORD LITERAL USING ARRAY OVERLAY''
 ARRAY DUM 5 H 10 $
 ITEM FIVE 45H (45 CHARACTER LITERAL OUTPUT BY USING ARRAY) $
 OVERLAY FIVE = DUM $
 PRINT (6H((5A10))) $
 FOR Z = 0,1,4 $
 LIST (DUM(Z)) $
 ENDL $

'' ''
8 ''PRINT FROM INTEGER ARRAY''
 ARRAY INTEGER 10 I 60 S $
 BEGIN 0 -100 200 -300 400 -500 600 -700 800 -900 END
 PRINT(7H((10I10))) $
 FOR C = 0,1,9 $

```

Program Reference Numbers

```

LIST (INTEGER(C)) $
ENDL $
'' ''
9 ''PACKED TABLE ITEMS''
TABLE TAB V 10 S 2 $
BEGIN
ITEM HH H 13 0 00 $ BEGIN 13H(ABCDEFGHJKLM) 13H(NOPQRSTUVWXYZ)
13H(0123456789012) END
ITEM H1 H 10 0 00 $
ITEM H2 H 3 1 00 $
ITEM II I 18 U 1 18 $ BEGIN 000 100 200 300 400 500 600 END
ITEM IS I 12 S 1 36 $ BEGIN -023 +123 -323 +423 -523 END
ITEM IU I 12 U 1 48 $ BEGIN 056 156 256 356 END
END
PRINT (20H((3X,I10,A10,R3,2I10))) $
LIST(II(1)) $
LIST (H1(1)) $
LIST (H2(1)) $
LIST (IS(1)) $
LIST (IU(1)) $
ENDL $
'' ''
10 ''PACKED TABLE ITEMS USING A FOR LOOP''
FOR A = 0,1,2 $
BEGIN
PRINT (20H((3X,I10,A10,R3,2I10))) $
LIST(II(A)) $
LIST(H1(A)) $
LIST(H2(A)) $
LIST (IS(A)) $
LIST (IU(A)) $
ENDL $
END
'' ''
11 ''BIT MODIFIER ON A ONE WORD ITEM''
ITEM BT H 10 $
BIT($0,59$(BT)) = 0(00000000000000000000) $
BIT($15,8$(BT)) = 100 $
PRINT(FMT) $
LIST (BIT($15,8$(BT))) $
ENDL $
'' ''
12 ''BYTE MODIFIER ON A ONE WORD ITEM''
ITEM HWD 10H(ABCDEFGHJIJ) $
PRINT (7H((X,A10))) $
LIST (BYTE($5,2$(HWD))) $
ENDL $
'' ''
13 ''BYTE MODIFIER ON ONE WORD ITEM USING R FORMAT''
PRINT (6H((X,R2))) $
LIST (BYTE($5,2$(HWD))) $
ENDL $
'' ''
14 ''USE PRINTF WITH LOC(FILE'NAME)''
FILE FILE1 H 0 R 80 V(OK) F1 $
ITEM MES1 30H(THIS WAS PRINTED ON FILE1) $
ARRAY ARY1 5 H 10 $

```

↑ Program Reference Numbers

```

OVERLAY MES1 = ARY1 $
 PRINTF (6H((3A10)) , LOC(FILE1)) $
 FOR A = 0,1,2 $
 LIST (ARY1($ A $)) $
 ENDL $
'' ''
15 ''USE PRINTF WITH A VARIABLE SET TO THE LOC (FILE'NAME)''
 FILE FILE2 H 0 R 80 V(OK) F2 $
 ITEM MES2 30H(THIS WAS PRINTED ON FILE2) $
 ARRAY ARY2 5 H 10 $
 OVERLAY MES2 = ARY2 $
 ITEM FET I 18 S $
 FET = LOC(FILE2) $
 PRINTF (6H((3A10)) , FET) $
 FOR B = 0,1,2 $
 LIST (ARY2($ B $)) $
 ENDL $

TERM $
NUMBER LINE MESSAGE

JOV502 (CPL) COMPOOL-DEFINED SYMBOLS
 ** NONE **

JOV503 (CPL) MODE-DEFINED SYMBOLS
 ** NONE **

** NO DIAGNOSTIC MESSAGES**

1 50
2 HI 56789
3 01234
4 56789
5 100
6 500
7 31
8 45 CHARACTER LITERAL OUTPUT BY USING ARRAY
9 0 -100 200 -300 400 -500 600 -700 800 -900
10 100NOPQRSTUVWXYZ 123 156
11 0ABCDEFGHIJKLM -23 56
12 100NOPQRSTUVWXYZ 123 156
13 2000123456789012 -323 256
14 100
15 FG
16 FG
17 THIS WAS PRINTED ON FILE1
18 THIS WAS PRINTED ON FILE2

```

```

03/05/71 **SCOPE**3.3 PSR247 8/15/70
17.50.05.PRNT03T
17.50.05.PRNT,CM70000,T77,P77.
17.50.05.JOVIAL(F,M)
17.50.14.MAP(OFF)

```

Program Reference Numbers

17.50.14.LGO.  
17.50.18.END MAINPG  
17.50.18.REWIND(F1,F2)  
17.50.18.COPYBF(F1,OUTPUT)  
17.50.19.COPYBF(F2,OUTPUT)  
17.50.19.CP 002.457 SEC.  
17.50.19.PP 005.587 SEC.  
17.50.19.IO 001.633 SEC.

---

A processing declaration causes the compiler to generate a closed set of code which can be invoked from elsewhere within the program. Processing declarations define switches and closed forms (CLOSE routines, procedures, and functions).

The legality of a processing declaration depends on the type of declaration and the area from which it is invoked. The following restrictions apply:

- A processing declaration can not be invoked by itself.
- Unless it causes a change in sequence of control, it will turn over control to the statement following the one from which it was invoked.
- Processing declarations can generally be declared anywhere in the program.
- The compiler collects code produced by closed forms, places it ahead of that from the remainder of the program, and provides jumps around the code produced by switch declarations.

### SWITCHES

A switch is a single JOVIAL statement providing a series of decision points to which control can be transferred. It can appear anywhere in the program; it is invoked by a GOTO followed by a switch name.

There are two types of switches: the index switch activated according to the value of an index and the item switch which bases decisions on the value of an item.

#### INDEX SWITCH DECLARATION

The format of the index switch declaration is:

```
SWITCH switch-name = (index-list) $
```

where:

```
switch-name = programmer-supplied name identifying the switch.
```

index-list = list of names separated by commas; each name is either a statement name, a CLOSE name, or another switch name followed by an index. The same name can appear more than once in a list. The names are sequence designators to which control passes depending on the position of the name in the list and the index value of the switch call. Null switch points can be specified by adjacent commas (,,) with no named switch point between them.

Examples:

- SWITCH CHOICE = (J1, J2, SW(\$I\$), LAST) \$  
CHOICE is the name of the switch; J1, J2, and LAST are statement names or CLOSE names; SW must be a switch name.
- SWITCH EITHER = (ONE, ONE,, TWO, TWO, ONE, TWO,, TWO) \$  
In switch EITHER, only the switch points ONE or TWO can be selected. The third and eighth positions are null switch points.

#### INDEX SWITCH CALL

The format of an index switch call is:

GOTO switch-name (\$index\$)\$

where:

switch-name = programmer-supplied name identifying the switch.  
index = any numeric formula

The following rules apply to an index switch call:

- If the formula does not yield an integer result, it is truncated to an integer value.
- The value of the index determines which switch point is selected. A value of 0 selects the first switch point; a value of n-1 selects the last switch point.
- If the value of the index is negative, equal to or greater than the number of switch points, or if a null switch point is selected, control is transferred to the statement following the switch call.
- If a switch cannot be executed (for instance, there is a subsequent undefined switch call), control transfers to the statement following the call.
- A switch point containing a close name could cause execution to return to the statement following the switch call after the close routine is executed, which would be the same as if the point of control had fallen through.



Examples:

- **GOTO CHOICE (\$KK\$)\$** The switch is defined in the first index switch example shown above. If the value of KK is 0, control transfers to J1; if the value is 1, to J2; and if the value is 3, to LAST. If the value is 2, control is transferred via switch SW according to the value of index (I).
- **GOTO EITHER (\$BB-AA\*\*2\$)\$** The switch is defined in the second index switch shown above. The value 0, 1, and 5 will transfer control to ONE; values 3, 4, 6 and 8 will transfer control to TWO. Other values will not transfer control, hence, the next statement would be executed.

#### ITEM SWITCH DECLARATION

In an item switch, the current value of the item specified in the switch declaration is compared with a list of constants until an equality is found or the list is exhausted. Control passes to the switch point corresponding to the first constant equal to the specified item. The format is:

**SWITCH** switch-name (switch-item) = item-switch-list) \$

where:

**switch-name** = programmer-supplied identifier  
**switch-item** = simple or table item name, array name, file name  
**item-switch-list** = string of pairs of names and constants linked by an equal sign in the format: constant = name. The pairs are separated by commas; there may not be two commas in succession.

The following rules apply to item switch declarations:

- The constants can be numeric literal, Boolean, or status. If the switch item is a file name, the constant must be a status constant.
- The switch item and the constants in the item switch list must be of the same type.
- The names are statement names, CLOSE names, or switch names; they are the switch points to which transfer can be made.

Examples:

- SWITCH EITHER (TOSS) =  
(0=ONE, 1=ONE, 5=ONE,  
3=TWO, 4=TWO, 6=TWO) \$
- SWITCH TYPE (DATA) =  
(1H(I) = NUMERIC,  
1H(A) = NUMERIC,  
1H(F) = NUMERIC,  
1H(T) = LITERAL,  
1H(H) = LITERAL,  
1H(B) = BOOLEAN,  
1H(S) = STATUS) \$
- SWITCH SW1000 (PIN1) =  
(V(AA) = PIN2, V(BB) = PIN2,  
V(CC) = PIN2, V(DD) = PIN3,  
V(EE) = PIN2, V(FF) = PIN2) \$
- SWITCH SWCH (ARY) =  
(4H(ZERO) = ZERO,  
3H(ONE) = ONE,  
3H(TWO) = TWO,  
5H(THREE) = THREE) \$

In switch EITHER, switch item TOSS is an integer item. If its value is 0, 1, or 5 when EITHER is activated, control is transferred to ONE; if the value of TOSS is 3, 4, or 6, control is transferred to TWO; otherwise, control transfers to the statement following the switch call.

A call is made to the switch TYPE to test the contents of a simple one-character display code item. If DATA is I, A, or F, control is transferred to NUMERIC; if T or H, control is transferred to LITERAL, and so forth. If no equivalence is found for the value of DATA, control is transferred to the statement following the switch call.

Item PIN1 must have been declared as a status item along with the six status constants used in the switch. If its value is AA, BB, CC, EE, or FF control is transferred to PIN2; if the value of PIN1 is DD, control is transferred to PIN3; otherwise, control is transferred to the statement following the switch call.

Item ARY must have been declared as a Hollerith array. When this item switch is called a subscript must be specified to indicate which item in the array is to be used for the compare with the switch values.

#### ITEM SWITCH CALL

The call to an item switch is GOTO followed by the switch name. The switch name is indexed only if the switch item is a table or array item. The format is:

GOTO switch-name [(\$index\$)] \$

where:

switch-name = programmer-defined identifier

The following rules apply to item switch calls:

- When the switch name is indexed, the index must have the appropriate number of occurrences to select the particular table or array item to be compared with the list of constants.
- If the switch point is a CLOSE or a switch name, control can eventually transfer back to the statement following the switch call, depending on the declaration of the CLOSE or switch, which would be the same as if the point of control had fallen through.

Examples:

- GOTO EITHER \$  
This call invokes the switch EITHER previously declared. It is not indexed because TOSS is a simple item.
- GOTO TYPE \$  
GOTO ERROR \$  
This call invokes the switch, TYPE, previously declared. If the value of DATA is found not to be equivalent, control is transferred to the next statement following the switch call.
- GOTO SW1000 \$  
This call invokes the switch, SW1000, previously declared. If the status variable is set to V(AA), V(BB), V(CC), V(EE), or V(FF) control will transfer to PIN2. If it is set to V(DD) transfer is made to PIN3. If the status variable is set to any other value, the next statement will be executed.
- GOTO SWCH (\$3\$) \$  
This call invokes the switch, SWCH, previously declared. The index (\$3\$) indicates that the switch points are to be compared with ARY(\$3\$).

A switch is more straightforward than an IFEITH ORIF statement. For example, the sequence of control shown in the second example of item switch calls could be accomplished by the following IFEITH ORIF alternative:

```
IFEITH DATA EQ 1H(I) OR DATA EQ 1H(A) OR DATA EQ 1H(F) $
 GOTO NUMERIC $
 ORIF DATA EQ 1H(H) OR DATA EQ 1H(T)$
 GOTO LITERAL $
 ORIF DATA EQ 1H(B)$
 GOTO BOOLEAN $
 ORIF DATA EQ 1H(S)$
 GOTO STATUS $
 ORIF 1 $
 GOTO ERROR $
 END
```

## CLOSED FORMS

Closed forms are one or more JOVIAL statements and related declarations which need only be specified once, but may be used at various points within the program. Invoking a closed form usually entails transfer of control to the form, execution of the closed form, and a return to the statement following the one which caused the transfer of control. However, the execution of a closed form can cause a transfer of control to statements other than the next statement. There are three types of closed forms: CLOSE routines, procedures, and functions.

- Procedures and functions can have input and output parameters and involve a change in the scope of definition. They are invoked by the use of the procedure or function name with the actual parameters to be used during its execution.
- CLOSE routines cannot have any input or output parameters and do not change the scope of definition. They are invoked by a GOTO with the name of a CLOSE.

The compiler isolates the code produced by closed forms and places it ahead of any code produced by statements outside of closed forms.

A closed form may call other closed forms, but may not call itself or a closed form which would invoke it. Closed forms involving a change of scope can not be nested; a JOVIAL program can have only one level of change of scope. A CLOSE can be declared inside of a procedure or function, or a procedure or function can be declared inside of a CLOSE because a CLOSE does not cause a change in the scope of definition. A CLOSE can have other CLOSEs nested in it to any level.

### CLOSE ROUTINE

A CLOSE routine has no parameters, so the data manipulated is contained in variables whose names are common to the CLOSE routine and the main routine or procedures from which a

CLOSE is called. Data can be declared within a CLOSE, but it is usually declared in the main routine or procedure in which the CLOSE is contained. CLOSE routines are invoked directly by a GOTO statement or indirectly by a switch call. A CLOSE has the same scope as the section of the program where it is declared.

The form of the CLOSE declaration is:

```
CLOSE close-name $
 BEGIN
 declarations and statements
 END
```

A CLOSE is called directly by a GOTO statement followed by the CLOSE name or indirectly by using the CLOSE name in a switch declaration which is called by a GOTO statement. But, it is permissible to transfer from within the CLOSE routine to an entirely independent point in the main routine or procedure from which the CLOSE call was declared within.

Examples:

|                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>• TABLE NAMES R 50 S 1 \$   BEGIN     ITEM NN H 10 0 0 \$   END   •   •   •   CLOSE CLEAR \$   BEGIN     FOR A = 49, -1, 0\$       NN(\$A\$) = 1H( ) \$   END   •   •   •   GOTO CLEAR \$   •   •   •   GOTO CLEAR \$   •   •   •   GOTO CLEAR \$</pre> | <p>A table is declared containing display code entries of one word per entry.</p> <p>A CLOSE routine is declared which sets this table item to Hollerith blanks.</p> <p>The CLOSE routine is called.</p> <p>It is called again, and may be called anytime during execution of the program when the table NAMES is to be reset to blanks.</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- ITEM DIST F \$  
ITEM SHOT F \$  
•  
•  
•  
CLOSE RANGE \$  
BEGIN  
DIST = DIST + SHOT \$  
IF DIST LS.01 \$ GOTO UNDER \$  
IF DIST GR 1056. \$ GOTO OVER \$  
END  
•  
•  
•  
GOTO RANGE \$
- Two floating items are declared.
- The CLOSE routine increments and tests item DIST. Note that there are two exits from the CLOSE that are not the standard return.
- A call to the CLOSE RANGE.

## PROCEDURES

A procedure is a closed form that can have both input and output parameters; formal data declarations must be made corresponding to formal input/output parameters in the procedure declaration. During execution of the procedure, the data transferred from the calling routine is referenced by the formal data declarations defined in the procedure. The outside data is specified in the actual parameters of the procedure call statement corresponding to the formal parameters of the procedure declaration. Thus, the procedure operates independently of the main routine or other procedures. The following restrictions apply to the use of procedures:

- A procedure declaration is independent of outside loop statements because loop variables are not recognized or affected within the procedure body.
- Loop variables defined within a procedure are not recognized or affected by operation of the routines outside the procedure.
- A loop variable can be specified as an actual parameter whenever a procedure is invoked from within a loop with that variable.
- A procedure declaration can appear anywhere within the program where a statement is permissible, except within another procedure or a function.
- A procedure may use or set any variable defined in the main program.
- It is not advisable to declare files within a procedure. Procedures may use files declared in the main program for input and output.

### PROCEDURE DECLARATION

The procedure declaration defines the procedure name and the formal input and/or output parameters; these are dummy names introduced in the procedure declaration that are

replaced by corresponding parameters introduced by the procedure call statement. The procedure declaration is followed optionally by a list of declarations defining the data used by the procedure, then by the statements defining the operation of the procedure enclosed in BEGIN-END brackets. The format is:

```
PROC name [([formal-input-parameters] [=formal-output-parameters])] $
 [declaration-list]
 BEGIN
 statements and declarations
 END
```

where:

|                          |   |                                                        |
|--------------------------|---|--------------------------------------------------------|
| name                     | = | user-supplied identifier of the procedure              |
| formal-input-parameters  | = | names of items, arrays, tables, or CLOSE routines      |
| formal-output-parameters | = | names of items, arrays, or statements                  |
| declaration-list         | = | declarations of data structures local to the procedure |

The following rules apply:

- CLOSE routines can be used only as input parameters. Statement names can be used only as output parameters.
- Both input and output parameters are separated by commas.
- In a formal parameter list, statement and CLOSE names are followed by periods.
- Every item, array, or table name in the formal parameter list as well as every data name introduced by the procedure must be declared in the procedure.
- The declarations can be placed immediately following the PROC declaration in the declaration list or they can be placed among the statements of the procedure body, or both.
- No procedure declaration can be included in the procedure. Other procedures can be called from the procedure, provided that they do not call the calling procedure either directly or indirectly.
- CLOSE names and statement names in the formal parameter list are not declared in the procedure.
- A formal input parameter cannot be a formal output parameter. To use an actual input parameter as an output parameter, the two separate formal parameters are declared with identical descriptions, then overlaid.

Examples:

- PROC SIMPL \$  
 BEGIN  
 •  
 •  
 •  
 END

This procedure declaration has no formal parameters.
- PROC COMP (LONG, LOG.) \$  
 BEGIN  
 ITEM LONG F \$  
 •  
 •  
 •  
 END

This procedure has only input parameters; one of which is a CLOSE routine. LOG is identified as a CLOSE routine because it is followed by a period and is an input parameter. The parameter LONG must be defined in the procedure.
- PROC POSIT(LIT, NUMB = RSLT)  
 BEGIN  
 ITEM LIT H 5 \$  
 ITEM NUMB I 60 5 \$  
 ITEM RSLT I 18 5 \$  
 IF LIT EQ 5H(START) \$  
 BEGIN  
 RSLT = 0 \$  
 RETURN \$  
 END  
 RSLT = NUMB \$  
 END

If the actual literal value passed to LIT is 5H(START), RSLT will be set to zero; if not, RSLT will be set to NUMB.
- PROC FIND(LOG., ALT, TAN. =  
 EX., AMT) \$  
 BEGIN  
 ITEM ALT A 60 S \$  
 ITEM AMT I 60 U \$  
 •  
 •  
 •  
 END

Procedure FIND has three input and two output parameters. LOG and TAN are CLOSE names because they are followed by a period and are in the input list. EX is a statement name providing an alternate exit from the procedure. It is identified as a statement name because it is followed by a period and is in the output list. ALT and AMT must be declared in the procedure.
- PROC XX (SUM1 = SUM2) \$  
 ITEM SUM1 F \$  
 ITEM SUM2 F \$  
 OVERLAY SUM1 = SUM2  
 BEGIN  
 •  
 •  
 •  
 END

The OVERLAY statement causes the input parameter SUM1 to be used as the output parameter SUM2. When the procedure is called in effect both the input and output parameters are set to the actual input parameter. This would be done when it was desired to manipulate the value of SUM1 and produce a



```

● PROC CHECK (TABIN =
 ARRAYOUT) $
 BEGIN'1''
 TABLE TABIN V 30 D $
 BEGIN
 ITEM INTGR I 60 S $
 ITEM FLOT F $
 END
 ARRAY ARRAYOUT 30 B $
 FOR Z = 0, 1, 29 $
 BEGIN'2''
 IF ENTRY ($ Z $) EQ 0 $
 BEGIN'3''
 ARRAYOUT ($ Z $) = 0 $
 TEST Z $
 END'3''
 ARRAYOUT ($ Z $) = 1 $
 END'2''
 END'1''

```

new value which would be returned to SUM1, by calling the procedure with both actual parameters being the same variable.

Procedure CHECK has a table as an input parameter and array as an output parameter. TABIN and ARRAYOUT must be declared in the procedure although only one word of storage will be allocated for each to hold the pointer to the actual array and table used in execution.

The input table is checked for empty entries which have both FLOT and INTGR set to zero. The corresponding item in array ARRAYOUT is set to zero or one depending on the result of the test.

The following example is intended to demonstrate the situations where nesting PROC and CLOSE declarations is permissible:

```

START $ '' SCOPE OF CLOSE AND PROC ''
 ITEM AA I 18 U $
 ITEM BB I 18 U $
 ITEM CC I 18 U $
 ITEM DD I 18 U $
 ITEM EE I 18 S $
 MONITOR AA BB CC DD EE CLS. PRC CLUZ. $
 MONITOR CLZ1. CLZ2. CLZ3. CLZ4. CLZ5. CLZ6. $
 AA = 5 $
 GOTO CLS $
 AA = 10 $
 PRC(AA = DD) $
 DD = DD $
 GOTO CLUZ $
 GOTO CLZ1 $
 GOTO CLZ2 $
 GOTO CLZ3 $
 GOTO CLZ4 $
 GOTO CLZ5 $
 GOTO CLZ6 $
 STOP $
CLOSE CLS $
 ''1''BEGIN
 BB = 5*AA $
 PRC(BB = CC) $
 CC = CC $
 '' OUTER CLOSE ''
 '' CALL PROC ''

```

```

PROC PRC(IN = OUT) $ '' DECLARED INSIDE OF CLOSE ''
''2''BEGIN
 MONITOR OUT CLOZ. $
ITEM IN I 18 U $
ITEM OUT I 18 U $
 OUT = IN * IN $
 GOTO CLOZ $
 GOTO CLZ1 $ '' CALL GLOBAL CLOSE IN A PROC ''
 RETURN $
CLOSE CLOZ $ '' INSIDE PROC INSIDE CLOSE ''
''3''BEGIN
 OUT = OUT * 2 $
''3''END
''2''END
CLOSE CLUZ $ '' INSIDE OF ANOTHER CLOSE ''
''4''BEGIN
 DD = DD/2 $
''4''END
''1''END
CLOSE CLZ1 $
''5''BEGIN
 EE = 1 $
CLOSE CLZ2 $
''6''BEGIN
 EE = 2 $
CLOSE CLZ3 $
''7''BEGIN
 EE = 3 $
CLOSE CLZ4 $
''8''BEGIN
 EE = 4 $
CLOSE CLZ5 $
''9''BEGIN
 EE = 5 $
CLOSE CLZ6 $
''10''BEGIN
 EE = 6 $
''10''END
''9''END
''8''END
''7''END
''6''END
''5''END
TERMS$

```

The results of the above PROC and CLOSE declarations are given in the following monitored output example:

```

*** MONITORED INTEGER DATA AA = 5 = 0(00000000000000000005)
*** MONITORED CLOSE CLS = 25 = 0(00000000000000000031)
*** MONITORED INTEGER DATA BB = 625 = 0(00000000000000000161)
*** MONITORED PROCEDURE PRC = 1250 = 0(000000000000000002342)
*** MONITORED INTEGER DATA OUT =
*** MONITORED CLOSE CLOZ
*** MONITORED INTEGER DATA OUT

```

```

*** MONITORED CLOSE CLZ1
*** MONITORED INTEGER DATA EE = 1 = 0(00000000000000000001)
*** MONITORED INTEGER DATA CC = 1250 = 0(000000000000000002342)
*** MONITORED INTEGER DATA AA = 10 = 0(00000000000000000012)
*** MONITORED PROCEDURE PRC
*** MONITORED INTEGER DATA OUT = 100 = 0(00000000000000000144)
*** MONITORED CLOSE CLOZ
*** MONITORED INTEGER DATA OUT = 200 = 0(00000000000000000310)
*** MONITORED CLOSE CLZ1
*** MONITORED INTEGER DATA EE = 1 = 0(00000000000000000001)
*** MONITORED INTEGER DATA DD = 200 = 0(00000000000000000310)
*** MONITORED CLOSE CLUZ
*** MONITORED INTEGER DATA DD = 100 = 0(00000000000000000144)
*** MONITORED CLOSE CLZ1
*** MONITORED INTEGER DATA EE = 1 = 0(00000000000000000001)
*** MONITORED CLOSE CLZ2
*** MONITORED INTEGER DATA EE = 2 = 0(00000000000000000002)
*** MONITORED CLOSE CLZ3
*** MONITORED INTEGER DATA EE = 3 = 0(00000000000000000003)
*** MONITORED CLOSE CLZ4
*** MONITORED INTEGER DATA EE = 4 = 0(00000000000000000004)
*** MONITORED CLOSE CLZ5
*** MONITORED INTEGER DATA EE = 5 = 0(00000000000000000005)
*** MONITORED CLOSE CLZ6
*** MONITORED INTEGER DATA EE = 6 = 0(00000000000000000006)

```

#### PARAMETER PASSING<sup>†</sup>

There are two types of parameter passing: by name and by value. If the user references a main program variable from within a procedure that was called with the variable as an actual input or output parameter, the effect will depend on whether the parameter was passed by name or by value.

Parameter passing by name is used for tables, arrays, CLOSE routines (input only), and statement labels (output only). The actual address of the parameter is passed; references to the corresponding formal parameter use the actual address. Referencing parameters passed by name have the same effect as if a formal parameter had been referenced.

Parameter passing by value is used for constants, variables, and formulas. Their value is computed and assigned to the formal parameter. Variables used as output parameters are passed by value at exit; the value of the parameter variable is assigned to the actual parameter on return from the procedure. Numeric parameters (input and output) that disagree in type are converted when the value is assigned. Referencing parameters passed by value yields the original value, regardless of any changes to the formal parameter. Conversely, if the procedure changes the value of the actual input parameter, the formal parameter will remain unchanged. The actual output parameter will reflect its original value until control is returned to the main program, or unless changed in an assignment statement using the name of the main program variable.

<sup>†</sup>See Appendix E for additional information on parameter passing for JOVIAL procedures compiled with the program, COMPOOL-defined JOVIAL subprograms, and COMPOOL or library non-JOVIAL programs.

There is no distinction between an array or table passed as an input parameter and one passed as an output parameter. The structure of a table or array need not be identical to that of the formal parameter except that arrays must be of the same type of variable (literal, integer, etc.). However, any deviation from an identical form is the programmers responsibility.

### PROCEDURE CALL

The procedure call statement is used to call a procedure. The format is:

name [( [actual-input-parameter] [=actual-output-parameter] ) ] \$

where:

- name = procedure identifier; this must be the same as the name in the procedure declaration.
- actual-input-parameters = the number, class, and order of the formal input parameters specified in the procedure declaration.
- actual-output-parameters = the number, class, and order of the formal output parameters specified in the procedure declaration.

The actual parameters use the corresponding formal parameters as aliases for the purpose of passing values and names to and from the procedure body.

The functional modifiers BIT, BYTE, CHAR, LOC, POS, MANT, NENT, and NWDSN can be used as actual input parameters or in formulas which are actual input parameters. The functional modifiers BIT, BYTE, CHAR, MANT, NENT, and POS can be used as actual output parameters. The following rules apply to the parameters of a procedure call:

- Actual input parameters are evaluated and the values are assigned to corresponding formal input parameters in the order (from left to right) that they appear in the procedure declaration.
- The value is computed and assigned when the procedure is called. The values of the formal output parameters are assigned to the corresponding actual parameters following execution of the procedure body at the time a normal exit is made.
- When a CLOSE routine named as a formal input parameter is executed; it must have been declared in the main routine.
- Statement names used as actual output parameters name the statement to which control transfers when the corresponding formal statement name is referenced in the procedure body.

- Control normally returns to the statement following the procedure call after execution of the procedure is complete, except when an exit is made to a statement named as an output parameter or an execution of a return statement. Exits from closed forms are discussed later in this section

Examples:

- `COMP(AA + BB/180, LNG.)$`  
This is a procedure call for the PROC COMP example declared under procedure declaration. The formula `AA + BB/180` is evaluated when the call is executed and becomes the value of item LONG in the procedure body. When the CLOSE LOG in the procedure is called, CLOSE LNG is executed. (LNG had been declared in the calling program.)
- `FIND(LL., 3000, TT., =  
OUT., HH)$`  
This calls procedure FIND declared previously. LL and TT are CLOSE names, 3000 is an integer constant which will be an initial value for item ALT, OUT is a statement name, and HH is a numeric item.
- `POSIT(BYTE($ZZ, 5$)(CARD),  
BIT($38, 5$)(CARD) =  
POS(FILEB))$`  
This example (and the following two) demonstrate the use of functional modifiers and formulas for input parameters and functional modifiers as output parameters. CARD is a literal variable, ZZ is an integer variable, and FILEB is a file declared in the program that called POSIT.
- `POSIT(HOL, POS(FILEC) =  
NENT(TABD))$`  
HOL is an Hollerith variable, FILEC is a file declared in the main program, and TABD is a variable table declared in the main program or a procedure from which the procedure POSIT was called.
- `POSIT(HOL, COMPUT  
(BIT($45, 3$)(INT)) =  
BIT($0, 15$(FIX))$`  
HOL is a literal variable, COMPUT is an integer function, INT is an integer variable, and FIX is a fixed point variable declared previously.

## FUNCTIONS

A function is a special form of a procedure. Only input items may be specified in a function call. An item declared within the procedure which has the same name as the procedure serves as the output parameter. Since a function results in a single value being output, it can, therefore, be treated as a variable. The type of the variable with the same name as the function will be the type of the function, Hollerith, integer, fixed, etc. It is the responsibility of the user to insure that this variable is set with the desired value at the time the function is exited. Whatever the value of the variable is at that time is the value the function will return. Several function calls may be included in one statement. The function declaration is introduced by the same word as a procedure (PROC). The format is:

```
PROC name [(formal-input-parameters)] $
 ITEM item-name item-description $
 [declaration-list]
BEGIN
 statements and declarations
END
```

where:

name = function identifier and is the function's formal output parameter. The formal input parameters are the same as those used for a procedure declaration.

item-name = must be the function name

item-description = describes the form the function will take

declaration-list = declaration of data structures local to the procedure.

## FUNCTION CALL

The function is called by its name followed by any actual parameters. The format is:

```
name ([actual-input-parameters])
```

where:

name = function name; this is the same as the item name declared in the function.

actual-input-parameters = same as described in procedure calls. If there are no actual input parameters, the pair of parentheses must be specified after the function name to identify the name as a function call.

When a function is called, the statements in the function declaration are executed and then control returns to the function call. The value resulting from the execution is assigned to the name in the function call.

Examples:

- PROC VALUE(AA, BB) \$  
  ITEM VALUE F \$  
  BEGIN  
    •  
    •  
    •  
    •  
  END  
  COMP = VALUE(XX, YY) \$

Function VALUE has two input parameters; it will return a floating-point result. Execution of VALUE will determine a value for VALUE dependent on the values input for AA and BB as actual input parameters. AA and BB must be declared in the declaration list.

This assignment statement calls the function VALUE using as input the current values of the items XX and YY. After execution the value determined for VALUE is returned and assigned to item COMP.

- PROC TAN \$  
  called by:  
  TAN( ) \$

This function has no parameters. When called it must be identified by the name TAN followed by an empty pair of parentheses.

- IF TAN( ) GR 1.50 \$

The procedure call may be used in an IF statement. The value of TAN is determined and subsequent action depends on its value compared to the constant 1.50.

#### EXIT FROM CLOSED FORMS

Normally, exit from a closed form occurs automatically and control is returned to the calling point when the sequence given in the closed form has been executed. Control is returned to the calling point before the entire sequence has been executed if the RETURN statement is used.

The format of the RETURN statement is:

```
RETURN $
```

The use of a GOTO statement allows exit to another point in the main program. When an exit is made from a function to a global statement label or switch by a GOTO statement, control is immediately returned to the main program at the point specified by the GOTO and the value of the function is not returned. In an exit from a procedure, the simple or subscripted items in the actual output parameter list are assigned the current value of the corresponding formal parameters only if the GOTO specifies a statement label which is named as one of the formal output parameters. If the GOTO leads directly to a statement

label or switch in the main program, names of the item parameters are assigned to values calculated by the procedure. Array and table actual parameters always reflect the change made during the execution of the procedure regardless of the means of exit from the procedure.

Examples:

- XX(ITA, ITB, CLOSE1. = ITD, ATAB, LABX.)\$  
•  
•  
•  
PROC XX(AA, BB, CL1. = DD, TABLA, STATX.)\$  
ITEM AA (item-description)\$  
ITEM BB (item-description)\$  
ITEM DD (item-description)\$  
TABLE TABLA (table-description)\$  
BEGIN  
ITEM TAA I 18 S \$  
ITEM TBB I 18 S \$  
END  
BEGIN 'XX'  
•  
•  
•  
DD = AA\$  
GOTO STATX\$  
•  
•  
•  
IF TAA(\$1\$) LS 0 OR TBB (\$1\$)  
LS 0 \$  
RETURN\$  
•  
•  
•  
IF TAA(\$1\$) GR 0  
AND TBB(\$1\$) GR 0 \$  
GOTO LC1.  
•  
•  
•  
IF TAA(\$1\$) EQ 0  
AND TBB(\$1\$) EQ 0 \$  
GOTO ERRXX \$  
•  
•  
•
- Call to procedure XX.
- Declaration of XX with two simple input items, one simple output item, an input CLOSE routine, an output table, and a statement label as an alternate.
- The exit is made here to statement LABX in the main routine; ITD takes the current value of DD.
- RETURN exits to statement after call to XX; ATAB and ITD contain current values of TABLA and DD.
- Since LC1 is local to XX, transfer to LC1 does not cause an exit.
- Exit to ERRXX (statement name in main routine but not an actual parameter) does not set any values for actual output parameters.



LC1. (continuation of procedure)

•  
•  
•  
END

END signals the normal exit from the procedure; all output parameters are set to current values, control transfers to statement after call to XX. If control flows out the bottom, it will automatically return.

• X1 = FCC(FF, MON.)\$

•  
•

Call to function FCC.

PROC FCC(F1, MNT.)\$

Declaration of FCC

ITEM FCC F \$  
ITEM F1 F \$

BEGIN

F1 = 0.\$  
GOTO MNT \$.

Call to CLOSE MNT executes close MON declared in main routine. Control returns to statement after call to MNT in function FCC.

•  
•  
•  
IF F1 GR 25.12 \$

FCC = F1 \$  
RETURN \$

RETURN exits to statement calling FCC. Item F1 is assigned current value of FCC.

•  
•  
•  
IF F1 LS -1.215 \$

GOTO ERRFC \$

Exit to ERRFC transfers control to main routine; statement calling FCC is not executed.

•  
•

FCC = F1 \$

END

Normal exit from function returns control to statement calling FCC; F1 is set to current value of FCC.

A communication pool (COMPOOL) is a dictionary of definitions referenced by the compiler during a source program compilation. The use of a COMPOOL enables the user to separate the definition of the system data base from the program in the system, thereby relieving the user from maintaining the definitions and minimizing the effect on his programs when the system data base structure changes. To produce a COMPOOL, the compiler interprets specifications prepared by the user. These specifications (written in JOVIAL syntax) define two major elements of a JOVIAL programming system. They are:

- A system data base comprised of definitions of data common to more than one program.
- A program library containing names and parameter descriptions of external subprograms that can be called by JOVIAL programs and subprograms.

Once a COMPOOL has been created, any program in the system may be compiled using the definitions contained in the COMPOOL.

### **COMPOOL SPECIFICATION**

A COMPOOL specification uses JOVIAL syntax and is enclosed in START-TERM brackets. It is similar to a JOVIAL program except that the COMPOOL specification contains only declarations. The format is:

```
START$
 COMPOOL declarations
TERM$
```

where:

COMPOOL declarations = Any number of declarations of common data and/or sub-program declarations. These can be in any order.

### **DATA DECLARATION**

Data declarations in the COMPOOL specification are organized into one or more common blocks. All the data declarations can be in one common block or there can be as many

blocks as there are data declarations. The following restrictions apply to data declarations in a COMPOOL specification:

- Data declarations are specified like any JOVIAL data declarations.
- An overlay statement may be used to specify the allocation of previously declared data within one common block, but data can not be overlaid from one common block to another.
- Presets may be included with the data declarations. The preset information is, however, not placed on the same file as the COMPOOL file output (see the section on COMPOOL compilation output).

### COMMON DECLARATION

If a common block of data has no name, it is called blank common. All unnamed blocks of common are treated as one block composed of the data in all such blocks. If a block is named, it must have a JOVIAL name; all blocks having the same name are treated as a single block.

The format of a common declaration is:

```
COMMON (block name)$
 BEGIN
 data declaration(s)
 END
```

### SUBPROGRAM DECLARATION

The description within the COMPOOL must define the subprogram name and describe the parameters.

Since JOVIAL has more than one method of passing parameters, the compiler will perform type conversion for numeric actual parameters in JOVIAL procedures, and may call non-JOVIAL procedures. The descriptor of the routine and its parameters must be known at compile time. If either a JOVIAL procedure which is not compiled in the program that called it or a non-JOVIAL routine which is not listed in the library table within the compiler is to be called, the procedure must be described in a COMPOOL used when compiling the calling program.

The format is:

```
PROC subprogram name [(formal-parameters)]$
 [BEGIN
 formal-parameter data-declarations
 END]
```

The following rules apply to subprogram declarations:

- The formal parameters for JOVIAL subprograms follow the same rules as in JOVIAL subprogram compilations.
- CLOSE names and statement names in a parameter list are followed by a period. Array, item, and table names used as formal parameters must be defined in data declarations within the BEGIN-END brackets following the subprogram heading.
- Function type subprograms are distinguished by an item declaration within the BEGIN-END brackets with the same name as the subprogram.
- Subprograms without parameters need only declare the name of the subprograms. The parenthesized formal parameter list and the data declarations are not required.
- Subprograms may be non-JOVIAL. The compiler recognizes a non-JOVIAL subprogram by the absence of data declarations for parameters that were specified with a subprogram. The compiler generates a calling sequence for non-JOVIAL subprograms that is different from a JOVIAL subprogram calling sequence. Parameter passing and calling sequences are described in Section 7 and Appendix E.

## COMPOOL CREATION

Once the data structure, common blocks, and the subprogram specifications have been coded, they may be submitted for a COMPOOL compilation. The COMPOOL compilation parameter, A=lfn, on the JOVIAL control card indicates to the compiler that it is to perform a COMPOOL compilation. When the A parameter is specified, the C, F, M, E, and W options are meaningless. The outputs of a COMPOOL compilation are:

- The binary file containing the COMPOOL. This is the name specified by the COMPOOL parameter, A=lfn. It may be any legal seven character SCOPE file name. It will contain the control information for the data items in the common blocks and the names of the external subprograms defined in the COMPOOL input.

- The Hollerith file containing listing output. This is the same specified by L = lfn. If no name is specified, the default is OUTPUT. The source listings, diagnostics, storage map, and cross reference appear here if they were requested on the JOVIAL card.
- The binary file containing the presets. This is an optional file. It will not be created if no presets were specified in the data declarations in the COMPOOL compilation input. If it is created, it is placed on the file specified by B = lfn on the JOVIAL control card. If no name is specified, the default preset file is LGO. The entry point name may be specified by N=name, where name is any six character JOVIAL name. If no entry point name is specified, the default name is MAINPG. It is the user's responsibility to see that the preset file output is included with the binary object programs with which it is to be used.

The COMPOOL file and preset file, if produced, should normally be saved so that they may be used when compiling the programs in the system under development. Placing them as permanent files will enable several users to access them at once if the multiple read parameter is used on the permanent file attach card. Refer to the SCOPE Reference Manual for instructions on cataloging and attaching a permanent file.

Once a COMPOOL is created, the dictionary of definitions contained in it may be referenced by the compiler during source program compilation by using the C = COMPOOL file option on the JOVIAL control card.

## COMPOOL REFERENCE

A COMPOOL is required for compiling a source program only if the program references data or external subprograms which are described in the COMPOOL. The COMPOOL may be created at the start of the job, or more efficiently, by loading an existing COMPOOL from tape or attaching a permanent file containing the COMPOOL. Once the job has access to the COMPOOL, JOVIAL programs using the definitions in COMPOOL may be compiled by specifying C = COMPOOL-file on the JOVIAL control card.

During the compilation of a program, the compiler first searches the program for definitions of names; if they are not defined in the program, the definitions are sought in a COMPOOL if one is present. If a name still remains undefined, the list of library functions and procedures within the compiler is checked; if the definition is found in the list, a definition is created for the name which will reference the corresponding library procedure or function. Finally, if a name still remains undefined, it is flagged as an undefined element.

If a name is defined both within the program being compiled and within a COMPOOL used for the compilation, the definition within the program will be used. (Any COMPOOL definition may be overridden by defining the same name within the program.)

When a name is used within a program but defined within a COMPOOL used during compilation, the compiler will locate the definition within the COMPOOL, retrieve it, and then process it just as if the name had been declared within the program. The only exception to this is that COMPOOL defined items will be noted in the listing output.

#### **DATA REFERENCE**

A reference in a JOVIAL source program to any COMPOOL defined item, array, or table is processed exactly as if the data element were declared in the program.

#### **SUBPROGRAM REFERENCE**

Once an external subprogram, a closed program compiled independently of the calling program, or a subprogram is described in a COMPOOL, it may be called by any JOVIAL program or subprogram which is compiled using the COMPOOL containing the description of the external subprogram. The external subprogram may be JOVIAL or any other language, provided that the subprogram will accept the non-JOVIAL calling sequence described in Appendix E.

Calls to JOVIAL subprograms for a JOVIAL program are identical in form to procedure and function calls declared within the main program. But unlike procedures and functions within the main program, an external subprogram can not reference names in the calling program. The only communication between a calling program and a subprogram is through the parameter list and, optionally, data items declared in a COMPOOL.

Calls to non-JOVIAL subprograms are similar to calls to JOVIAL procedures or functions except that no distinction is made between input and output parameters; the subprograms can modify the actual parameters passed to it.

The formats for procedure and function calls are:

Procedure call:

procedure name [(actual parameters)] \$

Function call:

function name ([actual parameters])

The rules that apply to procedure and function calls are:

- In both cases, the actual parameters can be omitted if there are no formal parameters in the COMPOOL subprogram declaration.
- The pair of parentheses must follow the function name in the function call even if there are no actual parameters.
- When actual parameters are specified, they should agree in order, number, and type with the formal parameter list in the subprogram declaration.
- The list of actual parameters can include one or more of the following:
  - Simple item, table, or array name; subscripted item name; arithmetic formula; and a statement or CLOSE name followed by a period. See Appendix E for calling sequences and linkage conventions.

## COMPOOL EXAMPLES

Appendix H contains a sample listing with a COMPOOL containing two common blocks and two JOVIAL subprograms. Appendix I contains a sample program with a COMPOOL which has both JOVIAL and non-JOVIAL subprograms.

## PROGRAM STRUCTURE

A program is a self-contained unit that can be compiled independently of any other program. A main program is called at execution time by the operating system. A subprogram is called by a main program or by another subprogram; when subprograms are called, parameters can be passed between main programs and subprograms and between subprograms and other subprograms.

For a subprogram to be called, the calling routine must be compiled using a COMPOOL in which the subprogram to be called is described. The resulting object program is combined with the called subprogram into one segment for execution.

## MAIN PROGRAM

A main program consists of all the statements and declarations that make up a program. It is enclosed by START-TERM brackets. The format is:

```
START$
 declarations and statements
TERM [statement-name] $
```

If the statement name is included, execution of the program starts with the named statement; if omitted, execution starts with the first executable statement.

## SUBPROGRAM

A subprogram, like a main program, consists of declarations and statements enclosed by START-TERM brackets. The START must be followed immediately by a subprogram declaration. The format is:

```
START subprogram-declaration $
 declarations and statements
TERM $
```

A TERM statement label is not legal for subprograms; execution begins with the first executable statement.

The format of a subprogram declaration is identical to that of a procedure declaration. The format is:

```
PROC name [(formal-input-parameters) [= formal-output parameters]] $
```

where:

```
name = user supplied identifier
formal-input-parameters }
formal-output-parameters } are identical to those described under procedure and
 } function declarations.
```

Subprograms are compiled separately. Once compiled, a subprogram is called in the same way a procedure or function is called. A subprogram can contain CLOSE routines, procedures, and functions just as a main program does.

Examples:

|                                                                             |                                                                                                                                |
|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| ● START 'PROGRAM FOLLOWUP'<br>(declaration list)<br>MODE B\$<br>●<br>●<br>● | This is a main program. Tables, arrays, and items are declared at the beginning of FOLLOWUP. Undeclared items will be Boolean. |
| INT. (data initialization)<br>●<br>●<br>●                                   | Data is set to initial values.                                                                                                 |
| IN. (input from external files)<br>FIND(AAA, TAB1 = ITT) \$<br>●            | Files are read<br>Call to procedure FIND                                                                                       |



|                                   |                                |                                                                       |
|-----------------------------------|--------------------------------|-----------------------------------------------------------------------|
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| EXEC(CC1., ARR1, FF = IT, OUT.)\$ |                                | Call to subprogram EXEC.                                              |
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| OUT. (output to external files)   |                                | Output files are written                                              |
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| FFAL = TIM(A2, B2) \$             |                                | Statement calls function TIM                                          |
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| PROC FIND(A1, TT1 = TOTAL) \$     |                                | Declaration of procedure FIND                                         |
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| PROC TIM(AA2, BB2) \$             |                                | Declaration of function TIM.                                          |
| •                                 |                                |                                                                       |
| •                                 |                                |                                                                       |
| TERM IN \$                        |                                | End of FOLLOWUP; first statement<br>to be executed is at location IN. |
| •                                 | START \$                       |                                                                       |
|                                   | PROC EXEC(LCO., RAY,           |                                                                       |
|                                   | IFT = TT, CONT.) \$            |                                                                       |
|                                   | BEGIN                          |                                                                       |
|                                   | (declaration list)             |                                                                       |
|                                   | •                              |                                                                       |
|                                   | AA. (first statement executed) |                                                                       |
|                                   | •                              |                                                                       |
|                                   | •                              |                                                                       |
|                                   | GOTO CONT \$                   | Exit to main routine location OUT.                                    |
|                                   | •                              |                                                                       |
|                                   | •                              |                                                                       |
|                                   | END                            | Normal return here.                                                   |
|                                   | TERM \$                        | TERM indicates end of subprogram                                      |

JOVIAL debugging aids provide the user a means of checking for errors in program design. These aids are:

- The MONITOR statement enables the user to obtain data values during execution or to perform a program trace.
- The Run-Time Error Monitor terminates program execution where the source errors were such that the compiler could not make corrections to produce valid code.

### MONITOR STATEMENT

The MONITOR statement can be used to obtain a trace of the program flow or to print the values of designated variables during execution. It is very effective in checking errors in program design. It also may be used to provide formatted output for obtaining quick answers to computational problems.

For MONITOR statements to execute, the M option must be indicated on the JOVIAL control card. If the M option is not on the JOVIAL control card, all monitor statements are treated as comments. Thus, the MONITOR cards do not need to be removed from the source deck to compile and execute the program without monitor output. The format is:

[name.] MONITOR [(Boolean-formula)] name<sub>1</sub> [ {  $\Delta$  } name<sub>n</sub>[.] ] \$

where:

Boolean-formula = Refer to page 3-4 for a description of a Boolean formula

name<sub>1</sub> } = simple or table items, arrays, statements, switches, CLOSE  
 name<sub>n</sub> } routines, functions, or procedures.

Monitor statements must conform with the following:

- A name associated with a statement label, switch, or CLOSE must be followed by a period.
- At least one name must be specified in the list of names.
- Names to be monitored must be separated by a blank or comma.

- A MONITOR declaration of a name is effective from the point of declaration until the end of the program. It may follow or precede the declaration of the name. References to a monitored name prior to the MONITOR declaration will not be monitored.
- If a Boolean formula is present in a MONITOR declaration, it will be evaluated each time the name is monitored. If the formula is false, the MONITOR output will be suppressed.
- A name may appear in more than one MONITOR declaration. If Boolean formulas are present, the one from the last MONITOR declaration containing the name will be the one used.

Monitoring will take place when:

- Simple items, table items or array items are on the left side of an assignment statement or on either side of an exchange statement.
- Statement labels have program control pass through the statement location.
- Switches are called.
- Functions, procedures and CLOSE routines begin execution.

MONITOR uses system I/O for output. The form of output is:

- Integer items are converted to integer numbers and the octal value is printed.
- Fixed-point items are converted to exponential notation, the number of fractional bits and the octal value of the variable are printed.
- Floating-point items are converted to exponential notation and the octal value is printed.
- Hollerith literals are printed as literals.
- Transmission code (STC) literals are converted to Hollerith equivalents. If the literal contains bytes which have no STC value, they will be represented by a down-arrow (↓), octal 71.
- Boolean items print the octal value; FALSE if it is 0 and TRUE if it is 1.
- Status items print the integer value of the status constant and the octal equivalent. That is, if the status constant had 12 status constants and the eleventh constant was set, it would have a decimal value of ten. The integer value is always one less than the position of the status constant in the status list.

- Table and array items will have their subscript value printed as well as the value of the item.
- Statement labels, switches, procedures, functions, and CLOSE routines will have their names printed.

If the monitored output is specified for the same file that the user has assigned as a device name in a file declaration, the MONITOR output will be interspersed with that of the program. This happens when the M option is used by itself and the program has a file assigned to OUTPUT, or if the M option has the same name as a device name in a file declaration. If monitored data is to be output separately from the program data, the output can be placed on a separate file by specifying M = file-name on the JOVIAL control card, where file-name is the name of a desired SCOPE file and is not a device name in a file declaration. Programmer action will be required to print the file.

The use of the M option for monitoring will cause a file environment table (FET) and a buffer to be produced; OUTPUT is assumed if no file is specified, or is the name specified. The buffer size will be 129 words unless the program contains a file declaration with the file name as a device name. In that case, the buffer size specified in the file declaration will be used.

Care should be taken in using MONITOR because large program traces or looping programs can result in extensive outputs.

Following is a source program using the MONITOR feature. The program was compiled using the M option on the JOVIAL control card. The output from MONITOR follows the program.





Figure 9-2 shows a sample deck of a program being compiled that has the M option specified on the JOVIAL control card. If no file was declared with OUTPUT as a device name, a FET and 129 word buffer is created for use by MONITOR. This FET and buffer may be used by PRINT, LIST, and ENDL for output. If they are used, their output will be interspersed with that from MONITOR. If a file declaration did use OUTPUT as a device name, the buffer size is as specified in the declaration. The MONITOR output is interspersed with the output from the program to the file using OUTPUT as a device name.

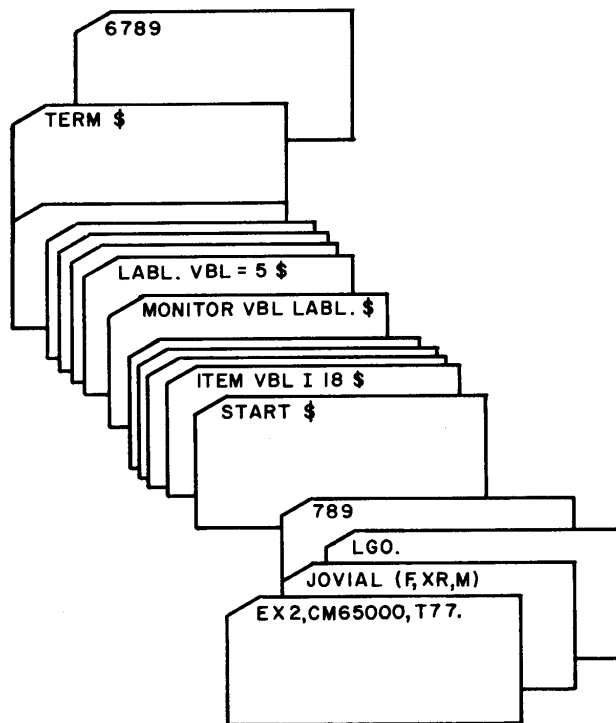


Figure 9-2. MONITOR to OUTPUT

Figure 9-3 shows a sample deck of a program being compiled using the M = file-name option specified on the JOVIAL control card. If no file was declared with TEMP as a device name, a FET and 129 word buffer is created. This may only be used for MONITOR output. To obtain the output, programmer action is required. It must be rewound and copied to the output file. If a file was declared with TEMP as a device name, the buffer size is the size specified in the declaration. The output to the file is interspersed with the MONITOR and program output.

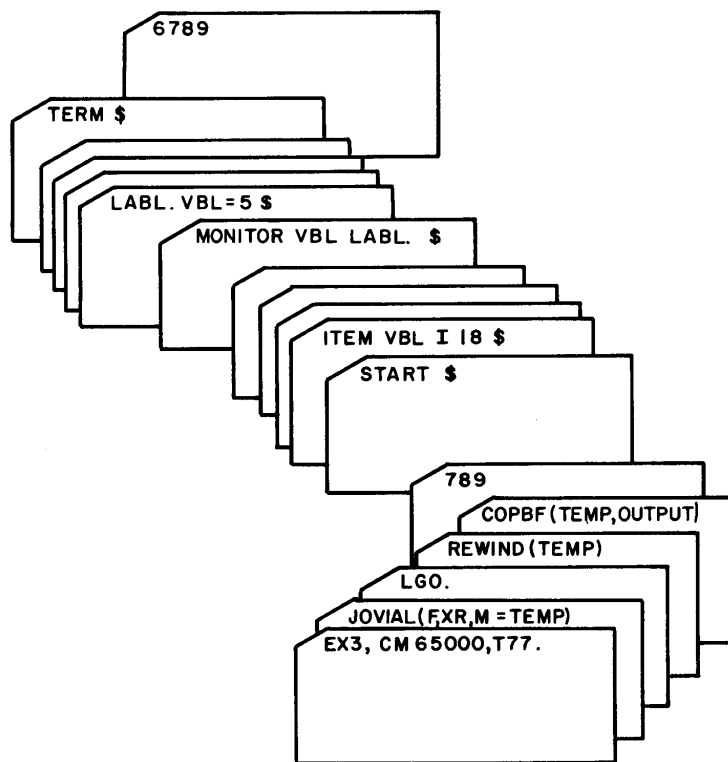


Figure 9-3. MONITOR to Separate File



## **RUN-TIME ERROR MONITOR ROUTINE**

The Run-Time Error Monitor will be called to terminate program execution if an attempt is made to execute a section of the program where source errors were of such severity that the compiler could not generate valid code.

Whenever possible, source errors will not prevent the completion of compilation and an attempt to execute the program. If the compiler encounters a source error for which it can take corrective action and produce valid code it will do so. A type W diagnostic message is issued if this action occurs. If the error encountered is such that corrective action cannot be taken but the compiler is able to continue compilation, a call to Run-Time Error Monitor with the parameters for the line and diagnostic numbers is inserted into the code at that point. A type E diagnostic message is issued if the action occurs.

### NOTE

Type E diagnostic message numbers and their interpretation may be found in Appendix A.

If during that execution, control of the program does not pass through the bad code, execution will proceed as if it were not there. If an attempt is made to execute the code which would have been produced, the Error Monitor routine is called. The number assigned to the JOVIAL source line by the compiler and the number of the diagnostic message are printed, and execution is terminated with control returning to the operating system.

Note that an output file must be available, either by declaring a file with device OUTPUT in the program or using the M option on the control card. If no output device is available when an Error Monitor is called, an abort will occur.

The dayfile message EXIT will be printed if the Error Monitor is called when an output device is unavailable.

The capability of compiling programs and executing them even though some statements may contain errors makes it possible to execute programs up to the point of the error. In the case of a program which is being tested and expanded, a test on the existing portions of the program and a diagnostic edit of the portions being added may be done in one compile by not executing the sections being added.

Example:

Given the statement containing a Hollerith variable, HOL, and an integer variable, INT, on line 203 of the program, the following action would occur:

- HOL = INT \$

This statement is not a legal JOVIAL statement and is such that the compiler cannot correct it or generate valid code. The code is replaced by a call to the Error Monitor.

```
***ERROR MONITOR CALL
LINE 203 HAD ERROR 109
EXECUTION TERMINATED
```

If an attempt is made to execute the above statement, the error monitor is called. The error message number and line number are printed and the program is terminated. The error number is only printed in this case. Normal compilation errors result in the printing of both the number and the message.

JOVIAL compilation of a source program is requested by the JOVIAL control card preceding the source deck.

### CARD FORMAT

The control card that calls for the compilation of a JOVIAL source program consists of the characters JOVIAL optionally followed by a parameter list enclosed in parenthesis. The card columns following the right parenthesis may be used for comments and are transcribed to the DAYFILE. If a parameter list is not given, a period must be used to separate the characters JOVIAL from any comments that appear on the card. Three formats of the JOVIAL control card are given below:

```
JOVIAL(P1, P2, . . . , P10) Comments
```

```
JOVIAL, P1, P2, . . . P10. Comments
```

```
JOVIAL. Comments.
```

### PARAMETERS

There are 12 optional parameters that can be specified on the JOVIAL control card. The parameter list has a free format; the parameters can be specified in any order or even omitted entirely. The file name, fn, and the deck name which may be used in some of the parameters, must begin with a letter and must be no longer than seven alphanumeric characters.

### SOURCE INPUT

If the source input parameter is omitted, the JOVIAL source input is assumed to be on INPUT. Parameters of the form I = COMPILE or I are equivalent and refer to the default output from UPDATE. If the source is on any other file, a source input parameter of the following form must be provided:

```
I = fn
```

where:

```
fn = the name of the file containing the input.
```

## BINARY OUTPUT

If the binary output parameter is omitted, a relocatable binary file is written on a file named LGO. Binary output parameters of the form B=LGO or B are equivalent to omitting the parameter. If B=0 is specified, all binary output is suppressed. For any other output file, a binary output parameter of the following format must be used:

B = fn

where:

fn = the file name on which the binary output is to be written.

## COMPOOL

If the COMPOOL parameter is omitted, a standard COMPOOL is assumed. Parameters of the form C=COMPOOL or C are equivalent to omitting the parameter. If the COMPOOL is on any other file, the format is:

C = fn

where:

fn = the name of the file containing the COMPOOL input.

## LIST

If the list parameter is omitted, a normal listing is provided on OUTPUT; this includes the source language and major diagnostics. Other list options are given below. The format is:

l = fn

where:

l = one or more of the following:

L = Normal listing, diagnostics follow the source.

D = Interlinear listing, phase 1 diagnostics interlinear with source. Phase 1 will continue until all source has been scanned and diagnostics issued. (If interlinear diagnostic had not been specified, compilation would be terminated when 200 phase 1 diagnostic message are issued.)

X = Storage map

R = Assembler cross-reference table.

} Common blocks listing

Q = If neither X nor R is specified, the common blocks listing is not listed. To override this, use the Q option. The common blocks listing contains the name, number, and length of each common block (including blank common, denoted by (BLANK) in the name column).

O = Listing of generated object code.

fn = The file name on which the output list is to be written; if fn is omitted, the listing will be on QUTPUT; if fn = 0 is specified all output except for diagnostics will be suppressed.

Any combination of the above options can be utilized. Commas are not required to separate the options although they may be used if desired. To select all the options, use LXRO or LXRO=fn.

### OPTIMIZATION

If the optimization parameter is omitted, global optimization is not performed. An F on the control card directs that optimization will take place. If it is requested, the optimizer phase of the compiler is called (before code generation) to:

- Eliminate redundant computation by recognizing common subexpressions.
- Redistribute code by moving invariant computations occurring within loops to potentially lower frequency regions.
- Substitute values for locally constant variables to increase common subexpression recognition.
- Reduce the strength of operations where the frequency of execution warrants the necessary setup cost. This is useful, for example, in the repeated computation of subscripts involving the induction variable of FOR loops.
- Collect frequency information in order to improve register assignment, and identify those quantities which have no further logical purpose in the program.

Because of its sophistication, the optimizer is extremely sensitive to both syntax and logical errors, such as jumps into loops. The optimizer should only be used on programs which have been thoroughly checked out and are completely free of errors. That is, the optimizer should be used only when a program is ready to be put on the system for use, and even then it should be used with caution.

### MONITOR

If the monitor parameter is missing, all monitor statements in the program are treated as comments. The monitor parameter format is:

M = fn

where:

fn = the file name on which the monitor output is placed. The monitor forms  
M = OUTPUT and M are the same.

The monitor parameter causes a file to be created for monitor output. The file name is fn, and unless fn is the same as a device name in a file declaration, a 129 word buffer is specified. It may only be used for monitor output, except that PRINT, LIST, and ENDL may be used if fn is OUTPUT. If fn is the same as a device name, the buffer size specified in the declaration is used and program output may be interspersed with the monitor output. If fn is other than OUTPUT, user action will be required to print the file.

#### **OPTIONAL PRIME**

If the optional prime parameter is omitted, MONITOR and LOC are recognized as primitives with or without a preceding prime. When the prime parameter is specified, they are recognized as primitives only if a preceding prime is present. This option permits the compilation of programs written for compilers that do not have LOC and MONITOR as primitives, which permits them to be user-defined names. The format is:

P

#### **COMPOOL ASSEMBLY**

When this parameter is used, a COMPOOL assembly is performed rather than a normal compilation. The file name COMPOOL should not be used. When this parameter is used, the C, F, M and W options are meaningless. The format is:

A = fn

where:

fn = the file name on which the COMPOOL assembly will be placed.

#### **TERMINATE COMPILATION**

The terminate compilation parameter terminates compilation following the output of all analysis error messages. No object code, object code listing, cross reference, or storage map may be produced. With the terminate parameter, the B, X, R, F, W and M options are meaningless. This option may be used to provide a very fast compilation to check for source errors. The format is:

T

### **SINGLE STATEMENT SCHEDULING**

When this is specified, the object code will be scheduled for maximum system efficiency only on a statement-by-statement basis rather than over as large a section of program as possible. Depending on the characteristics of the program, use of this option can degrade the generated object code. It usually requires more calls to the scheduler, thus increasing compilation time. The format is:

W

### **PROGRAM NAME**

Absence of this parameter is equivalent to N = MAINPG. The program name parameter is meaningless in subprogram compilations because the deck name used is the name of the subprogram entry point. This option provides a deck name for main programs and COMPOOL assemblies; when a COMPOOL assembly object deck and a main program deck are being loaded together, the N option must be used on at least one of them in order to prevent a duplicate program name. The format is:

N = deck-name

### **OVERLAY TRANSFER ADDRESS**

The overlay transfer address parameter is used to compile programs with a transfer address for overlays other than (0, 0). The format is:

E

Main programs that provide transfer addresses for overlays other than overlay (0, 0) must be compiled with this parameter. Subprogram compilations are not affected by the E parameter.

This parameter must not be used when compiling programs in overlay (0, 0) or JOVIAL main programs in a nonoverlay environment.

## **APPENDICES**



## COMPILER ERROR MESSAGES

A

This appendix contains two types of JOVIAL compiler error messages: source diagnostic messages and termination messages. Source diagnostic messages, which are described first are issued if errors occur during compilation of the source program. Termination messages are issued when compilation is terminated prematurely.

### SOURCE DIAGNOSTIC MESSAGES

There are three classes of source diagnostic message; each is distinguished by a code letter, as listed in Table A-1.

TABLE A-1. DIAGNOSTIC MESSAGE CLASSES

| Severity Code | Message Class | Error Description                                            | Compiler Action                                                                                                                |
|---------------|---------------|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| W             | Warning       | Actual or possible source errors                             | The compiler has attempted corrective action.                                                                                  |
| E             | Error         | Errors which probably cause generation of unexecutable code. | The compiler replaces statement in error by call to run-time error monitor routine. Partial check-out of programs is possible. |
| F             | Fatal         | Severe errors that prohibit creation of object code          | Processing is terminated after the message is printed on the device OUTPUT.                                                    |

When an E type error is encountered in the source, the code for the statement in error will be removed and replaced with a call to the run-time error monitor routine. The error-monitor routine prints on device OUTPUT a message indicating that it was called, the source line number that contained the diagnostic, and the diagnostic number. Execution is then terminated. The user must insure that a file with device name OUTPUT is available.

The source diagnostic messages are listed in numerical order in Table A-2. Whenever an underline appears in the message, it will be replaced by a source program name.

No more than 200 error messages will be produced for a single compilation.

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES

| Number | Severity | Message                                                        |
|--------|----------|----------------------------------------------------------------|
| 1      | F        | EXCESSIVE ERRORS IN SOURCE PROGRAM. DIAGNOSTICS TERMINATED     |
| 2      | W        | _ IS PRECEDED BY A PRIME. PRIME IGNORED                        |
| 3      | W        | ILLEGAL LETTER IN FLOATING CONSTANT. *E* ASSUMED               |
| 4      | W        | _ IS A DUPLICATE IDENTIFIER                                    |
| 5      | E        | AN ILLEGAL LETTER PRECEDES (                                   |
| 6      | W        | LEFT PARENTHESIS MISSING AT START OF LITERAL CONSTANT          |
| 7      | W        | DECLARATION ACCEPTED THOUGH PRIOR REFERENCES EXIST FOR _       |
| 8      | W        | FLOATING CONSTANT ENDING IN E IS ILLEGAL                       |
| 9      | W        | ILLEGAL LETTER IN INTEGER CONSTANT. *E* ASSUMED                |
| 10     | W        | FIXED POINT CONSTANT ENDING IN A IS ILLEGAL                    |
| 11     | W        | INTEGER CONSTANT ENDING IN E IS ILLEGAL                        |
| 12     | W        | PRIME IS USED OUT OF ALLOWABLE CONTEXTS                        |
| 13     | W        | MAGNITUDE OF EXPONENT EXCEED FLOATING POINT LIMITS             |
| 14     | W        | ILLEGAL CHARACTER FOUND IN OCTAL CONSTANT                      |
| 15     | W        | IMPROPERLY FORMED LITERAL CONSTANT TRUNCATED AT SIZE SPECIFIED |
| 16     | W        | ILLEGAL STATUS CONSTANT NAME                                   |
| 17     | W        | ILLEGAL JOVIAL SOURCE CHARACTER IS IGNORED                     |
| 18     | F        | NESTED PROC DECLARATIONS ARE NOT ALLOWED                       |
| 19     | W        | PROGRAM HAS EXTRA *END*                                        |
| 20     | W        | _ IS MISSING A FILE TYPE AND/OR SIZE SPECIFICATION             |
| 21     | W        | _ IS MISSING RECORD DESCRIPTION INFORMATION                    |
| 22     | W        | _ HAS AN ILLEGAL STATUS LIST DECLARATION                       |
| 23     | W        | ARRAY/TABLE SPECIFIES MORE ENTRIES THAN COMPILER LIMIT ALLOWS  |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                         |
|--------|----------|-----------------------------------------------------------------|
| 24     | W        | ILLEGAL TABLE DECLARATION HEADER                                |
| 25     | W        | BEGIN IS EXPECTED FOLLOWING TABLE DECLARATION HEADER            |
| 26     | W        | TOO MANY DIMENSIONS SPECIFIED FOR ARRAY                         |
| 27     | W        | DIMENSION LIST MISSING OR INCORRECTLY SPECIFIED FOR ARRAY       |
| 28     | E        | _ ILLEGAL OCCURRENCE OF LABEL                                   |
| 29     | W        | _ DECLARES MORE STATUS CONSTANTS THAN SIZE SPECIFICATION ALLOWS |
| 30     | W        | _ CANNOT BE DECLARED IMPLICITLY WITH A STATUS CONSTANT          |
| 31     | W        | PROC DECLARATION LACKS NAME                                     |
| 32     | W        | ITEM DECLARATION LACKS NAME                                     |
| 33     | W        | ARRAY DECLARATION LACKS NAME                                    |
| 34     | W        | FILE DECLARATION LACKS NAME                                     |
| 35     | W        | SWITCH DECLARATION LACKS NAME                                   |
| 36     | W        | CLOSE DECLARATION LACKS NAME                                    |
| 37     | W        | PROGRAM DECLARATION LACKS NAME                                  |
| 38     | W        | DEFINE DIRECTIVE LACKS NAME                                     |
| 39     | W        | TABLE DECLARATION TERMINATED ABNORMALLY                         |
| 40     | W        | STATEMENT HAS MISSING DOLLAR SIGN                               |
| 41     | F        | PROGRAM HAS MISSING ENDS                                        |
| 42     | F        | PROGRAM WAS NOT COMPLETED ON A MAIN SCOPE                       |
| 43     | W        | _ EXCEEDS MAXIMUM BIT SIZE ALLOWED FOR UNSIGNED VARIABLE        |
| 44     | W        | ILLEGAL LETTER IN FIXED CONSTANT. ** ASSUMED                    |
| 45     | E        | _ MUST HAVE SUBSCRIPTS ATTACHED WHEN REFERENCED                 |
| 46     | E        | _ IS A SIMPLE ITEM AND SHOULD NOT BE SUBSCRIPTED                |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                      |
|--------|----------|--------------------------------------------------------------|
| 47     | E        | TOO MANY RIGHT PARENTHESES IN A BOOLEAN FORMULA              |
| 48     | E        | PARENTHESES DO NOT MATCH IN BOOLEAN FORMULA                  |
| 49     | W        | ONLY EQ OR NQ ARE LEGAL IN A HOLLERITH RELATION FORMULA      |
| 50     | E        | _ HAS AN INCORRECT NUMBER OF SUBSCRIPTS ATTACHED             |
| 51     | W        | NAME STRING IS LONGER THAN COMPILER LIMIT                    |
| 52     | W        | LITERAL CONSTANT LONGER THAN SPECIFIED LIMIT                 |
| 53     | W        | _ EXCEEDS MAXIMUM BYTE SIZE ALLOWED FOR LITERAL VARIABLE     |
| 54     | W        | _ EXCEEDS MAXIMUM BIT SIZE ALLOWED FOR SIGNED VARIABLE       |
| 55     | W        | _ EXCEEDS MAXIMUM SCALING FACTOR ALLOWED FOR FIXED VARIABLES |
| 56     | E        | _ IS AN INCORRECT USAGE OF NAME IN GOTO STATEMENT            |
| 57     | E        | _ IS UNDEFINED OR INACCESSIBLE FROM POINT OF GOTO            |
| 58     | E        | _ SHOULD NOT BE SUBSCRIPTED WHEN CALLED OR REFERENCED        |
| 59     | E        | _ IS A CALL ON AN UNDEFINED SWITCH                           |
| 60     | E        | EXTRANEIOUS DATA FOLLOWS A LEGAL DECLARATION                 |
| 61     | E        | _ SHOULD BE SUBSCRIPTED WHEN CALLED OR REFERENCED            |
| 62     | W        | _ IS AN UNDEFINED OR INACCESSIBLE SWITCH POINT               |
| 63     | W        | _ TREATED AS IDENTIFIER AND NOT AS A RESERVED WORD           |
| 64     | W        | CONSTANT IS NOT COMPATIBLE WITH TYPE OF ITEM SWITCH          |
| 65     | E        | _ IS AN ILLEGALLY REFERENCED STATUS CONSTANT                 |
| 66     | W        | _ COMMA ASSUMED FOLLOWING SUBSCRIPT COMPONENT                |
| 67     | W        | ) OR \$) MISSING                                             |
| 68     | E        | ERROR IN *ENTRY* FUNCTIONAL MODIFIER                         |
| 69     | W        | _ IS ALREADY ACTIVE AS A LOOP VARIABLE                       |

(Continued)

TABLE A-2, SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                         |
|--------|----------|-----------------------------------------------------------------|
| 70     | E        | NO OPERAND SPECIFIED IN I/O STATEMENT                           |
| 71     | E        | STATUS CONSTANT ILLEGAL AS OPERAND IN I/O STATEMENT             |
| 72     | E        | CONSTANTS ARE NOT LEGAL IN INPUT I/O STATEMENTS                 |
| 73     | W        | _ IS MISSING A FILE STATUS LIST                                 |
| 74     | W        | _ IS MISSING A DEVICE NAME SPECIFICATION                        |
| 75     | W        | OCTAL CONSTANT TERMINATED BY A DOLLAR SIGN                      |
| 76     | W        | ILLEGAL CHARACTER PRESENT IN LITERAL CONSTANT                   |
| 77     | W        | ) ACCEPTED FOR \$) OR ( FOR (\$                                 |
| 78     | W        | MISSING ) OR ( ASSUMED TO BE PRESENT                            |
| 79     | W        | _ IS NOT AN ITEM IN A TABLE                                     |
| 80     | W        | _ IS AN ILLEGAL NAME IN AN OVERLAY STATEMENT                    |
| 81     | W        | _ PREVIOUS INDIRECT USE OF NAME AS A PATTERN TABLE NOT ACCEPTED |
| 82     | W        | _ IS IMPROPERLY REFERENCED WITHIN A SWITCH LIST                 |
| 83     | W        | _ CAUSES A LOGICAL INCONSISTENCY TO EXIST IN OVERLAY STATEMENT  |
| 84     | W        | _ IS REFERENCED ILLEGALLY IN SUBORDINATE OVERLAY STATEMENT      |
| 85     | W        | COMMENT WAS TERMINATED BY A DOLLAR SIGN                         |
| 86     | W        | PROGRAM IS MISSING A TERM STATEMENT                             |
| 87     | W        | INTERNAL OVERLAY TABLE OVERFLOW. OVERLAYS NO LONGER PROCESSED   |
| 88     | W        | TOO MANY OCTAL DIGITS IN OCTAL CONSTANT                         |
| 89     | W        | I-TYPE SPECIFIED SHOULD BE A-TYPE FOR ITEM_                     |
| 90     | W        | _ UNDEFINED OR NON-GLOBAL LABEL ON TERM STATEMENT IGNORED       |
| 91     | W        | FIRST VALUE SHOULD BE SMALLEST IN RANGE INFORMATION FOR_        |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                                          |
|--------|----------|----------------------------------------------------------------------------------|
| 92     | W        | COMPLEX STATEMENT MAY NOT FOLLOW AN IF CLAUSE                                    |
| 93     | W        | COMPLEX STATEMENT MAY NOT FOLLOW AN IFEITH CLAUSE                                |
| 94     | W        | COMPLEX STATEMENT MAY NOT FOLLOW AN ORIF CLAUSE                                  |
| 95     | W        | STATUS CONSTANT IS NOT TERMINATED BY )                                           |
| 96     | W        | V OR R SPECIFIER IS MISSING FROM TABLE DECLARATION                               |
| 97     | W        | NUMBER OF ENTRIES IS MISSING FROM TABLE DECLARATION                              |
| 98     | W        | _ IS LACKING A SIZE SPECIFICATION                                                |
| 99     | W        | _ IS MISSING AN S OR U SPECIFICATION                                             |
| 100    | E        | NON-STATUS CONSTANT USED IN STATUS ASSIGNMENT STATEMENT                          |
| 101    | W        | _ SPECIFIES A WORD/ENTRY LARGER THAN ENTRY SIZE OF TABLE                         |
| 102    | W        | _ SPECIFIES AN IMPOSSIBLE FIRST BIT                                              |
| 103    | W        | _ IS MISSING A FIRST BIT SPECIFICATION                                           |
| 104    | W        | _ IS MISSING BEAD AND FREQUENCY INFORMATION                                      |
| 105    | W        | _ SPECIFIES A WORD/ENTRY GREATER THAN COMPILER MAXIMUM LIMIT                     |
| 106    | W        | TABLE SPECIFIES MORE WORDS/ENTRY THAN COMPILER LIMIT ALLOWS                      |
| 107    | W        | _ CANNOT BE EXPANDED. CURRENT DEFINE SOURCE WOULD BE LARGER THAN SPACE AVAILABLE |
| 108    | W        | DECLARATION IS MISSING A TYPE SPECIFICATION                                      |
| 109    | E        | ILLEGAL STATEMENT                                                                |
| 110    | W        | PROGRAM IS MISSING A START STATEMENT                                             |
| 111    | W        | ILLEGAL SOURCE IGNORED WITHIN TABLE DECLARATION                                  |
| 112    | W        | IMPROPER FOR CLAUSE                                                              |
| 113    | E        | LOOP VARIABLE NOT FOUND FOLLOWING PRIMITIVE *FOR*                                |
| 114    | W        | COMPLEX STATEMENT MAY NOT FOLLOW A FOR CLAUSE                                    |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                         |
|--------|----------|-----------------------------------------------------------------|
| 115    | W        | BEGIN NOT FOUND FOLLOWING PROC DECLARATION                      |
| 116    | E        | IS A NON-ACTIVE LOOP VARIABLE AND IS REFERENCED ILLEGALLY       |
| 117    | W        | _ DEFINE NAME NOT EXPANDED. MISSING DEFINE STRING               |
| 118    | E        | STOP STATEMENT REFERENCES UNDEFINED OR INACCESSIBLE LABEL_      |
| 119    | W        | _ IS A NON-MONITORABLE NAME                                     |
| 120    | W        | SPECIAL COMPOUND MAY NOT BE USED WITH A 1-FACTOR FOR CLAUSE     |
| 121    | E        | TEST STATEMENT EXISTS OUTSIDE OF A LOOP STATEMENT               |
| 122    | E        | TEST STATEMENT REFERENCES NON-ACTIVE LOOP VARIABLE              |
| 123    | W        | NAME IS MISSING OR TOO SHORT FOR LIKE TABLE                     |
| 124    | E        | _ MUST HAVE ACTUAL PARAMETERS ATTACHED WHEN BEING CALLED        |
| 125    | E        | ORIF CLAUSE NOT PRECEDED BY AN IFEITH CLAUSE                    |
| 126    | W        | _ ACCEPTED AS FUNCTION REFERENCE THOUGH PARENTHESES ARE MISSING |
| 127    |          | Not Used                                                        |
| 128    | E        | _ WAS NOT DECLARED AS HAVING PARAMETERS                         |
| 129    | E        | ILLEGAL OCCURRENCE OF START STATEMENT IGNORED                   |
| 130    | E        | _ CALLED WITH WRONG NUMBER OF INPUT OR OUTPUT PARAMETERS        |
| 131    | E        | _ HAS ACTUAL PARAMETER NOT COMPATIBLE WITH FORMAL PARAMETER     |
| 132    | W        | SOURCE FOLLOWING A TERM STATEMENT IS IGNORED                    |
| 133    | W        | ILLEGAL OCCURRENCE OF *RETURN* TREATED AS A *STOP*              |
| 134    | W        | DECLARATION MISSING DOLLAR SIGN. PRESETS IGNORED                |
| 135    | W        | REFERENCE TO NULL STATUS CONSTANT NAME DELETED                  |
| 136    | E        | ILLEGAL CONSTANT IN LITERAL RELATIONAL EXPRESSION               |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                            |
|--------|----------|--------------------------------------------------------------------|
| 137    | E        | _ OCCURRENCE OF UNDEFINED NAME DELETED                             |
| 138    | E        | _ STATUS CONSTANT IS NOT LEGAL AS AN ACTUAL PARAMETER              |
| 139    | E        | FUNCTIONAL MODIFIER CONSTANT ILLEGAL AS OUTPUT PARAMETER           |
| 140    | E        | ENTRY FUNCTIONAL MODIFIER ILLEGAL AS AN ACTUAL PARAMETER           |
| 141    | W        | A SEPARATOR IS MISSING FROM FORMAL PARAMETER LIST                  |
| 142    | W        | ILLEGAL FORMAL PARAMETER LIST IS SPECIFIED                         |
| 143    | W        | AN ERROR EXISTS IN THE MONITOR STATEMENT                           |
| 144    | E        | CONSTANT USED AS LEFT SIDE VARIABLE IN EXCHANGE STATEMENT          |
| 145    | E        | EXCHANGE STATEMENT HAS INCOMPATIBLE LEFT AND RIGHT SIDES           |
| 146    | E        | EXCHANGE STATEMENT HAS AN ILLEGAL RIGHT SIDE                       |
| 147    | E        | INTEGER FORMULA MAY NOT BE COMPARED WITH A LITERAL FORMULA         |
| 148    | E        | _ HAS AN UNDECLARED FORMAL PARAMETER ILLEGAL CALL                  |
| 149    |          | Not Used                                                           |
| 150    | W        | DEFINE STRING TOO LONG--NOT ACCEPTED                               |
| 151    | W        | RESULT FRACTION SIZE EXCEEDS 48 BITS FOR DIVISION/MULTIPLICATION   |
| 152    | W        | _ MUST BE EXPLICITLY DECLARED BEFORE ITS USE AS A FORMAL PARAMETER |
| 153    | W        | START \$ REQUIRED FOLLOWING STAND ALONE CLOSE COMPILATION          |
| 154    | W        | LABEL FOLLOWING TERM LEGAL ONLY IN MAIN PROGRAMS                   |
| 155    | W        | _ HAS AN IMPROPERLY FORMED PRESET CONSTANT LIST: PRESETS IGNORED   |
| 156    | W        | TOO MANY PRESET VALUES STATED FOR A ROW OF ARRAY _                 |

(Continued)



TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                         |
|--------|----------|-----------------------------------------------------------------|
| 157    | W        | MORE PRESETS THAN POSSIBLE STATED FOR ITEM_                     |
| 158    | W        | PRESET VALUE MISSING OR NON-RECOGNIZABLE FOR ITEM_              |
| 159    | W        | UNDECLARED STATUS CONSTANT USED AS PRESET VALUE FOR_            |
| 160    | W        | PRESET VALUE IS INCOMPATIBLE WITH TYPE OF ITEM_                 |
| 161    | W        | OCTAL CONSTANT BIGGER THAN WORD NOT A LEGAL PRESET FOR_         |
| 162    | W        | PRESET PROCESSING TERMINATED. ILLEGAL INPUT FOR ITEM_           |
| 163    | E        | LITERAL CONSTANTS OF LENGTH ZERO ARE ILLEGAL                    |
| 164    | E        | AN OCTAL CONSTANT OF ZERO LENGTH IS ILLEGAL                     |
| 165    | W        | _PRESET IGNORED. ARRAY, TABLE ITEM FORMAL PARAMETERS NOT PRESET |
| 166    | W        | _ACCEPTED AS AN INDEX SWITCH ALTHOUGH DECLARATION ERRORS EXISTS |
| 167    | W        | _ACCEPTED AS AN ITEM SWITCH ALTHOUGH DECLARATION ERROR EXISTS   |
| 168    | E        | _CANNOT BE USED WITH THE NWDSFN FUNCTIONAL MODIFIER             |
| 169    | E        | _MUST BE A FILE NAME IF USED WITH THE POS FUNCTIONAL MODIFIER   |
| 170    | E        | _CANNOT BE USED WITH THE NENT FUNCTIONAL MODIFIER               |
| 171    | E        | CHAR AND MANT MAY ONLY BE APPLIED TO FLOATING VARIABLES         |
| 172    | E        | THE ODD MODIFIER MAY NOT BE APPLIED TO FLOATING VARIABLES       |
| 173    | E        | FUNCTIONAL MODIFIER APPLIED TO AN ILLEGAL TYPE VARIABLE         |
| 174    | E        | FUNCTIONAL MODIFIER ON LEFT SIDE IS NOT A VARIABLE              |
| 175    | E        | ILLEGAL OCCURRENCE OF BIT/BYTE DELETED                          |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                            |
|--------|----------|--------------------------------------------------------------------|
| 176    | E        | _ ILLEGAL OCCURRENCE OF RESERVED WORD DELETED                      |
| 177    | E        | ILLEGAL OCCURRENCE OF ENTRY VARIABLE                               |
| 178    | W        | _ IS NOT DECLARED WITHIN THE SCOPE OF THE MONITOR STATEMENT        |
| 179    |          | Not Used                                                           |
| 180    | W        | _ IS MISSING AN ASSOCIATED STATUS CONSTANT LIST                    |
| 181    | W        | _ IS MISSING BOTH A SIZE AND A SIGN DECLARATION                    |
| 182    | E        | CONSTANT TYPE IS ILLEGAL FOR ASSIGNMENT TO LITERAL VARIABLE        |
| 183    | W        | PARENTHESES WERE FOUND TO BE NON-MATCHING IN STATEMENT             |
| 184    | W        | BRACKETS WERE FOUND TO BE NON-MATCHING IN STATEMENT                |
| 185    | W        | CONSTANT BEING ASSIGNED TO ENTRY VARIABLE IS NOT ZERO              |
| 186    | W        | CONSTANT BEING ASSIGNED TO BOOLEAN VARIABLE IS NOT ZERO OR ONE     |
| 187    | E        | _ IS AN UNDEFINED OR INACCESSIBLE LABEL OR PROGRAM NAME            |
| 188    | W        | TWO FACTOR FOR CLAUSE NOT LEGAL PRECEDING A COMPLETE CLAUSE        |
| 189    | W        | ONLY ONE COMPLETE FOR CLAUSE IS LEGAL IN A COMPLETE LOOP STATEMENT |
| 190    | W        | EXTRANEIOUS SOURCE NOT PROCEEDED FOLLOWING LEGAL STATEMENT         |
| 191    | W        | _ DOES NOT HAVE A TYPE SPECIFICATION                               |
| 192    | W        | BEGIN MUST IMMEDIATELY FOLLOW DECLARATION FOR THE CLOSE _          |
| 193    | W        | STRING DECLARATION IS NOT WITHIN A DEFINED ENTRY TABLE DECLARATION |
| 194    | E        | GOTO IS NOT FOLLOWED BY A LEGAL IDENTIFIER                         |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                            |
|--------|----------|--------------------------------------------------------------------|
| 195    |          | Not Used                                                           |
| 196    | W        | RESERVED WORD OVERLAY IS NOT FOLLOWED BY A LEGAL NAME              |
| 197    | W        | AN ERROR EXISTS IN THE OVERLAY STATEMENT                           |
| 198    | W        | AN ERROR EXISTS IN THE INDEX SWITCH LIST OF _                      |
| 199    | W        | AN ERROR EXISTS IN THE ITEM SWITCH LIST OF _                       |
| 200    | W        | CONSTANT BEING ASSIGNED TO ENTRY VARIABLE IS NOT A ZERO            |
| 201    | W        | ONLY EQ OR NQ ARE LEGAL IN AN ENTRY RELATIONAL FORMULA             |
| 202    | W        | CONSTANT USED AS A BOOLEAN FORMULA IS ILLEGAL                      |
| 203    | E        | _ UNDEFINED LABEL OR CLOSE NAME IN ACTUAL PARAMETER LIST           |
| 204    | W        | ILLEGAL SINGLE LETTER WITHIN DECLARATION IGNORED                   |
| 205    | E        | UNRECOGNIZABLE SOURCE OR PARTIAL STATEMENT                         |
| 206    | W        | _ IS DECLARED WITH AN ORDINARY ITEM DECLARATION IN A DEFINED ENT   |
| 207    | W        | ALL OF THE REQUIRED STRING SPECIFIERS ARE MISSING FOR _            |
| 208    | W        | _ HAS A FREQUENCY VALUE THAT EXCEEDS COMPILER LIMIT                |
| 209    | W        | _ HAS A BEADS/WORD VALUE THAT EXCEEDS COMPILER LIMIT               |
| 210    | W        | _ ALREADY APPEARS AS A FORMAL PARAMETER TO THIS PROCEDURE/FUNCTION |
| 211    | W        | _ CANNOT BE CONTAINED IN A SINGLE WORD AS DECLARED                 |
| 212    | W        | _ IS DECLARED WITH AN IMPOSSIBLE BEAD/WORD SPECIFICATION           |
| 213    | W        | _ MUST BE DECLARED WITH A FIRST BIT FALLING ON A BYTE BOUNDARY     |
| 214    |          | Not Used                                                           |
| 215    | E        | _ IS AN IMPROPER USE OF A TABLE NAME                               |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                                                       |
|--------|----------|---------------------------------------------------------------|
| 216    | E        | _ IS AN IMPROPER USE OF A FILE NAME                           |
| 217    |          | Not Used                                                      |
| 218    |          | Not Used                                                      |
| 219    |          | Not Used                                                      |
| 220    | E        | TOO MANY INDICES FOR BIT/BYTE MODIFIER                        |
| 221    | W        | PACKING SPECIFICATION IGNORED FOR DEFINED ENTRY TABLE         |
| 222    | W        | ILLEGAL SCALING SPECIFIED FOR FIXED POINT CONSTANT            |
| 223    |          | Not Used                                                      |
| 224    |          | Not Used                                                      |
| 225    | W        | _ LOOP VARIABLE REFERENCE IN COMPOOL DELETED                  |
| 226    | W        | _ LABEL DEFINITION IN COMPOOL DELETED                         |
| 227    | W        | _ IDENTIFIER REFERENCE IN COMPOOL DELETED                     |
| 228    | W        | ILLEGAL SYMBOL IN COMPOOL DELETED                             |
| 229    | W        | ILLEGAL SOURCE IN COMPOOL DELETED                             |
| 230    | E        | ILLEGAL STATEMENT                                             |
| 231    | W        | DECLARATIVE STATEMENT NOT LEGAL OUTSIDE OF COMMON BLOCK       |
| 232    | W        | BEGIN* EXPECTED FOLLOWING A COMMON BLOCK DECLARATION          |
| 233    | W        | _ CONFLICTS WITH PREVIOUS DECLARATION FOR THIS NAME           |
| 234    | W        | _ COMMON BLOCK TERMINATED TO PREVENT NESTING OF COMMON BLOCKS |
| 235    | W        | LABEL TOO LONG                                                |
| 236    | W        | LABEL NOT TERMINATED BY BLANK                                 |
| 237    | W        | DUPLICATE LABEL_                                              |
| 238    | W        | OPERATION CODE ERROR                                          |

(Continued)

TABLE A-2. SOURCE DIAGNOSTIC MESSAGES (Cont'd)

| Number | Severity | Message                       |
|--------|----------|-------------------------------|
| 239    | W        | ERROR***CARD IGNORED          |
| 240    | W        | VARIABLE FIELD ERROR          |
| 241    | W        | MISSING \$                    |
| 242    | W        | ASSIGN STATEMENT SYNTAX ERROR |

### TERMINATION MESSAGES

A termination message is issued when certain source conditions will cause compilation to be terminated prematurely. This termination will be indicated by a message of the form:

COMPILATION TERMINATION NO. number: message

The number consists of three or four digits. The first two digits indicates the phase during which compilation occurred. The last two digits are assigned uniquely within each phase.

The first or first two digits are assigned as follows:

- 0 Cradle and initialization
- 1 JOVIAL analyzer, pass 1, and COMPOOL assembler
- 2 Allocator, COMPOOL resolver, and editor
- 3 JOVIAL analyzer, pass 2
- 4 Diagnostic processor
- 5 Global optimizer, pass 1
- 6 Global optimizer, pass 2
- 7 Code generator
- 8 Scheduler
- 9 Editor
- 10 Map
- 11 Cross-reference

Except for the messages indicating that a user correction can be made, all compiler abort messages should be reported to CDC as they indicate a compiler malfunction. Table A-3 lists the termination messages in numerical order.

TABLE A-3. TERMINATION MESSAGES

| Number | Message                                                                  | Explanation                                                             |
|--------|--------------------------------------------------------------------------|-------------------------------------------------------------------------|
| 5      | FIND LOOP                                                                |                                                                         |
| 6      | SYMBOL TABLE OVERFLOW                                                    | Resubmit with larger field length.                                      |
| 100    | MALFORMED SYNTAX TABLES                                                  |                                                                         |
| 101    | LEFT END OF C.S. REACHED ILLEGALLY                                       |                                                                         |
| 102    | CONSTRUCT STRING OVERFLOW                                                | Symbols in statement probably exceeds compiler limit                    |
| 110    | PTPSET TABLE OVERFLOW                                                    |                                                                         |
| 120    | F.A.T. OVERFLOW FOR PRESETS                                              | Too many preset arrays.                                                 |
| 150    | NUMBER OF PROCS/FUNCS EXCEEDS COMPILER LIMIT                             |                                                                         |
| 290    | TOO MANY NAMES IN COMPOOL – COMPILATION ABORTED                          |                                                                         |
| 298    | REQUESTED BLOCK IS GREATER THAN MAXIMUM BLOCK NUMBER IN THE COMPOOL FILE | Probable malformed COMPOOL file.                                        |
| 299    | COMPOOL RESOLVER REQUEST FOR BLOCK FROM COMPOOL FILE OUT OF SEQUENCE     | Probable malformed COMPOOL file.                                        |
| 300    | MALFORMED SYNTAX TABLES                                                  |                                                                         |
| 302    | CONSTRUCT STRING OVERFLOW                                                | Symbols in statement probably exceed compiler limit.                    |
| 303    | UNEXPECTED EOF ON CONSTRUCT STRING                                       | Probable system I/O ERROR; resubmit, if recurs continually, notify CDC. |
| 310    | IL F.A.T. OVERFLOW                                                       |                                                                         |
| 330    | PROC/FUNC/CLOSE NESTING LIMIT EXCEEDED                                   |                                                                         |
| 340    | SAVE CONTROL TABLE OVERFLOW                                              |                                                                         |
| 341    | WDT TABLE OVERFLOW                                                       |                                                                         |
| 342    | SPURIOUS ENDSAV REQUESTED                                                |                                                                         |
| 343    | RESTR, OSAV FOR NON-EXISTENT SAVE BUFFER                                 |                                                                         |
| 344    | RESTR OF A CURRENT SAVE LIST                                             |                                                                         |

(Continued)

TABLE A-3. TERMINATION MESSAGES (Cont'd)

| Number | Message                                                           | Explanation                                                              |
|--------|-------------------------------------------------------------------|--------------------------------------------------------------------------|
| 350    | ALTERNATIVE STATEMENTS NESTED TOO DEEPLY                          | Reduce Nesting                                                           |
| 351    | IF/ORIF FALSE LABEL STACK OVERFLOW                                | Reduce nesting by use of temporary variable to hold intermediate result. |
| 352    | SUBSCRIPTS TOO DEEPLY NESTED                                      |                                                                          |
| 355    | FUNC CALLS NESTED TOO DEEPLY                                      |                                                                          |
| 356    | GENERATED LABEL STACK OVERFLOW                                    |                                                                          |
| 400    | TOO MANY SOURCE DIAGNOSTICS                                       | Correct and resubmit.                                                    |
| 501    | OPT1 - OPERAND STACK OVERFLOW                                     | Statement in program is too complex.                                     |
| 502    | OPT1 - OPERAND STACK UNDERFLOW - COMPILER ERROR                   |                                                                          |
| 503    | OPT1 - ILLEGAL IL OPERATOR - COMPILER ERROR                       |                                                                          |
| 504    | OPT1 - RESIDUE LOOPS AT PROCEDURE TERMINATION                     |                                                                          |
| 505    | OPT1 - LOOP STACK OVERFLOW                                        |                                                                          |
| 506    | OPT1 - LOOP ENTRY NOT POSTED - UNABLE TO CLOSE LOOP               |                                                                          |
| 507    | OPT1 - LOOP STACK UNDERFLOW - COMPILER ERROR                      |                                                                          |
| 608    | OPT2 - NAME STACK OVERFLOW                                        | Statement in program too complex.                                        |
| 609    | OPT2 - PROC CALL STACK OVERFLOW                                   |                                                                          |
| 610    | OPT2 - LIST STACK OVERFLOW                                        |                                                                          |
| 611    | OPT2 - INDUCTION VARIABLE STACK OVERFLOW                          |                                                                          |
| 612    | OPT2 - WINDOW OVERFLOW - ALL IL SPILLED OPTIMIZATION DISCONTINUED |                                                                          |
| 700    | FILTRD ROUTINE FALL-THRU                                          |                                                                          |
| 701    | STACK UNDERFLOW                                                   |                                                                          |
| 702    | UNEXPECTED END-OF-FILE                                            |                                                                          |
| 703    | ILLEGAL IL SEEN                                                   |                                                                          |

(Continued)

TABLE A-3. TERMINATION MESSAGES (Cont'd)

| Number | Message                             | Explanation                                                                       |
|--------|-------------------------------------|-----------------------------------------------------------------------------------|
| 704    | STACK OVERFLOW                      |                                                                                   |
| 705    | TRIAD TABLE OVERFLOW                |                                                                                   |
| 706    | STACK NON-EMPTY AT BREAKPOINT       |                                                                                   |
| 707    | BAD SWITCH STATE INDEX              |                                                                                   |
| 708    | ZERO-DIVIDE ATTEMPT                 |                                                                                   |
| 709    | NON-EXISTENT VALU DELETION ATTEMPT  |                                                                                   |
| 710    | NON-EXISTENT VALU RETRIEVAL ATTEMPT |                                                                                   |
| 711    | LOOP STACK OVERFLOW                 |                                                                                   |
| 712    | LOOP STACK UNDERFLOW                |                                                                                   |
| 713    | ILLEGAL ERROR MESSAGE NUMBER        |                                                                                   |
| 714    | ILLEGAL LEFT SIDE                   |                                                                                   |
| 715    | VALB OF LOCALLY SAVED VALUE         |                                                                                   |
| 716    | STACK HISTORY OVERFLOW              |                                                                                   |
| 717    | INFINITE OPERAND                    |                                                                                   |
| 718    | ZERO STACK PTR                      |                                                                                   |
| 719    | INDEFINITE OPERAND                  |                                                                                   |
| 720    | FILE STATUS VALUE OUT OF RANGE      | The file status value is greater than three, recompile with correct status value. |
| 721    | CONSTANT VALUE FILE RELATIONAL      |                                                                                   |
| 722    | PARAM STACK NON-EMPTY               |                                                                                   |
| 723    | PARAM STACK OVERFLOW                |                                                                                   |
| 724    | PARAM STACK UNDERFLOW               |                                                                                   |
| 802    | ILLEGAL ICF OPCODE                  |                                                                                   |
| 803    | ILLEGAL SEQUENCE TERMINATING OPCODE |                                                                                   |

(Continued)



TABLE A-3. TERMINATION MESSAGES (Cont'd)

| Number | Message                                            | Explanation                                                                                                                                                                                                        |
|--------|----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 804    | REPL OR LOAD HAS ICF POINTER FOR OPN1              | Which does not point to SUBS or OFFS.                                                                                                                                                                              |
| 805    | TEMP CANNOT BE FOUND FOR ICF VALUE                 |                                                                                                                                                                                                                    |
| 806    | NO PTRM IN ICF                                     |                                                                                                                                                                                                                    |
| 850    | ICFT FULL                                          | The Scheduler has attempted to schedule too large a block of code. Use the single statement scheduling parameter (W) or reduce the size of block, and place a dummy label in the block with a GOTO after the STOP. |
| 851    | LOST COMPUTATION                                   |                                                                                                                                                                                                                    |
| 852    | EMPTY READY SET                                    |                                                                                                                                                                                                                    |
| 900    | NUMBER OF COMMON BLOCKS EXCEEDS SYSTEM LIMIT OF 61 |                                                                                                                                                                                                                    |

## SAMPLE DECK SETUPS

B

This appendix contains sample deck setups for ordinary compilations. Sample programs for COMPOOL compilations are shown in Appendixes H and I. Monitor compilation examples are shown in Section 9. Appendix M gives examples of overlays.

This example (Figure B-1) is compiled with the source, object code, cross reference, and storage map listings, and binary output is placed on the default file LGO; program execution is accomplished with data from file INPUT. Optimization has not been specified.

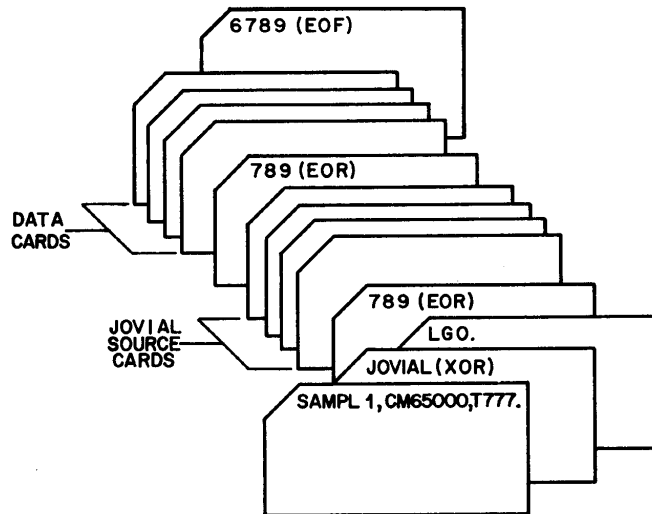


Figure B-1. Compile And Execute With INPUT File

This example (Figure B-2) is compiled with a syntax check from a program maintained on an UPDATE program library. Correction cards are input to UPDATE to produce the desired COMPILE file, which is then used as the input to the JOVIAL compiler.

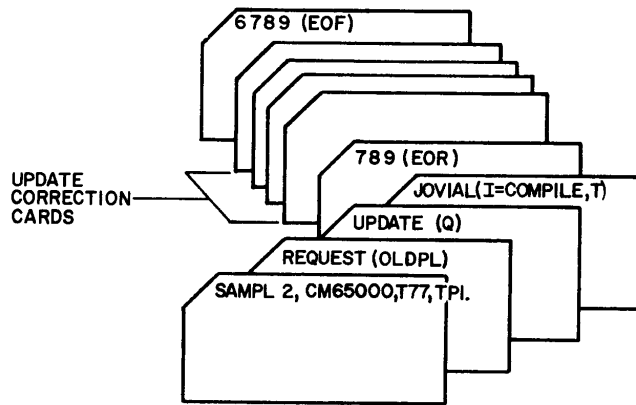


Figure B-2. Syntax Check Program Library Corrections

This example (Figure B-3) is compiled with source, storage map, and cross reference listings, and binary output is placed in file SVE, with entry point SUM. Optimization has not been specified.

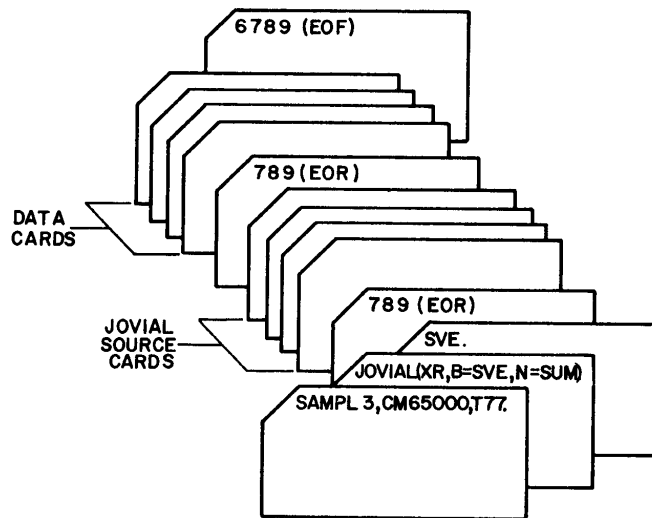


Figure B-3. Compile With Binary on SVE

This example (Figure B-4) is compiled with optimization specified. The object code, storage map, and cross reference listing is output, but no execution occurs. A binary check is produced to be saved for later use.

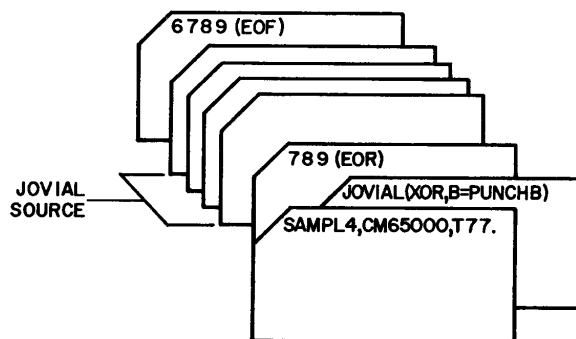


Figure B-4. Compilation to Produce Binary Deck

## COMPILER LIMITATION AND RESTRICTIONS

C

---

This appendix provides a summary of the limitations and restrictions placed upon the programmer by implementation of the JOVIAL(J3) language. Some of these limitations reflect system hardware features; others represent restrictions generated by compiler design; still others were deemed practical and unburdensome limits.

- Constants - The number of characters in a Hollerith or Transmission code constant is limited to 250; the maximum number of digits in an octal constant is 500.
- Data - Fixed point data and constant scale factors can be between -59 and +59.
- Names - There can be 30 signs in any name; the first five characters of external procedure names must be unique; the first six characters of file device names must be unique; a defined name can not expand to more than 320 characters.
- Statements - There can be at least 100 symbols in any one statement, the exact number depends on the complexity of the statement.
- Tables - The number of table entries is limited to  $2^{17}-1$ ; there can be no more than 2047 words per table entry and 31 beads per word of a string item. The first 5800 words of a table can be preset.
- Arrays - An array can be  $2^{17}-1$  words. In one source program 25 arrays can be preset; the first 5800 words of an array can be preset.
- Strings - No defined string may contain more than 250 characters.
- Overlays - The maximum number of names allowed in overlay statements is 2000, minus the number of overlay statements.
- Subscripts - The number of subscripts that may be applied to an array is limited to seven.
- Nesting - There can be 25 nested BEGIN-END brackets; 50 nested IF statements; 20 nested IFEITH statements; 20 nested function calls; 20 nested subscripts.

- Diagnostics - There can be 200 diagnostic messages produced in any compilation and a total of 200 for the pass two summary. If the D option is specified on the JOVIAL control card, Phase 1 will continue until all source has been scanned and diagnostics issued.
- The number of local PROCs, CLOSEs, and blocks of direct code per compilation is limited to 2200.
- Symbol table capacity is determined by the user's field length. The larger the field length, the larger the symbol table.
- The storage map table will hold 6000 names. Storage maps and cross reference listings may not be produced for programs with more than 6000 names.
- Common blocks - The maximum number of common blocks for which a cross-reference or storage map can be generated (X, R, or Q control card options) is 61.

## HINTS ON COMPILER USE

D

Listed below are suggestions by which the writer of a source program can utilize the compiler to produce more efficient code.

- In a defined table, the numeric item declarations are more or less efficient according to the definitions shown in Table D-1. Priority 1 produces the most efficient code, priority 6 the least.

TABLE D-1. EXTRACTING A SOURCE FIELD

| Priority | Source Type | First Bit         | Number of Bits | Instructions Req.              |
|----------|-------------|-------------------|----------------|--------------------------------|
| 1        | S           | 42                | 18             | 0 or 1                         |
| 2        | S           | 0                 | <60            | 1                              |
| 3        | S           | <42               | 18             | 2:different units <sup>†</sup> |
| 4        | U           | 60-number of bits | <60            | 2:different units <sup>†</sup> |
| 5        | S           | <60               | <60            | 2:same units                   |
| 6        | U           | <60               | <60            | 3                              |

<sup>†</sup>On 6400 computers, the 2 instructions are processed sequentially.

- BIT rather than BYTE should be used where possible; this eliminates the blank padding.
- EOR loop increments and termination expressions should be invariant throughout a loop. Where loop direction is unimportant, decrementing down to 0 is more efficient than incrementing up from 0.
- The program flow should be essentially straight-line and forward for optimum processing. Single reference code bodies should be coded in-line rather than as procedures. Global references are more efficient than parameters.
- OVERLAYS hamper scheduling and global optimization.
- AND and OR combined relationals are sometimes less efficient than successive IF statements.

- Successive negations are not combined.
- Where possible, numeric types should match to minimize conversions. Since multiplications and divisions must be done in floating-point, floating type variables are probably more efficient in expressions.
- Expressions which result in a Boolean value as the right side of an assignment statement are more efficiently coded as conditionally executed assignment statements.
- Better optimization is performed when procedures are compiled with the caller.
- Use of functions in long expressions and parameters should be minimized.
- Valid switch points should be grouped in the center of the switch; null points on either end are optimized.
- Item sizes should be kept as small as possible; sizes 18 bits or less make better use of the instruction repertoire and machine capability.
- Better code is generated for a conditionally executed block of code than for a conditional GOTO around a block of code.
- Calls to PROCEDURES and CLOSEs hamper register memory and may, therefore, cause less efficient code to be generated. Programs with many such calls may be more efficient if the global optimizer is not used. Besides calls written by the user the compiler generates calls to JOVIAL library routines for I/O statements, monitoring variables, and for literal moves and compares when at least one of the literal variables involved crosses a word boundary.
- Comparisons between Hollerith and transmission code literals should not be performed because the variables are padded, as required, with their own types of blanks before comparison is made.
- Multiplication and division containing fixed-point data should be carefully analyzed as only the rightmost 48 bits of precision are available on any intermediate results.



## CALLING SEQUENCES AND ERROR TRACING

E

---

The calling sequence used by the JOVIAL compiler for calling procedures and functions not compiled with the calling programs depends on whether or not the procedure or function is defined as JOVIAL. All procedures and functions called by a JOVIAL program are assumed to be JOVIAL routines except in the following cases:

- The routine is a library subprogram (as defined in Appendix G)
- The routine was defined in COMPOOL and contained no parameter declarations.

The calling sequences generated by the JOVIAL compiler for non-JOVIAL routines are compatible with FORTRAN Extended and allow the FORTRAN Extended library traceback to be used. The parameter list consists of a list of the addresses of the parameters (one address for each parameter), terminated by a word of zeros. The subprogram is entered via an RJ instruction. Upon entry, the address of the parameter list is contained in register A1 and the first word in the parameter list is in register X1. Function results will be found in register X6. For character functions longer than 10 characters, the address of the result will be in register X6.

The distinction between input and output parameters is meaningless for non-JOVIAL routines. The equal sign, used to separate input and output parameters for JOVIAL procedure calls, should not be used in calls to non-JOVIAL routines. All parameters are passed by name for non-JOVIAL routine calls. Simple items, array and table names may be used as both input and output parameters. For single occurrences of array and table items, table entries, and functional modifiers, the compiler moves the data to a temporary storage location which is used in the call, and thus, may be used as input parameters only. To use them as output parameters, the programmer must move them to a simple item, or if possible, to an overlaid simple item. Overlaid items may be used only in the case of array items, serial table items which are not packed, or serial table entries. If a move to a simple item is used, the data must be moved back after the routine call.

If the routine being called is of type JOVIAL, only arrays, tables, and CLOSEs will be passed in a parameter list. Value parameters are passed by assignment statements before (for input) and after (for output) the call. These assignment statements will reference the formal parameters directly. To facilitate this technique, formal value parameters to independent procedures and functions are given a unique name, based on the procedure name, and defined

externally. The naming convention consists of the first five characters of the procedure (function) name followed by a \$ and a letter (A-Z) or a number (0-9) depending on the value parameter position; i. e., name\$A for the first formal value parameters, name\$B for the second, etc. The name is extended on the right with \$s if it is less than five characters.

For external procedures with label parameters, a data item is created with a name similar to the above except the 7th character is a \$. This item is also externally defined. A return code is assigned to this item to indicate which return was invoked; 0 for a normal return, 1 for the first label parameter, 2 for the second, etc.

When a JOVIAL procedure or function is called, it is assumed that it is called by a JOVIAL routine. Therefore, the calling program must conform to the JOVIAL calling conventions just described. It is possible to call a JOVIAL routine from a FORTRAN Extended program if all parameters in the JOVIAL formal parameter list are defined as name parameters. This is accomplished by converting all parameters to arrays and tables. Single value parameters passed by FORTRAN Extended should be declared as one-dimensional JOVIAL arrays of length 1.

In addition to the above conventions, JOVIAL programs protect the contents of A0 across calls in order to maintain FORTRAN Extended compatibility.

Other differences the programmer calling FORTRAN must be aware of are:

- True and false are represented differently in the two languages; 1 and 0, respectively, in JOVIAL, -1 and 0 in FORTRAN.
- JOVIAL does not have double precision or complex types.
- JOVIAL actual parameters are converted to agree with the formal parameters; parameters to non-JOVIAL routines are not converted since no attributes are declared for them in COMPOOL.
- JOVIAL subscripts are 0 based; FORTRAN subscripts and the computed GOTO is 1 based.
- The unary minus has a higher precedence than exponentiation in JOVIAL; the opposite is true in FORTRAN.
- JOVIAL has no equivalent to the assigned GOTO.
- The rules for DO loops and FOR loops are somewhat different. Refer to the respective language manuals for the exact differences.

# CHARACTER SET

F

Table F-1 shows the octal and decimal representations of Hollerith (display code) and STC (transmission code) in cross reference format.

TABLE F-1. CHARACTER SET

| Octal | Hollerith | Transmission Code | Decimal | Octal | Hollerith | Transmission Code | Decimal |
|-------|-----------|-------------------|---------|-------|-----------|-------------------|---------|
| 0     |           |                   |         | 40    | 5         | )                 | 32      |
| 1     | A         |                   | 1       | 41    | 6         | -                 | 33      |
| 2     | B         |                   | 2       | 42    | 7         | +                 | 34      |
| 3     | C         |                   | 3       | 43    | 8         |                   | 35      |
| 4     | D         |                   | 4       | 44    | 9         | =                 | 36      |
| 5     | E         | BLK               | 5       | 45    | +         |                   | 37      |
| 6     | F         | A                 | 6       | 46    | -         |                   | 38      |
| 7     | G         | B                 | 7       | 47    | *         | \$                | 39      |
| 10    | H         | C                 | 8       | 50    | /         | *                 | 40      |
| 11    | I         | D                 | 9       | 51    | (         | (                 | 41      |
| 12    | J         | E                 | 10      | 52    | )         |                   | 42      |
| 13    | K         | F                 | 11      | 53    | \$        |                   | 43      |
| 14    | L         | G                 | 12      | 54    | =         |                   | 44      |
| 15    | M         | H                 | 13      | 55    | BLK       |                   | 45      |
| 16    | N         | I                 | 14      | 56    | ,         | ,                 | 46      |
| 17    | O         | J                 | 15      | 57    | .         |                   | 47      |
| 20    | P         | K                 | 16      | 60    | ≡         | 0                 | 48      |
| 21    | Q         | L                 | 17      | 61    | [         | 1                 | 49      |
| 22    | R         | M                 | 18      | 62    | ]         | 2                 | 50      |
| 23    | S         | N                 | 19      | 63    | :         | 3                 | 51      |
| 24    | T         | O                 | 20      | 64    | ≠         | 4                 | 52      |
| 25    | U         | P                 | 21      | 65    | ↗         | 5                 | 53      |
| 26    | V         | Q                 | 22      | 66    | v         | 6                 | 54      |
| 27    | W         | R                 | 23      | 67    | ^         | 7                 | 55      |
| 30    | X         | S                 | 24      | 70    | †         | 8                 | 56      |
| 31    | Y         | T                 | 25      | 71    | ‡         | 9                 | 57      |
| 32    | Z         | U                 | 26      | 72    | <         | ≠                 | 58      |
| 33    | 0         | V                 | 27      | 73    | >         |                   | 59      |
| 34    | 1         | W                 | 28      | 74    | ≤         | /                 | 60      |
| 35    | 2         | X                 | 29      | 75    | ≥         | .                 | 61      |
| 36    | 3         | Y                 | 30      | 76    | ⌋         |                   | 62      |
| 37    | 4         | Z                 | 31      | 77    | ;         |                   | 63      |

Table F-2 shows a cross-reference of Hollerith, octal, card punch (26 and 29), and external BCD representations.

TABLE F-2. CROSS-REFERENCE REPRESENTATIONS

| Hollerith | Octal | Card Punch (26) | Card Punch (29) | External BCD | Hollerith | Octal | Card Punch (26) | Card Punch (29) | External BCD       |
|-----------|-------|-----------------|-----------------|--------------|-----------|-------|-----------------|-----------------|--------------------|
| A         | 01    | 12-1            | 12-1            | 61           | 7         | 42    | 7               | 7               | 07                 |
| B         | 02    | 12-2            | 12-2            | 62           | 8         | 43    | 8               | 8               | 10                 |
| C         | 03    | 12-3            | 12-3            | 63           | 9         | 44    | 9               | 9               | 11                 |
| D         | 04    | 12-4            | 12-4            | 64           | +         | 45    | 12              | 12-8-6          | 60                 |
| E         | 05    | 12-5            | 12-5            | 65           | -         | 46    | 11              | 11              | 40                 |
| F         | 06    | 12-6            | 12-6            | 66           | *         | 47    | 11-8-4          | 11-8-4          | 54                 |
| G         | 07    | 12-7            | 12-7            | 67           | /         | 50    | 0-1             | 0-1             | 21                 |
| H         | 10    | 12-8            | 12-8            | 70           | (         | 51    | 0-8-4           | 12-8-5          | 34                 |
| I         | 11    | 12-9            | 12-9            | 71           | )         | 52    | 12-8-4          | 11-8-5          | 74                 |
| J         | 12    | 11-1            | 11-1            | 41           | \$        | 53    | 11-8-3          | 11-8-3          | 53                 |
| K         | 13    | 11-2            | 11-2            | 42           | =         | 54    | 8-3             | 8-6             | 13                 |
| L         | 14    | 11-3            | 11-3            | 43           | Blank     | 55    | Space           | Space           | 20                 |
| M         | 15    | 11-4            | 11-4            | 44           | .         | 56    | 0-8-3           | 0-8-3           | 33                 |
| N         | 16    | 11-5            | 11-5            | 45           | .         | 57    | 12-8-3          | 12-8-3          | 73                 |
| O         | 17    | 11-6            | 11-6            | 46           | ≡         | 60    | 0-8-6           | 8-3             | 36                 |
| P         | 20    | 11-7            | 11-7            | 47           | [         | 61    | 8-7             | 8-5             | 17                 |
| Q         | 21    | 11-8            | 11-8            | 50           | ]         | 62    | 0-8-2           | 12-8-7          | 32                 |
| R         | 22    | 11-9            | 11-9            | 51           | :         | 63    | 8-2             | 8-2             | 00 <sup>†</sup>    |
| S         | 23    | 0-2             | 0-2             | 22           | ≠         | 64    | 8-4             | 8-7             | 14 <sup>††</sup>   |
| T         | 24    | 0-3             | 0-3             | 23           | →         | 65    | 0-8-5           | 0-8-5           | 35                 |
| U         | 25    | 0-4             | 0-4             | 24           | √         | 66    | 11-0            | 11-0            | 52 <sup>†††</sup>  |
| V         | 26    | 0-5             | 0-5             | 25           | ^         | 67    | 0-8-7           | 12              | 37                 |
| W         | 27    | 0-6             | 0-6             | 26           | ↑         | 70    | 11-8-5          | 8-4             | 55                 |
| X         | 30    | 0-7             | 0-7             | 27           | ↓         | 71    | 11-8-6          | 0-8-7           | 56                 |
| Y         | 31    | 0-8             | 0-8             | 30           | <         | 72    | 12-0            | 12-0            | 72 <sup>††††</sup> |
| Z         | 32    | 0-9             | 0-9             | 31           | >         | 73    | 11-8-7          | 0-8-6           | 57                 |
| 0         | 33    | 0               | 0               | 12           | ≤         | 74    | 8-5             | 12-8-4          | 15                 |
| 1         | 34    | 1               | 1               | 01           | ≥         | 75    | 12-8-5          | 0-8-2           | 75                 |
| 2         | 35    | 2               | 2               | 02           | ┌         | 76    | 12-8-6          | 11-8-7          | 76                 |
| 3         | 36    | 3               | 3               | 03           | ;         | 77    | 12-8-7          | 11-8-6          | 77                 |
| 4         | 37    | 4               | 4               | 04           | EOL       | 0000  |                 |                 | 1632               |
| 5         | 40    | 5               | 5               | 05           |           |       |                 |                 |                    |
| 6         | 41    | 6               | 6               | 06           | Blank     | 55    | 6-8             | 0-8-4           | 16 <sup>††††</sup> |

Display code 00g is not associated with any card punch and cannot be represented on magnetic tape. Instead, it is converted to BCD 12. On input it is translated to display code 33.

<sup>†</sup>Written as 12 on magnetic tape

<sup>††</sup>In JOVIAL, the single prime (') is represented by ≠

<sup>†††</sup>11-0 and 11-8-2 are equivalent

<sup>††††</sup>12-0 and 12-8-2 are equivalent

<sup>†††††</sup>A 6-8 and 0-8-4 punch is converted to a display code 55 with no diagnostic given.

---

The JOVIAL library consists of pre-compiled FORTRAN and JOVIAL functions and procedures existing in the system library in relocatable binary format. The portion of the FORTRAN Extended Library which is available to JOVIAL provides a full set of computational and exponential routines and object time compatibility between JOVIAL object routines and FORTRAN Extended. Library subprograms may be shared by JOVIAL object programs, object programs from other compilers, or may be peculiar to JOVIAL object programs. This minimizes the size of the system library. The size of object program loads can be minimized by loading only one set of library routines or when object modules from more than one compiler are intermixed.

Only those programs described within the library table in the COMPOOL resolver may be called from JOVIAL. All other subprograms must either be compiled with the program calling it, or described in a COMPOOL. All library programs (JOVIAL or non-JOVIAL) use the non-JOVIAL calling sequence described in Section 8 and Appendix E.

The user may override the subprogram names in the library table by declaring a subprogram with the same name in the program being compiled or in a COMPOOL used by a program being compiled.

If a source library is required, the subprograms and programs may be placed on an UPDATE program library. The source library routines are common decks which are inserted in the program being compiled by means of the \*CALL Card. This procedure is described in detail in the UPDATE Section of the SCOPE Reference Manual.

**CAUTION**

The user should be aware that FORTRAN literal parameters are left-justified within a word and JOVIAL partial word literals are right justified. Literal parameters input to FORTRAN routines should use the format, 10H(            ), to force left justification (refer to Appendix E).

### FORTRAN LIBRARY FUNCTIONS

Table G-1 lists the FORTRAN library functions which may be called by JOVIAL programs.

TABLE G-1. FORTRAN LIBRARY FUNCTIONS

| Function and Number of Arguments     | Definition                                                                                                                                                          | Example                                                                        | Symbolic Name                  | Type of                                    |                                            |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|--------------------------------|--------------------------------------------|--------------------------------------------|
|                                      |                                                                                                                                                                     |                                                                                |                                | Argument                                   | Function                                   |
| Absolute value (1)                   | $ a $                                                                                                                                                               | YY=ABS(X)<br>JJ=IABS(I)                                                        | ABS<br>IABS                    | Floating<br>Integer                        | Floating<br>Integer                        |
| Truncation (1)                       | $\text{trunc}(a) = [a]$ if $a \geq 0$ $-[-a]$ if $a < 0$ where the function represented by $[a]$ is defined to be the integer $i$ that satisfies $i \leq a < i + 1$ | YY=AINT(X)<br>II=INT(X)                                                        | AINT<br>INT                    | Floating<br>Floating                       | Floating<br>Integer                        |
| Modulo                               | MOD or AMOD ( $a_1, a_2$ ) is defined to be $a_1 - \text{trunc}(a_1/a_2) * a_2$                                                                                     | BB=AMOD(A1, A2)<br>JJ=MOD(I1, I2)                                              | AMOD<br>MOD                    | Floating<br>Integer                        | Floating<br>Integer                        |
| Choosing largest value ( $\geq 2$ )  | Max ( $a_1, a_2, \dots$ )                                                                                                                                           | XX=AMAX0(I, J, K)<br>AA=AMAX1(X, Y, Z)<br>LL=MAX0(I, J, K, N)<br>II=MAX1(A, B) | AMAX0<br>AMAX1<br>MAX0<br>MAX1 | Integer<br>Floating<br>Integer<br>Floating | Floating<br>Floating<br>Integer<br>Integer |
| Choosing smallest value ( $\geq 2$ ) | Min ( $a_1, a_2, \dots$ )                                                                                                                                           | YY=AMIN0(I, J)<br>ZZ=AMIN1(X, Y)<br>LL=MIN0(I, J, K)<br>JJ=MIN1(X, Y)          | AMIN0<br>AMIN1<br>MIN0<br>MIN1 | Integer<br>Floating<br>Integer<br>Floating | Floating<br>Floating<br>Integer<br>Integer |
| Float (1)                            | Conversion from integer to floating                                                                                                                                 | X=FLOAT(I)                                                                     | FLOAT                          | Integer                                    | Floating                                   |
| Fix (1)                              | Conversion from floating to integer                                                                                                                                 | IY=IFIX(Y)                                                                     | IFIX                           | Floating                                   | Integer                                    |
| Transfer of sign (2)                 | Sign of $a_2$ times $ a_1 $                                                                                                                                         | ZZ=SIGN(X, Y)<br>JJ=ISIGN(I1, I2)                                              | SIGN<br>ISIGN                  | Floating<br>Integer                        | Floating<br>Integer                        |
| Positive difference (2)              | $a_1 - \text{Min}(a_1, a_2)$                                                                                                                                        | ZZ=DIM(X, Y)<br>JJ=IDIM(I1, I2)                                                | DIM<br>IDIM                    | Floating<br>Integer                        | Floating<br>Integer                        |
| Shift (2)                            | Shift $a_1$ by $a_2$ bit positions: left circular if $a_2$ is positive; right with sign extension if $a_2$ is negative                                              | BB=SHIFT(A, I)                                                                 | SHIFT                          | $a_1$ : Single word<br>$a_2$ : Integer     | Octal                                      |
| Logical product (2)                  | $a_1 \wedge a_2$                                                                                                                                                    | CC=AND(A1, A2)                                                                 | AND                            | Single word                                | Octal                                      |
| Logical sum (2)                      | $a_1 \vee a_2$                                                                                                                                                      | DD=OR(A1, A2)                                                                  | OR                             | Single word                                | Octal                                      |
| Complement (1)                       | $a$                                                                                                                                                                 | BB=COMPL(A)                                                                    | COMPL                          | Single word                                | Octal                                      |
| Exponential (1)                      | $e^a$                                                                                                                                                               | ZZ=EXP(Y)                                                                      | EXP                            | Floating                                   | Floating                                   |
| Natural logarithm (1)                | $\log_e(a)$                                                                                                                                                         | ZZ=ALOG(Y)                                                                     | ALOG                           | Floating                                   | Floating                                   |
| Common logarithm (1)                 | $\log_{10}(a)$                                                                                                                                                      | BB=ALOG10(A)                                                                   | ALOG10                         | Floating                                   | Floating                                   |

(Continued)

TABLE G-1. FORTRAN LIBRARY FUNCTIONS (Cont'd)

| Functions and Number of Arguments                        | Definition                                                                            | Example          | Symbolic Name | Type of        |           |
|----------------------------------------------------------|---------------------------------------------------------------------------------------|------------------|---------------|----------------|-----------|
|                                                          |                                                                                       |                  |               | Argument       | Function  |
| Trigonometric sine (1)                                   | sin (a)                                                                               | YY=SIN(X)        | SIN           | Floating       | Floating  |
| Trigonometric cosine (1)                                 | cos (a)                                                                               | XX=COS(Y)        | COS           | Floating       | Floating  |
| Hyperbolic tangent (1)                                   | tanh (a)                                                                              | BB=TANH(A)       | TANH          | Floating       | Floating  |
| Square root (1)                                          | $(a)^{1/2}$                                                                           | YY=SQRT(X)       | SQRT          | Floating       | Floating  |
| Arctangent (1)                                           | arctan (a)                                                                            | YY=ATAN(X)       | ATAN          | Floating       | Floating  |
| Arctangent (2)                                           | arctan ( $a_1/a_2$ )                                                                  | BB=ATAN2(A1, A2) | ATAN2         | Floating       | Floating  |
| Arccosine (1)                                            | arccos (a)                                                                            | XX=ACOS(Y)       | ACOS          | Floating       | Floating  |
| Arcsine (1)                                              | arcsin (a)                                                                            | XX=ASIN(Y)       | ASIN          | Floating       | Floating  |
| Trigonometric tangent (1)                                | tan (a)                                                                               | YY=TAN(X)        | TAN           | Floating       | Floating  |
| Random number generator (1)                              | ranf (a) returns values uniformly distributed over the range [0, 1)                   | XX=RANF(DUM)     | RANF          | Dummy          | Floating  |
| Address of argument a (1)                                | loc (a)                                                                               | PP=LOCF(X)       | LOCF          | Symbolic       | Integer   |
| I/O status on buffer unit (1)                            | = -1 unit ready; no error<br>= 0 EOF on last operation<br>= +1 parity error           | IO=UNIT(6)       | UNIT          | Integer        | Floating  |
| I/O status on non-buffer unit (1)                        | = 0 no EOF in previous read                                                           | IFL=EOF(4)       | EOF           | Integer        | Floating  |
| Length (1)                                               | Number of central memory words read on the previous I/O request for a particular file | LL=LENGTH(J)     | LENGTH        | Integer        | Integer   |
| Variable characteristic (1)                              | -1 = indefinite<br>+1 = out of range<br>0 = normal                                    | LEN=LEGVAR(V)    | LEGVAR        | Floating       | Integer   |
| Parity status on non-buffer unit (1)                     | 0 = no parity error on previous read                                                  | IP=IOCHEC(5)     | IOCHEC        | Integer        | Integer   |
| Date as returned by SCOPE (1)                            | date(a)                                                                               | WHEN=DATE(D)     | DATE          | Value Returned | Hollerith |
| Current reading of system clock as returned by SCOPE (1) | time(a)                                                                               | CLTIM=TIME(A)    | TIME          | Variable       | Hollerith |
| Time in seconds (1)                                      | second (a) (accumulated CP time)                                                      | CLTM=SECOND(A)   | SECOND        | Floating       | Floating  |

## FORTRAN LIBRARY SUBROUTINES

Table G-2 lists the FORTRAN library subroutines which may be called by JOVIAL programs.

TABLE G-2. FORTRAN LIBRARY SUBROUTINES

| Subroutine and Number of Arguments                          | Definition                                                                                                            | Example                         | Symbolic Name | Type of Argument                                  |
|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|---------------------------------|---------------|---------------------------------------------------|
| Set Sense Light (1)                                         | $1 \leq i \leq 6$ turn sense light <i>i</i> on. $i = 0$ turn off all sense lights.                                    | SLITE(I)                        | SLITE         | Integer                                           |
| Test Sense Light (2)                                        | If sense light <i>i</i> is on $j = 1$ . If off $j = 2$ always turn sense light <i>i</i> off.                          | SLITET(I, J)                    | SLITET        | Integer                                           |
| Test Sense Switch (2)                                       | If sense switch <i>i</i> is down $j = 1$ . If sense switch <i>i</i> is up $j = 2$ .                                   | SSWTCH(I, J)                    | SSWTCH        | Integer                                           |
| Terminate (0)                                               | Terminate program execution and return control to the monitor                                                         | EXIT                            | EXIT          |                                                   |
| Console Comment (1)                                         | Place a message of up to 80 characters on dayfile†                                                                    | REMARK(2HH I)                   | REMARK        | Hollerith                                         |
| Console Value (2)                                           | Display up to a 10 character message and value in the dayfile†                                                        | DISPLA(2HX=, 20.2)              | DISPLA        | $a_1$ =Hollerith<br>$a_2$ =floating<br>of integer |
| Obtain current generative value of RANF between 0 and 1 (1) | ranget (a)                                                                                                            | RANGET(X)                       | RANGET        | Symbolic                                          |
| Initialize generative value of RANF (1)                     | ranset (a), the generic value is set to the nearest odd number $\geq a$                                               | RANSET(X)                       | RANSET        | Floating                                          |
| Dump memory (3-60)                                          | dump(a,b,f)<br>dump A to B according to f                                                                             | DUMP(A, B, 1)<br>PDUMP(X, Y, 0) | DUMP<br>PDUMP | Logical<br>Integer<br>Floating                    |
| Input Checking (2)                                          | ERRSET (a,b), set maximum number of errors, b, allowed in input data before fatal termination. Error count kept in a. | ERRSET(A, B)                    | ERRSET        | Symbolic<br>Integer                               |

† Characters with a display code value above  $57_8$  are not allowed. The message must be terminated with binary zeros if less than 80 characters.



## JOVIAL LIBRARY PROCEDURES

The JOVIAL library procedures consist of the following routines:

- DECODE
- ENCODE
- HOLSTC
- OVRLOD
- PAUSE (see STOP in Section 4)
- REMQUO
- SEGLOD
- STCHOL

### DECODE

DECODE is a procedure designed to decode a Hollerith variable into subfields, while at the same time perform format conversion.<sup>†</sup> The format is:

```
DECODE(NO, FMT, VBL, FLD1, FLD2, ..., FLDn) $
```

where

- NO = number of bytes to be converted as specified in FMT
- FMT = specifies the conversion format
- VBL = defines the variable to be decoded
- FLD<sub>1</sub> } = identifies the fields for storing the decoded output
- FLD<sub>n</sub> }

Example:

- ITEM GAMMA 40H(HEADER 121HEAD,  
0131HEADER 122HEAD 0231) \$  
ITEM FMT2 8H(A10, A8) \$  
ITEM RR H 10 \$  
ITEM SS H 10 \$  
ITEM TT H 10 \$  
ITEM UU H 10 \$  
DECODE(18, FMT2, GAMMA, RR, SS,  
TT, UU) \$
- In this example DECODE is called to decode 18 bytes as specified in the FMT2 format, and continues decoding until the entire field list has been processed. After execution, the fields are set to the following values:
- RR = HEADER 121
  - SS = HEAD 01
  - TT = HEADER 122
  - UU = HEAD 02

<sup>†</sup> DECODE calls the FORTRAN conversion routine.

## ENCODE

ENCODE is a procedure designed to encode variables into a Hollerith variable, while at the same time perform format conversion. † The format is:

```
ENCODE(NO, FMT, VBL, FLD1, FLD2, ..., FLDn) $
```

where

NO = number of bytes to be converted as specified in FMT.  
FMT = specifies the conversion format  
VBL = defines the Hollerith variable to be encoded into  
FLD<sub>1</sub> } = identifies the variables to be encoded into one Hollerith variable.  
FLD<sub>n</sub> }

Example:

```
• ITEM FMT 15H(A10, A5/A10, A6) $
 ITEM ALPHA H 40 $
 ARRAY AA 2 H 10 $
 BEGIN 10H(ABCDEFGHIJ)
 10H(KLMNO) END
 ARRAY BB 2 H 10 $
 BEGIN 10H(PQRSTUVWXYZ)
 10H(Z12345) END
 ENCODE(20, FMT, ALPHA, AA(0),
 AA(1), BB(0), BB(1)) $
```

In this example ENCODE is called to encode the specified fields of arrays AA and BB into literal ALPHA. Twenty bytes are converted at a time as specified in the FMT format. The contents of the arrays will remain unchanged.

After execution, the contents of ALPHA is set to the following value:

```
ALPHA = ABCDEFGHIJKLMNOΔΔΔΔ PQRSTUVWXYZ12345ΔΔΔΔ
```

## HOLSTC

The HOLSTC routine converts literal characters from Hollerith to transmission code. The format is:

```
HOLSTC(LIT, WDS)$
```

---

† ENCODE calls the FORTRAN conversion routines.



where

- fn = variable name of a location which contains the name of the file (left-justified display code) that contains the overlay
- I = primary level of overlay
- J = secondary level of overlay
- p = recall parameter. If p equals 6HRECALL, the overlay is not reloaded if it is already in memory
- l = load parameter. Used to determine which value of the fn will be used. l may be any value. If l is present and non-zero, the overlay designated by fn will be loaded from the system library; otherwise, it will be loaded from the file designated by fn.

Numbers used in the OVERLAY card are octal; thus, to call OVERLAY(GARY, 1, 11) the statement OVRLOD (O(0701223100000000000), 1, 9) \$ must be used.

Prior to execution of this call which causes loading and execution of the overlay, the overlay must have been made absolute and written on file fn. When an END statement in the main program of an overlay is encountered, control returns to the statement following the call to OVRLOD which initialized execution of the overlay in question. (See Appendix M for an overlay example.)

#### REMQUO

The REMQUO procedure yields the quotient and the remainder in a division of two integers. The format is:

REMQUO(NUM, DEN=QUO, REM) \$

where

|                   |   |                                                     |
|-------------------|---|-----------------------------------------------------|
| NUM = numerator   | } | all input parameters are full word signed integers. |
| DEN = denominator |   |                                                     |
| QUO = quotient    |   |                                                     |
| REM = remainder   |   |                                                     |

This form is equivalent to the JOVIAL statements below:

QUO = NUM/DEN \$  
REM = NUM-QUO \* DEN \$

**Example:**

- REMQUO(IN1, IN2 = OUT1, OUT2) \$      With integers IN1 set to 35 and IN2 set to 15, integers OUT1 is set to 2 and OUT2 set to 5 after execution.

**SEGLOD**

The SEGLOD routine serves as a JOVIAL entry point to the standard SEGMENT routine in the FORTRAN library to load the specified segments. The format is:

SEGLOD(fn, e, a, lib, m)

where

- fn = variable name of location which contains the file name (left-justified display code) from which the segment load takes place.
- e = level of the segment load.
- a = variable name of array containing a list of SEGMENTS, SECTIONS and/or SUBPROGRAMS to be loaded with this call. In this list, the name must be in left-justified display code, and the list must be terminated by a zero entry. An initial list entry of zero signals a segment load of all subprograms remaining on the file fn.
- lib = if zero or blank, unsatisfied externals are to be satisfied, if possible, from the system library.
- m = if zero or blank, a map of the segment load is not produced. lib and m need not be specified.

Once the named subprograms are loaded control returns to the statement following the call to SEGMENT. The programmer is free to call on the loaded subprograms as desired.

**STCHOL**

The STCHOL routine converts literal characters from transmission code to Hollerith. The format is:

STCHOL(LIT, WDS) \$

where

- LIT = literal name of characters to be converted from transmission code to Hollerith.
- WDS = an integer number that specifies the number of words to be converted.

Full word conversion is performed starting at the location of LIT. Packed table items and parallel table items cannot be converted successfully.

Example:

- ITEM HOL H 80 \$  
ITEM STC T 80 \$  
OVERLAY HOL = STC \$  
FOR A=0, 1, 63 \$  
BIT (\$A\*6, 6\$) (STC) = A \$
- Eighty byte Hollerith and transmission code literals which overlay each other.
- Sets literals to all possible byte patterns; byte zero, decimal value zero; byte one, decimal value one;... byte 63, decimal value 63.
- MONITOR STC \$  
STC = STC \$  
STCHOL (STC, 8) \$  
HOL = HOL \$
- Monitor as transmission code variable
- Calls STCHOL to convert to Hollerith code
- Monitor as Hollerith variable.

After execution, STC and HOL will have the following monitored output values:

```
STC = ↓ ↓ ↓ ↓ ↓ ABCDEFGHIJKLMNOPQRSTUVWXYZ-+↓ = ↓ ↓ ↓ ↓ ↓ $*(↓ ↓ ↓ ↓ ↓ ,
 0123456789#↓ /.
```

```
HOL = Δ Δ Δ Δ Δ ABCDEFGHIJKLMNOPQRSTUVWXYZ-+Δ = Δ Δ $*(Δ Δ Δ Δ Δ ,
 0123456789#Δ /.
```

#### NOTE

The bytes containing bit patterns with no transmission code representation (indicated by the down arrows) are converted to Hollerith blanks.

## JOVIAL LIBRARY FUNCTION

The function REM is a predefined function in the JOVIAL library.

### REM

REM is a function that yields only the remainder in a division of two integers. The format is:

```
REM(NUM, DEN) $
```

where,

NUM = numerator

DEN = denominator

REM = remainder

This form is equivalent to the JOVIAL statements below:

REM = NUM/DEN \$

REM = NUM - REM \* DEN \$

## SAMPLE LISTINGS

H I

This appendix contains two sample listings that consist of a COMPOOL creation run, a main program, and two stand-alone PROCs. The listings were too large to show all portions for each listing. The cross-reference, storage map, partial load map, and part of the program output are shown for the first listing. The object code, a full load map, and the program output are shown for the second listing.

The first program listing computes the sine and cosine values for each degree from -360 to +360. The dayfile printout for this listing occurred in the following sequence:

| <u>Dayfile Sequence</u>            | <u>Description</u>                                                                                                                                  |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| DEMO, CM70000, T77.                | Identifies job.                                                                                                                                     |
| JOVIAL(XR, A=CMPL, N=CMPL, B=TRIG) | Performs the COMPOOL compilation.                                                                                                                   |
| JOVIAL(XR, C=CMPL, B=TRIG, N=TRIG) | Compiles the main program and the two stand-alone PROCs. The multiple compilation feature was used.                                                 |
| MAP(PART)                          | Maps only the routines, not the entry points.                                                                                                       |
| TRIG.                              | Executes the program.                                                                                                                               |
| END TRIG                           | Notification of program termination. This is not produced by a control card, but is generated by code placed in the object program by the compiler. |

The second program listing tests a binary-to-decimal conversion procedure subprogram, and a decimal-to-binary conversion function subprogram. The code and full load map are shown to demonstrate the way in which JOVIAL subprogram parameters are passed (see Appendix E). The dayfile is shown later in this appendix.



This COMPOOL compilation was run using the following control card:

JOVIAL(XR A=CMPL, N=CMPL, B=TRIG)

where

- X storage map
- R cross reference
- A=CMPL performs a COMPOOL compilation to file CMPL
- N=CMPL the name of the presets
- B=TRIG the file on which the presets are placed

```

0001. START*
0002. ** * * * * *
0003. * * * * *
0004. * * * * *
0005. * * * * *
0006. * * * * *
0007. * * * * *
0008. * * * * *
0009. * * * * *
0010. * * * * *
0011. * * * * *
0012. * * * * *
0013. * * * * *
0014. * * * * *
0015. * * * * *
0016. * * * * *
0017. * * * * *
0018. * * * * *
0019. * * * * *
0020. * * * * *
0021. * * * * *
0022. * * * * *
0023. * * * * *
0024. * * * * *
0025. * * * * *
0026. * * * * *
0027. * * * * *
0028. * * * * *
0029. * * * * *
0030. * * * * *
0031. * * * * *
0032. * * * * *
0033. * * * * *
0034. * * * * *
0035. * * * * *
0036. * * * * *
0037. * * * * *
0038. * * * * *
0039. * * * * *
0040. * * * * *
0041. * * * * *
0042. * * * * *
0043. * * * * *
0044. * * * * *
0045. * * * * *
0046. * * * * *

DEFINE WSIZE 24000
DEFINE SIZE01 24000
DEFINE SIZE02 24000
DEFINE SIZE03 24000
DEFINE SIZE04 24000
DEFINE SIZE05 24000
DEFINE SIZE06 24000
DEFINE SIZE07 24000
DEFINE SIZE08 24000
DEFINE SIZE09 24000
DEFINE SIZE10 24000
DEFINE SIZE11 24000
DEFINE SIZE12 24000
DEFINE SIZE13 24000
DEFINE SIZE14 24000
DEFINE SIZE15 24000
DEFINE SIZE16 24000
DEFINE SIZE17 24000
DEFINE SIZE18 24000
DEFINE SIZE19 24000
DEFINE SIZE20 24000
DEFINE SIZE21 24000
DEFINE SIZE22 24000
DEFINE SIZE23 24000
DEFINE SIZE24 24000
DEFINE SIZE25 24000
DEFINE SIZE26 24000
DEFINE SIZE27 24000
DEFINE SIZE28 24000
DEFINE SIZE29 24000
DEFINE SIZE30 24000
DEFINE SIZE31 24000
DEFINE SIZE32 24000
DEFINE SIZE33 24000
DEFINE SIZE34 24000
DEFINE SIZE35 24000
DEFINE SIZE36 24000
DEFINE SIZE37 24000
DEFINE SIZE38 24000
DEFINE SIZE39 24000
DEFINE SIZE40 24000
DEFINE SIZE41 24000
DEFINE SIZE42 24000
DEFINE SIZE43 24000
DEFINE SIZE44 24000
DEFINE SIZE45 24000
DEFINE SIZE46 24000

PRG SIN(NEG)
PRG ITEM DEG A WSIZE S SIZE01 FND
PRG COS(NEG)
PRG ITEM COS A WSIZE S SIZE01 FND
COMMON MAIN
DEF
ITEM TCHAB T 64 P
ITEM TLIST T 120
ITEM INTEG A WSIZE S
ITEM FRACT A WSIZE S SIZE01
ITEM DEG A WSIZE S SIZE01
ITEM PI A WSIZE S SIZE02
ITEM TEN A 4 0 P 10
FND
COMMON SURR
DEF
ITEM TEMP A WSIZE S SIZE01
ITEM QUAN A 2 U
ITEM T1 A WSIZE S SIZE01
ITEM T2 A WSIZE S SIZE01
ITEM T3 A WSIZE S SIZE01
ITEM T4 A WSIZE S SIZE01
ITEM T5 A WSIZE S SIZE01
ITEM T6 A WSIZE S SIZE01
ITEM T7 A WSIZE S SIZE01
ITEM T8 A WSIZE S SIZE01
ITEM T9 A WSIZE S SIZE01
ITEM T10 A WSIZE S SIZE01
FND

```

Common Block Main Program

Common Block Subroutine

The object code was not printed in order to reduce the size of the example listing.

CMPL

Program Name

GDC 6000 SERIES JOVIAL COMPILER VERSION 1.0 01/1 \* DIAGNOSTICS \*

NUMBER LINE MESSAGE

\*\* NO DIAGNOSTIC MESSAGES\*\*

| NAME  | TYPE   | H LOC      | FBIT NUM | NAME  | TYPE   | H LOC      | FBIT NUM | NAME | TYPE | H LOC      | FBIT NUM |
|-------|--------|------------|----------|-------|--------|------------|----------|------|------|------------|----------|
| COS   | DATA   | A 000000X  | 12 48    | COS   | FUNC   | A 000000   | 12 48    | DEG  | DATA | A 000025C3 | 12 48    |
| DEG   | DATA   | A 000000X  | 12 48    | DEG   | DATA   | A 000000X  | 12 48    | PI   | DATA | A 000026C3 | 12 48    |
| INTEG | DATA   | I 000023C3 | 12 48    | MAIN  | COMMON |            |          | SIN  | FUNC | A 000000   | 12 48    |
| QUAD  | DATA   | I 000001C4 | 56 2     | SIN   | DATA   | A 000000X  | 12 48    | TEMP | DATA | A 000008C4 | 12 48    |
| SUBR  | COMMON |            |          | TCHAR | DATA   | Y 000000C3 | 0 64     | T1   | DATA | A 000002C4 | 12 48    |
| TEN   | DATA   | I 000027C3 | 56 4     | TLIST | DATA   | A 000007C3 | 0 120    | T2   | DATA | A 000006C4 | 12 48    |
| T10   | DATA   | A 000013C4 | 12 48    | T2    | DATA   | A 00003C4  | 12 48    | T3   | DATA | A 000006C4 | 12 48    |
| T4    | DATA   | A 000005C4 | 12 48    | T5    | DATA   | A 000006C4 | 12 48    | T6   | DATA | A 000007C4 | 12 48    |
| T7    | DATA   | A 000010C4 | 12 48    | T8    | DATA   | A 000011C4 | 12 48    | T9   | DATA | A 000012C4 | 12 48    |

Number of Bits or Bytes (Bits, unless specified literal)

First Bit

Location within Common Block and Common Block Identifier

Mode for Data Items

Type of Item

Name of Item

In a COMPOOL assembly, the data map acts as the COMPOOL disassembly. It lists all of the items declared in COMPOOL and the addresses assigned. Addresses for procedures and their parameters are not exact as they will be relocated by the loader at execution time. However, for all data items in common, the address shown is the actual address relative to the common block followed by a C, indicating the variable is as a common block, followed by the loader relocation number for the common block.

| NAME   | TYPE   | M | DEF | SCOPE | SET/USED ( USED INDICATED BY * ) | CMPL | TIME | DATE |
|--------|--------|---|-----|-------|----------------------------------|------|------|------|
| BEG    | DEFINE |   | 1*  | CMPL  | 15*                              | 18*  | 21*  | 32*  |
| SIZE00 | DEFINE |   | 8   | CMPL  | 26*                              | 35*  | 36*  | 37*  |
| SIZE01 | DEFINE |   | 9   | CMPL  | 43*                              | 44*  |      |      |
| SIZE02 | DEFINE |   | 10  | CMPL  | 16*                              | 19*  |      |      |
| SIZE09 | DEFINE |   | 11  | CMPL  | 28*                              |      |      |      |
| WDSIZE | DEFINE |   | 7   | CMPL  | 15*                              | 18*  | 27*  | 33*  |
|        |        |   |     |       | 35*                              | 36*  | 37*  | 38*  |
|        |        |   |     |       | 44*                              |      |      |      |
|        |        |   |     |       |                                  |      | 26*  | 27*  |
|        |        |   |     |       |                                  |      | 40*  | 41*  |
|        |        |   |     |       |                                  |      | 42*  | 43*  |

Use of Define

Scope of Item

Line on which Defined

Type

Defined Name

Since a COMPOOL may not contain executable statements, the cross reference will show only DEFINE statements and where they are used.

The batch compilation of the main program and subprograms was run using the following control card:  
 JOVIAL(XR,C=CMPL,B=TRIG,N=TRIG)  
 where  
 X storage map  
 R cross reference  
 C=CMPL performs the compilation using the COMPOOL in file CMPL  
 B=TRIG file location of the object code  
 N=TRIG program name

```

0001. START **
0002. **
0003. ** DRIVER FOR SIN/COS GENERATION *
0004. **
0005. **
0006. **
0007. DEFINE WDSIZE 24**
0008. DEFINE SIZE00 243**
0009. DEFINE SIZE01 242**
0010. DEFINE SIZE02 241**
0011. DEFINE SIZE03 240**
0012. DEFINE CHSIZE 26**
0013. DEFINE HBLNK 21H()**
0014. DEFINE TBLNK 21T()**
0015. DEFINE REG 2REGIN**
0016. FILE LIST H 999 V 130 V(READY) OUTPUT *
0017. ITEM SINT A WDSIZE S SIZE00 *
0018. OPEN OUTPUT LIST *
0019. TLIST=TBLNK*
0020. PI=3.141592654
0021. BYTE(47,38)(TLIST)=3T(PI)=3*
0022. INTEG=PI* FRACT=PI*
0023. RTOD(INTEG,FRACT,50,16)*
0024. STCHOL(TLIST,12) * OUTPUT LIST TLIST *
0025. TLIST=TBLNK*
0026. OUTPUT LIST HBLNK*
0027. OUTPUT LIST HBLNK*
0028. BYTE(1, 7)(TLIST)=7T(DEGREES) *
0029. BYTE(65, 7)(TLIST)=7T(DEGREES) *
0030. BYTE(85,13)(TLIST)=13T(SINE(DEGREES)) *
0031. BYTE(73,13)(TLIST)=13T(SINE(DEGREES)) *
0032. BYTE(90,15)(TLIST)=15T(COSINE(DEGREES)) *
0033. BYTE(90,15)(TLIST)=15T(COSINE(DEGREES)) *
0034. STCHOL(TLIST,12) * OUTPUT LIST TLIST *
0035. TLIST=TBLNK*
0036. OUTPUT LIST HBLNK*
0037. FOR I=0,1,360*
0038. REG DEG=I*
0039. INTEG=DEG* FRACT=DEG*
0040. RTOD(INTEG,FRACT, 1, 7)*
0041. INTEG=-DEG* FRACT=-DEG*
0042. RTOD(INTEG,FRACT,61, 7)*
0043. INTEG=SIN(DEG)* FRACT=SIN(DEG)*
0044. SINT=FRAC *
0045. RTOD(INTEG,FRACT,11,15)*
0046. INTEG=SIN(-DEG)* FRACT=SIN(-DEG)*
0047. RTOD(INTEG,FRACT, 7,15)*
0048. INTEG=COS(DEG)* FRACT=COS(DEG)*
0049. RTOD(INTEG,FRACT,30,15)*
0050. IF (1.0*SINT*SINT+1.0*FRACT*FRACT-1.0) GR 0.00005*
0051. BYTE(51, 5) (TLIST)=5T(ERDOR)*
0052. INTEG=COS(-DEG)* FRACT=COS(-DEG)*
0053. RTOD(INTEG,FRACT, 90,15)*
0054. STCHOL(TLIST,12) * OUTPUT LIST TLIST *
0055. TLIST=TBLNK*
0056. END

```

Program Name

```

0057. OUTPUT LIST HBLNK$
0058. OUTPUT LIST HBLNK$
0059. OUTPUT LIST HBLNK$
0060. FOR I=0,8,11% BYTE($I,0$)(TLIST)=8T(*+DONE**)$
0061. SYCHOL(TLIST,12) $ OUTPUT LIST TLIST $
0062. TLIST=TBLNK$
0063. SHUT OUTPUT LISTS STOPS
0064.
0065. PROC BTOD(INTEG,FRACT,FBYTE,NBYTE) $
0066.
0067. ITEM INTEG A WDSIZE S$
0068. ITEM FBYTE A 15 US$
0069. ITEM NBYTE A 15 US$
0070. ITEM Y1 A WDSIZE S$
0071. ITEM Y2 A WDSIZE S$
0072. ITEM Y3 A WDSIZE S$
0073. ITEM TTEMP T 1$
0074. ITEM POINT A 15 US$
0075. ITEM LIMIT A 15 US$
0076.
0077. IF NBYTE EQ 0% RETURN$
0078. TTEMP=IT(1) $ T2=INTEG$ T3=FRACT$
0079. IF INTEG LS 0$
0080.
0081. BEG TTEMP=IT(-1) $ T2=-INTEG$ END
0082. IF FRACT LS 0$
0083.
0084. BEG TTEMP=IT(-1) $ T3=-FRACT$ END
0085.
0086. POINT=FBYTE+1$ T1=0$
0087. FOR I=9,1+9,9,9999999999999999$
0088. BEG POINT=POINT+1$ T1=T1+1$
0089. IF T2 LQ I$ GOTO L1$
0090. IF T1 GO 15$ GOTO L1$ END
0091. LIMIT=FBYTE+NBYTE-1$
0092. FOR I=POINT-1,-1,-1,FBYTE$
0093. BEG IF T2 EQ 0 AND I NO POINT-1$ GOTO L2$
0094. T1=T2$ Y2=T2/TENS$
0095. BYTE($I)(TLIST)=BYTE($T1-(T2*TENS)+0$)(TCHAR) $ END
0096. L2. IF POINT GR LIMIT$ RETURN$
0097. BYTE($POINT$)(TLIST)=IT(.)$
0098. IF POINT+1 GR LIMIT$ RETURN$
0099. BIT($0,5$)(T3)=0$
0100. FOR I=POINT+1,1,LIMIT$
0101. BEG T2=T3*TENS T3=T3*TENS BIT($0,5$) T3=0$
0102. BYTE(I)(TLIST)=BYTE($T2+0$)(TCHAR) $ END
0103.
0104. END
0105. TERM $

```

The brackets are missing from the object of the BIT functional modifier to show the error diagnostics and error recoveries performed by the compiler. The correct statement is:  
 BIT(\$0,5\$)(T3) \$

NUMBER LINE MESSAGE

JOV502 (CMPL ) COMPOOL-DEFINED SYMBOLS

- COS
- DEG
- FRACT
- INTEG
- PI
- SIN
- TCHAR
- TEN
- TLIST

JOV503 (CMPL ) MODE-DEFINED SYMBOLS

- \*\* NONE \*\*
- JOV070 (0100.)M MISSING ) OR ( ASSUMED TO BE PRESENT.
- JOV067 (0100.)M ) OR \$) MISSING.

\*\*002 DIAGNOSTIC MESSAGES\*\*

These diagnostic messages are produced because of the error in the statement at line 100 on the previous page. The compiler was able to make corrections to the source statement which yielded a correct JOVIAL statement, hence the messages are only warnings and no call to the Run Time Error Monitor was inserted.

| NAME     | TYPE  | M LOC      | FBIT NUM | NAME     | TYPE   | M LOC      | FBIT NUM | NAME   | TYPE   | M LOC      | FBIT NUM |
|----------|-------|------------|----------|----------|--------|------------|----------|--------|--------|------------|----------|
| COS\$\$A | DATA  | A 000000X  | 12 48    | DEC      | DATA   | A 000025C3 | 12 48    | END.   | PROC   | X 000000   |          |
| FBYTE    | DATA  | I 000230   | 45 15    | I        | FORVAR | I 000240   |          | I      | FORVAR | I 000225   |          |
| INTEG    | DATA  | I 000023C3 | 12 48    | INTEG    | DATA   | I 000226   | 12 48    | JOVIO. | PROC   | X 000000   |          |
| JOVSN\$  | PROC  | X 000000   |          | LIMIT    | DATA   | I 000237   | 45 15    | LIST   | FILE   | 000000     |          |
| L1       | LABEL | I 000583   |          | L2       | LABEL  | 000533     |          | MAIN   | COMMON | I 000000   |          |
| NBYTE    | DATA  | I 000231   | 45 15    | PI       | DATA   | A 000026C3 | 12 48    | POINT  | DATA   | I 000236   | 45 15    |
| QENTRY.  | PROC  | X 000000   |          | SIN      | FUNC   | A 000000X  | 12 48    | SINT   | DATA   | A 000224   | 12 48    |
| SIN\$\$A | DATA  | A 000000X  | 12 48    | STCHOL\$ | PROC   | X 000000   |          | TCHAR  | DATA   | T 000000C3 | 0 64     |
| TEN      | DATA  | I 000027C3 | 56 4     | TTEMP    | DATA   | T 000235   | 54 1     | T1     | DATA   | I 000232   | 12 48    |

Number of Bits or Bytes (Bits, unless specified Literal)

First Bit

Location within Program or Common Block

Mode for Data Items

Type of Item

Name Of Item



| NAME     | TYPE   | M | DEF | SCOPE | SET/USED ( USED INDICATED BY * ) | TRIG | TIME |
|----------|--------|---|-----|-------|----------------------------------|------|------|
| BEG      | DEFINE |   | 15  | TRIG  | 38*                              | 80*  | 92*  |
| CHSIZE   | DEFINE |   | 12  | TRIG  | 65*                              | 86*  | 100* |
| DEC      | DATA   | A | C3  | TRIG  | 38                               | 41*  | 43*  |
|          |        |   |     |       | 48*                              | 52*  | 46*  |
|          |        |   |     |       | 63*                              | 84*  | 89*  |
| FBYTE    | DATA   | I | 68  | BTOD  | 26*                              | 27*  | 59*  |
| FBYTE    | FORPAR | I | 64  | BTOD  | 85                               | 91   | 92*  |
| HBLNK    | DEFINE | I | 13  | TRIG  | 37                               | 60   | 94*  |
| I        | FORVAR | I |     | BTOD  | 22                               | 39   | 101* |
| I        | FORVAR | I |     | TRIG  | 42*                              | 45*  | 52   |
| INTEG    | DATA   | I | C3  | TRIG  | 77*                              | 79*  | 80*  |
| INTEG    | DATA   | I | 66  | BTOD  | 89                               | 90*  | 99*  |
| INTEG    | FORPAR | I | 64  | BTOD  | 18                               | 24   | 36   |
| LIMIT    | DATA   | I | 75  | BTOD  | 59                               | 61   | 57   |
| LIST     | FILE   |   | 16  | TRIG  | 63                               |      | 58   |
| L1       | LABEL  |   | 89  | BTOD  | 76*                              | 89*  |      |
| L2       | LABEL  |   | 95  | BTOD  | 92*                              |      |      |
| NBYTE    | DATA   | I | 69  | BTOD  | 20                               | 22*  | 91*  |
| NBYTE    | FORPAR | I | 64  | BTOD  | 84                               | 86   | 92*  |
| PI       | DATA   | A | C3  | TRIG  | 97*                              | 99*  | 95*  |
| POINT    | DATA   | I | 74  | BTOD  | 43*                              | 46*  | 34   |
| SIN      | FUNC   | A |     | TRIG  | 17*                              | 67*  | 58   |
| SINT     | DATA   | A | 17  | TRIG  | 22*                              | 22*  | 98*  |
| SIZE00   | DEFINE | A | 8   | TRIG  | 43*                              | 46*  | 91*  |
| SIZE01   | DEFINE |   | 9   | TRIG  | 50*                              | 50*  | 92*  |
| SIZE02   | DEFINE |   | 10  | TRIG  | 67*                              | 72*  | 95*  |
| SIZE09   | DEFINE |   | 11  | TRIG  | 17*                              | 67*  | 96*  |
| SYMBOL\$ | PROC   | X |     |       |                                  |      |      |
| TBLNK    | DEFINE |   | 14  | TRIG  | 24*                              | 34*  | 61*  |
| TCHAR    | DATA   | T | C3  | BTOD  | 19*                              | 25*  | 35*  |
| TEN      | DATA   | I |     | TRIG  | 94*                              | 101* | 62*  |
| TTPMP    | DATA   | I | C3  | BTOD  | 93*                              | 94*  | 100* |
| T1       | DATA   | I | 73  | BTOD  | 77                               | 80   | 83*  |
| WDSIZE   | DATA   | I | 78  | BTOD  | 84                               | 86   | 88*  |
|          | DEFINE |   | 7   | TRIG  | 17*                              | 66*  | 71*  |
|          |        |   |     |       |                                  |      | 72*  |

Scope of Item  
 Line or common block where defined, in the case of a formal PROC parameter, the line defining the PROC.

Mode

Type

Name of Item

All COMPOOL defined items set or used within the program and all items defined within the program are listed. Dead items which are defined but not set or used are shown to enable the user to eliminate them.



NUMBER LINE MESSAGE

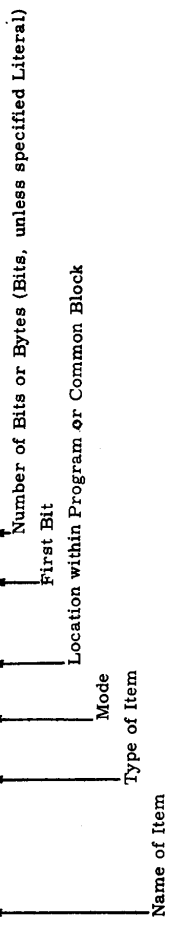
JOV502 (CMPL ) COMPOOL-DEFINED SYMBOLS

- PI
- QUAD
- TEMP
- T1
- T10
- T2
- T3
- T4
- T5
- T6
- T7
- T8
- T9

JOV503 (CMPL ) MODE-DEFINED SYMBOLS  
\*\* NONE \*\*

\*\* NO DIAGNOSTIC MESSAGES\*\*

| NAME | TYPE  | M LOC      | FBIT NUM | NAME | TYPE   | M LOC      | FBIT NUM | SIN  | TYPE   | M LOC      | FBIT NUM |
|------|-------|------------|----------|------|--------|------------|----------|------|--------|------------|----------|
| L1   | LABEL | 000069     |          | L2   | LABEL  | 000067     |          | L3   | LABEL  | 000074     |          |
| L4   | LABEL | 000172     |          | L5   | LABEL  | 000271     |          | MAIN | COMMON |            |          |
| PI   | DATA  | A 000026C3 | 12 48    | QUAD | DATA   | I 000001C4 | 58 2     | SIN  | DATA   | A 000000   | 12 48    |
| T1   | FUNC  | A 000052   | 12 48    | SUPR | COMMON |            |          | TEMP | DATA   | A 000000C4 | 12 48    |
| T6   | DATA  | A 00002C4  | 12 48    | T10  | DATA   | A 000013C4 | 12 48    | T5   | DATA   | A 000006C4 | 12 48    |
| T9   | DATA  | A 00007C4  | 12 48    | T7   | DATA   | A 000010C4 | 12 48    | T8   | DATA   | A 000011C4 | 12 48    |



| NAME   | TYPE   | M | DEF | SCOPE | SET/USED ( USED INDICATED BY * ) | SIN                  |
|--------|--------|---|-----|-------|----------------------------------|----------------------|
| L1     | LABEL  |   | 17  | SIN   | 18*                              |                      |
| L2     | LABEL  |   | 19  | SIN   | 17*                              |                      |
| L3     | LABEL  |   | 21  | SIN   | 19*                              |                      |
| L4     | LABEL  |   | 34  | SIN   | 21*                              |                      |
| L5     | LABEL  |   | 46  | SIN   | 33*                              |                      |
| P1     | DATA   | A | C3  | SIN   | 22*                              | 34*                  |
| QUAD   | DATA   | I | C4  | SIN   | 16                               | 10 18 20 18* 20* 20* |
|        |        |   |     |       | 46*                              | 47                   |
|        |        |   |     |       | 33                               | 45                   |
| SIN    | DATA   | A | 14  | SIN   |                                  |                      |
| SIN    | FUNC   | A | 1   | SIN   |                                  |                      |
| SIZE03 | DEFINE | A | 8   | SIN   | 14*                              |                      |
| SIZE01 | DEFINE |   | 9   | SIN   |                                  |                      |
| SIZE02 | DEFINE |   | 10  | SIN   |                                  |                      |
| SIZE09 | DEFINE |   | 11  | SIN   |                                  |                      |
| TEMP   | DATA   | A | C4  | SIN   | 15*                              |                      |
|        |        |   |     |       | 16                               | 17 17* 17* 19* 19*   |
| T1     | DATA   | A | C4  | SIN   | 21*                              | 21 21*               |
| T10    | DATA   | A | C4  | SIN   | 22                               | 22* 23* 35* 36*      |
|        |        |   |     |       | 23                               | 34* 35* 45*          |
|        |        |   |     |       | 29*                              | 44 25* 26*           |
|        |        |   |     |       | 40*                              | 31* 32* 37* 27*      |
|        |        |   |     |       | 27                               | 36* 28* 29*          |
| T5     | DATA   | A | C4  | SIN   | 40*                              | 43* 44* 45*          |
| T6     | DATA   | A | C4  | SIN   | 27                               | 39 20* 45*           |
| T7     | DATA   | A | C4  | SIN   | 28                               | 40 29* 38*           |
| T8     | DATA   | A | C4  | SIN   | 29                               | 41 30* 41* 45*       |
| T9     | DATA   | A | C4  | SIN   | 30                               | 33* 33* 42*          |
| WDSIZE | DEFINE | A | C4  | SIN   | 31                               | 31* 43* 45*          |
|        |        |   |     |       | 18*                              | 32* 33* 44*          |
|        |        |   |     |       | 15*                              | 33* 44*              |

Mode  
 Scope of Item  
 Line or common block where defined,  
 in the case of a formal PROC para-  
 meter, the line defining the PROC.

Type  
 Name of Item  
 Set/Used Information



NUMBER LINE MESSAGE

JOV502 (CMPL ) COMPOOL-DEFINED SYMBOLS

- PI
- QUAD
- TEMP
- T1
- T2
- T3
- T4
- T5
- T6
- T7
- T8
- T9

JOV503 (CMPL ) MODE-DEFINED SYMBOLS

- \*\* NONE \*\*

\*\* NO DIAGNOSTIC MESSAGES\*\*

| NAME | TYPE  | M LOC      | FBIT NUM | NAME | TYPE       | M LOC      | FBIT NUM | COS  | TYPE   | M LOC      | FBIT NUM |
|------|-------|------------|----------|------|------------|------------|----------|------|--------|------------|----------|
| L1   | LABEL | 000052     |          | L2   | LABEL      | 000071     |          | L3   | LABEL  | 000076     |          |
| L4   | LABEL | 000174     |          | L5   | LABEL      | 000273     |          | MAIN | COMMON |            |          |
| PI   | DATA  | A 000026C3 | 12 48    | QUAD | I 000001C4 | 58 2       |          | SURR | COMMON |            |          |
| TEMP | DATA  | A 000000C4 | 12 48    | T1   | A 000002C4 | 12 48      |          | T10  | DATA   | A 000013C4 | 12 48    |
| T5   | DATA  | A 000006C4 | 12 48    | T6   | DATA       | A 000007C4 | 12 48    | T7   | DATA   | A 000010C4 | 12 48    |
| T8   | DATA  | A 000011C4 | 12 48    | T9   | DATA       | A 000012C4 | 12 48    |      |        |            |          |

↑ Number of Bits or Bytes (Bits, unless specified Literal)  
 ↑ First Bit  
 ↑ Location within Program or Common Block  
 ↑ Mode  
 ↑ Type of Item  
 ↑ Name of Item



| NAME    | TYPE   | M | DEF | SCOPE | SET/USED | USED INDICATED BY * |
|---------|--------|---|-----|-------|----------|---------------------|
| L1      | LABEL  |   | 17  | COS   | 10*      |                     |
| L2      | LABEL  |   | 19  | COS   | 17*      |                     |
| L3      | LABEL  |   | 21  | COS   | 19*      |                     |
| L4      | LABEL  |   | 34  | COS   | 21*      |                     |
| L5      | LABEL  |   | 46  | COS   | 33*      |                     |
| PI      | DATA   | A | C3  | COS   | 22*      | 34*                 |
| QUAD    | DATA   | I | C4  | COS   | 16       | 18 18* 20* 20*      |
|         |        |   |     |       | 46*      | 46*                 |
| SIZE00  | DEFINE |   | 8   | COS   | 14*      |                     |
| SIZE01  | DEFINE |   | 9   | COS   | 15*      |                     |
| SIZE02  | DEFINE |   | 10  | COS   | 16       |                     |
| SIZE09  | DEFINE |   | 11  | COS   | 15*      |                     |
| TEMP    | DATA   | A | C4  | COS   | 16       | 17 17* 17* 19* 19*  |
| T1      | DATA   | A | C4  | COS   | 21*      | 21*                 |
| T10     | DATA   | A | C4  | COS   | 22       | 22* 34*             |
|         |        |   |     |       | 23       | 23* 24*             |
|         |        |   |     |       | 32       | 35* 35*             |
|         |        |   |     |       | 29       | 24* 25* 26*         |
|         |        |   |     |       | 40*      | 31* 32* 36*         |
|         |        |   |     |       | 27       | 33* 37*             |
|         |        |   |     |       | 39       | 42* 43* 45*         |
| T5      | DATA   | A | C4  | COS   | 27       | 40* 45*             |
| T6      | DATA   | A | C4  | COS   | 28       | 41* 45*             |
| T7      | DATA   | A | C4  | COS   | 29       | 42* 45*             |
| T8      | DATA   | A | C4  | COS   | 30       | 43* 45*             |
| T9      | DATA   | A | C4  | COS   | 31       | 43* 45*             |
| WDOSIZE | DEFINE | A | 7   | COS   | 14*      | 15*                 |

Set/Used Information

Scope of Item

Line or common block where defined, in the case of a formal PROC parameter, the line defining the PROC.

Mode

Type

Name of Item

CORE MAP 17.26.02. NORMAL CONTROL  
 FWA LOANEP 063771 FWA TABLES 061523  
 CHPL 000144  
 TRIG 000144  
 SIN 001164  
 COS 001467  
 GET9A 001770  
 SYSTEME 002007  
 RACKSOR 002772  
 REFINME 003317  
 JOVIMC 003637  
 HOLSTIC 004447  
 SIKR 004666  
 INPUTR 006233  
 ----UNSATISFIED EXTERNALS-----  
 REFERENCE  
 PI = 3.14159265499998

000100 004511 000000 000000  
 FWA LOAN--LWA LOAN--BLNK COMN--LENGTH--  
 000100  
 000170  
 000100  
 000100  
 000130  
 000100  
 000130

This load map shows the common block presets,  
 the main program, subprograms, common blocks,  
 and library routines.

L1--L2-----USER-----CALL-----  
 MAIN  
 SUPR  
 MAIN  
 SUPR  
 MAIN  
 SUPR  
 MAIN  
 SUPR  
 LARELFD--COMMON--  
 000100  
 000170  
 000100  
 000100  
 000130  
 000100  
 000130

Program Output

| DEGREES | SINE (DEGREES) | COSINE (DEGREES) |
|---------|----------------|------------------|
| 0.0000  | 0.000000000000 | 1.000000000000   |
| 1.0000  | 0.017452406417 | 0.999847695156   |
| 2.0000  | 0.034899496662 | 0.999790827020   |
| 3.0000  | 0.052333956183 | 0.999629534757   |
| 4.0000  | 0.069756473664 | 0.9993564050265  |
| 5.0000  | 0.087155742644 | 0.9989749698100  |
| 6.0000  | 0.104528463144 | 0.9984821895780  |
| 7.0000  | 0.121869343266 | 0.9978846151568  |
| 8.0000  | 0.139173100801 | 0.997186058763   |
| 9.0000  | 0.156434464462 | 0.98640340623    |
| 10.0000 | 0.17364817470  | 0.984807753046   |
| 11.0000 | 0.190808995161 | 0.981627183449   |
| 12.0000 | 0.207911690583 | 0.977147600783   |
| 13.0000 | 0.224951054091 | 0.971370064447   |
| 14.0000 | 0.241921895328 | 0.964295726343   |
| 15.0000 | 0.258819044813 | 0.955925826366   |
| 16.0000 | 0.275637755510 | 0.946126169626   |
| 17.0000 | 0.292371704394 | 0.935004756062   |
| 18.0000 | 0.309016994033 | 0.921565164495   |
| 19.0000 | 0.325568154094 | 0.905918575722   |
| 20.0000 | 0.342020142950 | 0.939692620922   |
| 21.0000 | 0.358367949154 | 0.933580426647   |
| 22.0000 | 0.374606593009 | 0.927183854771   |
| 23.0000 | 0.390731128067 | 0.920504453631   |
| 24.0000 | 0.406736545787 | 0.913545457837   |
| 25.0000 | 0.422618261288 | 0.906307787247   |
| 26.0000 | 0.438371146322 | 0.898794046526   |
| 27.0000 | 0.453990499259 | 0.891006524432   |
| 28.0000 | 0.469471562292 | 0.882947593121   |
| 29.0000 | 0.484809619740 | 0.874619707419   |
| 30.0000 | 0.499999999481 | 0.866025404683   |
| 31.0000 | 0.515078074380 | 0.857167701020   |
| 32.0000 | 0.529919263691 | 0.848048096494   |
| 33.0000 | 0.544590326697 | 0.838670568303   |

CDC 6000 SERIES JOVIAL COMPILER VERSION 1.0 PSR90\* SOURCE LISTING \*

Time and Date of Compilation  
TIME 12.01.59. DATE 01/07/72 PAGE 00001.

Compiler Version

```

0001. START$
0002. PROC
0003. BDEC (HOL , NUMB = BCDF) $
0004. **1#BEGIN
0005. ITEM HOL H 1 $
0006. ITEM NUMB I 60 S $ Procedure Subprogram
0007. ITEM BCDF H 10 $
0008. **1#END
0009. PROC
0010. **2#BEGIN
0011. ITEM LIT H 10 $
0012. ITEM DT0B I 60 S $ Function Subprogram
0013. **2#END
0014. TERM $

```

```

CP000000
CP000100
CP000200
CP000300
CP000400
CP000500
CP000600
CP000700
CP000800
CP000900
CP001000
CP001100
CP001200

```

This COMPOOL compilation was run using the following control card:

JOVIAL(A=CMPL,N=CMPL)

where

A=CMPL performs a COMPOOL compilation to file CMPL

N=CMPL name of COMPOOL output

```

0001. START $ ## TEST OF CONVERSION PROCEDURES ##
0002. ITEM HOL H 10 $
0003. ITEM STC T 10 $
0004. ITEM INT I 18 $
0005. FILE OUT H 0 R 128 V(OK) OUTPUT $
0006. OPEN OUTPUT OUT $
0007. FOR A = -1,1,20 $ ## TEST HOLLERITH CONVERSION ##
0008. ##1##BEGIN
0009. BDEC(1H(H),A=HOL) $
0010. OUTPUT OUT HOL $
0011. INT = DTOB(HOL) $
0012. IF INT NQ (/A/) $
0013. OUTPUT OUT 10H(ERROR H) $
0014. ##1##END
0015. FOR A = -1,1,20 $ ## TEST STC CONVERSION ##
0016. ##2##BEGIN
0017. BDEC(1H(T),A=STC) $
0018. INT = DTOB(STC) $
0019. IF INT NQ (/A/) $
0020. OUTPUT OUT 10H(ERROR S) $
0021. ##2##END
0022. SHUT OUTPUT OUT $
0023.

```

TERM \$

- CV000000
- CV000100
- CV000200
- CV000300
- CV000400
- CV000500
- CV000600
- CV000700
- CV000800
- CV000900
- CV001000
- CV001100
- CV001200
- CV001300
- CV001400
- CV001500
- CV001600
- CV001700
- CV001800
- CV001900
- CV002000
- CV002100
- CV002200

The batch compilation of the main program and subprograms was run using the following control card:

JOVIAL,O,B=CMPL,N=CDEMO,B=CDEMO.

where

- O object code
- C=CMPL perform the compilation using the COMPOOL in file CMPL
- N=CDEMO program name
- B=CDEMO file location of the object code

Program Name

CDEMO

| LINE   | LOCATION | OCTAL                    | LABEL  | OP                           | OPERANDS |
|--------|----------|--------------------------|--------|------------------------------|----------|
| 000000 |          |                          | OUTPUT | BSSZ 1                       |          |
| 000001 |          | 00000000000000000000     |        | VFD 15/1,26/08,18/OUTPUT=+17 |          |
| 000002 |          | 0000040000000000000021 + |        | VFD 60/OUTPUT=+17            |          |
| 000003 |          | 0000000000000000000021 + |        | VFD 60/OUTPUT=+17            |          |
| 000004 |          | 0000000000000000000022 + |        | BSSZ 12                      |          |
| 000005 |          | 0000000000000000000000   |        | BSS 129                      |          |
| 000021 |          |                          | HOL    | DATA 10A                     |          |
| 000222 |          | 55555555555555555555     | STC    | DATA 8050505050505050505     |          |
| 000223 |          | 05050505050505050505     |        | DATA 975463514113            |          |
| 000226 |          | 0000016143612000001      |        | DATA 10A H                   |          |
| 000227 |          | 555555555555555555510    |        | DATA 412317122561            |          |
| 000230 |          | 00000060000001000001     |        | DATA 0                       |          |
| 000231 |          | 00000000000000000000     |        | DATA 10H ERROR H             |          |
| 000232 |          | 55052222172255105555     |        | DATA 10A T                   |          |
| 000233 |          | 555555555555555555524    |        | DATA 10H ERROR S             |          |
| 000234 |          | 55052222172255235555     |        | DATA 1511828750337           |          |
| 000235 |          | 0000026000001000001      |        | VFD 42/7LOUTPUT ,18/OUT      |          |
| 000236 |          | 1725242025240000000 +    |        | VFD 60/0                     |          |
| 000237 |          | 00000000000000000000     |        | VFD 60/=975463514113         |          |
| 000240 |          | 00000000000000000226 +   |        | VFD 60/OUT                   |          |
| 000241 |          | 00000000000000000000 +   |        | VFD 60/=412317122561         |          |
| 000242 |          | 00000000000000000230 +   |        | VFD 60/OUT                   |          |
| 000243 |          | 00000000000000000000 +   |        | VFD 60/HOL                   |          |
| 000244 |          | 00000000000000000222 +   |        | VFD 60/=0                    |          |
| 000245 |          | 00000000000000000231 +   |        | VFD 60/=0                    |          |
| 000246 |          | 00000000000000000231 +   |        | VFD 60/=412317122561         |          |
| 000247 |          | 00000000000000000230 +   |        | VFD 60/OUT                   |          |
| 000250 |          | 00000000000000000000 +   |        | VFD 60/=10H ERROR S          |          |
| 000251 |          | 00000000000000000232 +   |        | VFD 60/=0                    |          |
| 000252 |          | 00000000000000000231 +   |        | VFD 60/=0                    |          |
| 000253 |          | 00000000000000000231 +   |        | VFD 60/=1511828750337        |          |
| 000254 |          | 00000000000000000230 +   |        | VFD 60/OUT                   |          |
| 000255 |          | 00000000000000000000 +   |        | VFD 42/7LCDEMO ,18/0         |          |
| 000256 |          | 00000000000000000234 +   |        | SA1 G000236                  |          |
| 000257 |          | 00000000000000000231 +   |        | RJ Q8NTRY.                   |          |
| 000260 |          | 00000000000000000231 +   |        | SA1 G000240                  |          |
| 000261 |          | 00000000000000000235 +   |        | NO                           |          |
| 000262 |          | 00000000000000000000 +   |        | NO                           |          |
| 000263 |          | 0304051517555500000      | CDEMO  | RJ JOVIO.                    |          |
| 000264 |          | 5110000236 +             |        | VFD 12/0,18/CDEMO-1          |          |
| 000265 |          | 5110000240 +             |        | MX6 59                       |          |
| 000266 |          | 46000                    |        | SA6 A                        |          |
| 000267 |          | 0100000000 X             |        | NO                           |          |
| 000268 |          | 46000                    |        | NO                           |          |
| 000269 |          | 0100000000 X             |        | RJ JOVIO.                    |          |
| 000270 |          | 0000000253 +             |        | VFD 12/0,18/CDEMO-1          |          |
| 000271 |          | 43673                    |        | MX6 59                       |          |
| 000272 |          | 5160000225 +             |        | SA6 A                        |          |
| 000273 |          | 46000                    |        | NO                           |          |
| 000274 |          | 5110000225 +             |        | SA1 A                        |          |
| 000275 |          | 5120000227 +             |        | SA2 =10H                     |          |
| 000276 |          | 10711                    |        | RX7 X1                       |          |
| 000277 |          | 22602                    |        | LX6 R0,X2                    |          |

00007.

| LINE   | LOCATION | OCTAL        | LABEL | OP  | OPERANDS         |                                                                                                                                                    |
|--------|----------|--------------|-------|-----|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| 000271 |          | 5170000000 X |       | SA7 | BDEC\$\$B        | Store input parameters for subprogram BDEC                                                                                                         |
| 000272 |          | 5160000000 X |       | SA6 | BDEC\$\$A        |                                                                                                                                                    |
| 000272 |          | 46000        |       | NO  |                  |                                                                                                                                                    |
| 000272 |          | 0100000000 X |       | RJ  | BDEC             | Execute BDEC                                                                                                                                       |
| 000273 |          | 0007000253 + |       | VFD | 12/7,18/COEMO-1  |                                                                                                                                                    |
| 000274 |          | 5110000000 X |       | SA1 | BDEC\$\$C        | Fetch output parameter from subprogram BDEC                                                                                                        |
| 000274 |          | 10611        |       | RX6 | X1               |                                                                                                                                                    |
| 000274 |          | 5160000222 + |       | NO  |                  |                                                                                                                                                    |
| 000275 |          | 5110000242 + |       | SA6 | HOL              |                                                                                                                                                    |
| 000275 |          | 0100000000 X |       | SA1 | G000242          |                                                                                                                                                    |
| 000276 |          | 0012000263 + |       | RJ  | JOVIO.           |                                                                                                                                                    |
| 000276 |          | 5110000222 + |       | VFD | 12/10,18/COEMO-1 |                                                                                                                                                    |
| 000277 |          | 10611        |       | SA1 | HOL              |                                                                                                                                                    |
| 000277 |          | 46000        |       | RX6 | X1               |                                                                                                                                                    |
| 000300 |          | 5160000000 X |       | NO  |                  |                                                                                                                                                    |
| 000300 |          | 46000        |       | SA6 | DT0B\$\$A        | Store input parameter for subprogram DTOB                                                                                                          |
| 000300 |          | 0100000000 X |       | NO  |                  |                                                                                                                                                    |
| 000301 |          | 0013000263 + |       | RJ  | DT0B             | Execute subprogram DTOB                                                                                                                            |
| 000302 |          | 5110000225 + |       | VFD | 12/11,18/COEMO-1 |                                                                                                                                                    |
| 000302 |          | 10011        |       | SA1 | A                | Subprogram DTOB has no output parameter; it is a function with the result returned in the same way as a function declared within the main program. |
| 000302 |          | 21073        |       | AX0 | 59               |                                                                                                                                                    |
| 000303 |          | 5160000224 + |       | SA6 | INT              |                                                                                                                                                    |
| 000303 |          | 13210        |       | AX2 | X1-X0            |                                                                                                                                                    |
| 000304 |          | 0300000306 + |       | IX0 | X6-X2            |                                                                                                                                                    |
| 000304 |          | 5110000247 + |       | ZR  | X0,G000306       |                                                                                                                                                    |
| 000305 |          | 0100000000 X |       | SA1 | G000247          |                                                                                                                                                    |
| 000305 |          | 5110000225 + |       | RJ  | JOVIO.           |                                                                                                                                                    |
| 000306 |          | 0014000263 + |       | VFD | 12/12,18/COEMO-1 |                                                                                                                                                    |
| 000306 |          | 7100000024   |       | SA1 | A                |                                                                                                                                                    |
| 000307 |          | 7261000001   |       | SX0 | 20               |                                                                                                                                                    |
| 000307 |          | 54610        |       | SX6 | X1+1             |                                                                                                                                                    |
| 000307 |          | 0321000270 + |       | SA6 | A1               |                                                                                                                                                    |
| 000310 |          | 43673        |       | PL  | X1,X0-X6         |                                                                                                                                                    |
| 000310 |          | 5160000225 + |       | MX6 | X1,G000270       |                                                                                                                                                    |
| 000311 |          | 46000        |       | NO  |                  |                                                                                                                                                    |
| 000311 |          | 5110000225 + |       | SA6 | A                |                                                                                                                                                    |
| 000312 |          | 5120000233 + |       | NO  |                  |                                                                                                                                                    |
| 000312 |          | 10711        |       | SA1 | A                |                                                                                                                                                    |
| 000313 |          | 22602        |       | SA2 | =10H             |                                                                                                                                                    |
| 000313 |          | 5170000000 X |       | RX7 | X1               |                                                                                                                                                    |
| 000314 |          | 51F0000000 X |       | LX6 | B0,X2            |                                                                                                                                                    |
| 000314 |          | 46000        |       | SA7 | BDEC\$\$B        |                                                                                                                                                    |
| 000314 |          | 0100000000 X |       | NO  |                  |                                                                                                                                                    |
| 000314 |          | 0016000263 + |       | SA6 | BDEC\$\$A        |                                                                                                                                                    |
| 000315 |          | 5110000000 X |       | NO  |                  |                                                                                                                                                    |
| 000315 |          | 10611        |       | RJ  | BDEC             |                                                                                                                                                    |
| 000316 |          | 5110000000 X |       | VFD | 12/14,18/COEMO-1 |                                                                                                                                                    |
| 000316 |          | 10611        |       | SA1 | BDEC\$\$C        |                                                                                                                                                    |
| 000316 |          |              |       | RX6 | X1               |                                                                                                                                                    |

| LINE   | LOCATION | OCTAL        | LABEL        | OP  | OPERANDS         |
|--------|----------|--------------|--------------|-----|------------------|
| 00016. | 000316   | 5160000223 + | 46000        | NO  |                  |
|        | 000317   |              |              | SA6 | STC              |
|        | 000317   | 5160000000 X |              | SA6 | DT0B\$\$A        |
|        | 000320   | 0100000000 X |              | RJ  | DT0B             |
|        | 000320   | 0022000263 + |              | VFD | 12/18,18/CDEMO-1 |
|        | 000321   | 5110000225 + |              | SA1 | A                |
|        | 000321   |              | 10011        | BX0 | X1               |
|        | 000321   |              | 21073        | AX0 | 59               |
|        | 000322   | 5160000224 + |              | SA6 | INT              |
|        | 000322   |              | 13210        | BX2 | X1-X0            |
|        | 000322   |              | 37052        | IX0 | X6-X2            |
|        | 000323   | 0300000325 + |              | ZR  | X0,6000325       |
|        | 000323   |              | 5110000254 + | SA1 | 6000254          |
|        | 000324   | 0100000000 X |              | RJ  | JOVIO.           |
|        | 000324   | 0023000263 + |              | VFD | 12/19,18/CDEMO-1 |
|        | 000325   | 5110000225 + | 6000325      | SA1 | A                |
|        | 000325   |              | 7100000024   | SX0 | Z0               |
|        | 000326   | 7261000001   |              | SX6 | X1+1             |
|        | 000326   |              | 54610        | SA6 | A1               |
|        | 000326   | 0321000312 + | 37106        | IX1 | X0-X6            |
|        | 000327   |              | 5110000261 + | PL  | X1,6000312       |
|        | 000330   | 0100000000 X |              | SA1 | 6000261          |
|        | 000330   | 0025000263 + |              | RJ  | JOVIO.           |
|        | 000331   | 0100000000 X |              | VFD | 12/21,18/CDEMO-1 |
|        | 000331   |              | 0027000263 + | RJ  | END.             |
|        | 000331   |              |              | VFD | 12/23,18/CDEMO-1 |
|        |          |              |              | END | CDEMO            |

\*\*PROGRAM LENGTH IS 000328 WORDS\*\*

\*\*\*\*\* THIS COMPILATION REQUIRED 0606508 WORDS OF CORE ----- .925 SECONDS TO COMPILE \*\*\*\*\*

BDEC is a procedure subprogram with two input parameters and one output parameter.

```

0001. START PROC BDEC (HOL , NUMB = BCDF) $
0002. #1#BEGIN
0003. ## CONVERT INTEGER NUMBER TO HOLLERITH OR STC LITERAL AS REQUESTED
0004. ## IF NUMBER IS NEGATIVE ABSOLUTE VALUE WILL BE TAKEN
0005. ## IF NUMBER IS GREATER THAN TEN DIGITS # TOO BIG# WILL BE RETURNED##8000400
0006. ## THE NUMBER WILL BE RIGHT JUSTIFIED WITH LEFT FILL OF BLANKS OF
0007. ## THE TYPE OF CONVERSION REQUESTED
0008. ITEM BCDF H 10 $
0009. ITEM HOL H 1 $
0010. ITEM NUMB I 60 S $
0011. ITEM QUO A 60 S $
0012. ITEM BASE I 18 S $
0013. ITEM FILLER I 60 S $
0014. NUMB = (/NUMB/) $
0015. ##2## IFEITH HOL EQ 1H(H) $
0016. ##3##BEGIN
0017. IF NUMB GR 9999999999 $
0018. #4##BEGIN
0019. BCDF = 10H(TOO BIG) $
0020. RETURN $
0021. #4##END
0022. BASE = 27 $
0023. FILLER = 0(555555555555555555) $##HOLLERITH BLANKS##8000200
0024. ##3##END
0025. ORIF 1 $
0026. #5##BEGIN
0027. FILLER = 0(050505050505050505) $
0028. IF NUMB GR 9999999999 $
0029. #6##BEGIN
0030. BCDF = 10T(TOO BIG) $
0031. RETURN $
0032. #6##END
0033. BASE = 48 $
0034. #2##END
0035. ##7##BEGIN
0036. FOR A = 54,-6,0 $
0037. QUO = NUMB/10 $
0038. BIT($A,6$(BCDF)) = BASE + NUMB - QUO * 10 $
0039. NUMB = QUO $
0040. IF NUMB EQ 0 $
0041. #8##BEGIN
0042. IF A EQ 0 $
0043. RETURN $
0044. BIT($0,A$(BCDF)) = FILLER $
0045. RETURN $
0046. #8##END
0047. ##7##END
0048. #1##END
0049. TERM $
0050.

```



| LINE  | LOCATION | OCTAL                | LABEL   | OP                         | OPERANDS |
|-------|----------|----------------------|---------|----------------------------|----------|
| 00000 |          | 5555555555555555555  | BCDF    | DATA 10A                   |          |
| 00001 |          | 5555555555555555555  | HOL     | DATA 10A                   |          |
| 00007 |          | 5555555555555555555  |         | DATA 10A                   | H        |
| 00010 |          | 7777777777777222267  |         | DATA -187208               |          |
| 00011 |          | 0000000112402761777  |         | DATA 9999999999            |          |
| 00012 |          | 0000000112402762000  |         | DATA 1000000000            |          |
| 00013 |          | 5555524171755021107  |         | DATA 10H TOO BIG           |          |
| 00014 |          | 0055555555555555555  |         | DATA 85555555555555555     |          |
| 00015 |          | 0050505050505050505  |         | DATA 850505050505050505    |          |
| 00016 |          | 05050531242405071614 |         | DATA 05050531242405071614B |          |
| 00017 |          | 1723500000000000000  |         | DATA 275739924060176384    |          |
| 00020 |          | 020405035555500021 + |         | VFD 42/7LBDEC ,18/ADEC     |          |
| 00021 |          | 333434235534353335   | BDEC    | VFD 60/10L01072 1202       |          |
| 00022 |          | 612000000 +          |         | SB2 BCDF                   |          |
| 00022 |          | 6130000002 +         |         | SB3 NUMB                   |          |
| 00023 |          | 6140000004 +         |         | SB4 BASE                   |          |
| 00023 |          | 6150000005 +         |         | SB5 FILLER                 |          |
| 00024 |          | 6160000003 +         |         | SB6 QUO                    |          |
| 00024 |          | 56130                |         | SA1 B3                     |          |
| 00024 |          |                      |         | BX0 X1                     |          |
| 00025 |          | 5120000001 + 10011   |         | SA2 HOL                    |          |
| 00025 |          | 7232222267           |         | SX3 X2+74935               |          |
| 00026 |          | 21073                |         | AX0 59                     |          |
| 00026 |          | 13610                |         | BX6 X1-X0                  |          |
| 00026 |          | 56630                |         | SA6 B3                     |          |
| 00026 |          |                      |         | NO                         |          |
| 00027 |          | 0313000036 + 46000   |         | NZ X3,G000036              |          |
| 00027 |          | 5140000011 +         |         | SA4 =999999999             |          |
| 00030 |          | 37046                |         | IX0 X4-X6                  |          |
| 00030 |          | 0320000033 +         |         | PL X0,G000033              |          |
| 00030 |          |                      |         | NO                         |          |
| 00031 |          | 5150000013 + 46000   |         | SA5 =10H TOO BIG           |          |
| 00031 |          | 22705                |         | LX7 80,X5                  |          |
| 00031 |          | 56720                |         | SA7 B2                     |          |
| 00032 |          | 0400000021 +         |         | EQ BDEC                    |          |
| 00032 |          | 46000                |         | NO                         |          |
| 00032 |          |                      |         | NO                         |          |
| 00033 |          | 5110000014 + 46000   | G000033 | SA1 =85555555555555555     |          |
| 00033 |          | 7170000033           |         | SX7 27                     |          |
| 00034 |          | 56740                |         | SA7 94                     |          |
| 00034 |          | 10711                |         | BX7 X1                     |          |
| 00034 |          | 56750                |         | SA7 B5                     |          |
| 00034 |          |                      |         | NO                         |          |
| 00035 |          | 0400000045 + 46000   |         | EQ G000045                 |          |
| 00035 |          | 46000                |         | NO                         |          |
| 00035 |          |                      |         | NO                         |          |
| 00036 |          | 7100000001 46000     | G000036 | SX0 1                      |          |
| 00036 |          | 0300000045 +         |         | ZR X0,G000045              |          |
| 00037 |          | 5120000015 +         |         | SA2 =850505050505050505    |          |
| 00037 |          | 5130000011 +         |         | SA3 =9999999999            |          |
| 00040 |          | 22702 37036          |         | LX7 80,X2                  |          |
| 00040 |          | 56750                |         | IX0 X3-X6                  |          |
| 00040 |          |                      |         | SA7 B5                     |          |

| LINE   | LOCATION | OCTAL        | LABEL        | OP  | OPERANDS               |
|--------|----------|--------------|--------------|-----|------------------------|
| 00027. | 000040   |              |              | NO  |                        |
|        | 000041   | 0320000043 + | 46000        | PL  | X0,G000043             |
|        | 000042   | 10644        | 5140000016 + | SA4 | =805050531242405071614 |
|        | 000043   | 56620        |              | SA6 | B2                     |
| 00030. | 000042   |              |              | EQ  | BDEC                   |
| 00031. | 000043   | 7170000060   | 0400000021 + | SX7 | 48                     |
|        | 000044   | 56740        | G000043      | SAT | B4                     |
| 00033. | 000043   |              |              | NO  |                        |
|        | 000044   | 0400000045 + | 46000        | EQ  | G000045                |
|        | 000045   | 46000        |              | NO  |                        |
|        | 000046   | 6110000066   | G000045      | NO  |                        |
|        | 000047   | 46000        | G000045      | SBI | 54                     |
|        | 000048   |              |              | NO  |                        |
|        | 000049   | 56530        | 46000        | NO  |                        |
|        | 000050   | 5110000017 + | 46000        | SA5 | B3                     |
|        | 000051   | 44431        | G000046      | SA1 | =275739924060176384    |
|        | 000052   | 56240        |              | MX0 | 6                      |
|        | 000053   | 27305        | 43006        | SA2 | B4                     |
|        | 000054   | 617177703    |              | PX3 | B0,X5                  |
|        | 000055   | 36125        |              | SB7 | B1+-60                 |
|        | 000056   | 56320        |              | FX4 | X3/X1                  |
|        | 000057   | 23270        |              | IX1 | X2+X5                  |
|        | 000058   | 15032        |              | SA3 | B2                     |
|        | 000059   | 76370        |              | AX2 | B7,X0                  |
|        | 000060   | 26574        |              | BX0 | -X2+X3                 |
|        | 000061   | 22775        |              | SX3 | B7                     |
|        | 000062   | 56760        |              | UX5 | B7,X4                  |
|        | 000063   | 6273000006   |              | LX7 | B7,X5                  |
|        | 000064   | 10477        |              | SA7 | B6                     |
|        | 000065   | 56730        |              | SB7 | X3+6                   |
|        | 000066   | 37315        |              | IX3 | X7+X7                  |
|        | 000067   | 23173        |              | BX4 | X7                     |
|        | 000068   | 11421        |              | SA7 | B3                     |
|        | 000069   | 56620        |              | LX4 | 3                      |
|        | 000070   | 0317000063 + | 36543        | IX5 | X4+X3                  |
|        | 000071   | 0510000057 + | 46000        | IX3 | X1-X5                  |
|        | 000072   | 43001        |              | AX1 | B7,X3                  |
|        | 000073   | 56450        |              | BX4 | X2+X1                  |
|        | 000074   | 6171777704   |              | BX6 | X4+X0                  |
|        | 000075   | 22170        |              | SA6 | B2                     |
|        | 000076   | 6171777703   |              | NZ  | X7,G000063             |
|        | 000077   | 23271        |              | NO  |                        |
|        | 000078   | 11126        |              | NE  | B1,G000057             |
|        | 000079   |              |              | EQ  | BDEC                   |
|        | 000080   |              |              | MX0 | 1                      |
|        | 000081   |              |              | SA4 | B5                     |
|        | 000082   |              |              | SB7 | B1+-59                 |
|        | 000083   |              |              | LX1 | B7,X0                  |
|        | 000084   |              |              | SB7 | B1+-60                 |
|        | 000085   |              |              | BX0 | -X1+X4                 |
|        | 000086   |              |              | AX2 | B7,X1                  |
|        | 000087   |              |              | BX1 | X2+X5                  |

| LINE   | LOCATION | OCTAL        | LABEL   | OP  | OPERANDS   |
|--------|----------|--------------|---------|-----|------------|
|        | 000061   | 23370        |         | AX3 | B7,X0      |
|        | 000061   |              | 12713   | BX7 | X1,X3      |
|        | 000062   | 56720        |         | SA7 | B2         |
| 00045. | 000062   | 0400000021 + |         | EQ  | BDEC       |
|        | 000062   |              | 46000   | NO  |            |
| 00046. | 000063   | 611177771    | 6000063 | SBI | B1+-6      |
|        | 000063   | 0610000046 + |         | GE  | B1,6000046 |
|        | 000064   | 0400000021 + |         | EQ  | BDEC       |
|        | 000064   | 46000        |         | NO  |            |
|        | 000064   | 46000        |         | NO  |            |
|        |          |              |         | END |            |

\*\*PROGRAM LENGTH IS 000065R WORDS\*\*

\*\*\*\*\* THIS COMPILATION REQUIRED 0605528 WORDS OF CORE ---- 1.052 SECONDS TO COMPILE \*\*\*\*\*

```

0001. START PROC DTOB (LIT) $
0002. **1*#BEGIN
0003. ** DTOB IS A FUNCTION TO CONVERT NUMBERS IN HOLLERITH OR STC
0004. ** FORMAT TO BINARY FORMAT
0005. ** OR STC NUMERIC VALUE THEY WILL BE SKIPPED WITH NO MESSAGE
0006. ** IF ANY CHARACTERS ARE FOUND WHICH DO NOT HAVE A HOLLERITH
0007. ** ITEM LIT H 10 $
0008. ** INPUT LITERAL
0009. ** FUNCTION OUTPUT PARAMETER
0010. ** DTOB I 60 S $
0011. ** TEMPOARY FOR MULTIPLICATION FACTOR
0012. ** TEMPORARY FOR BYTE BEING CONVERTED
0013. ** INITIALIZE OUTPUT
0014. ** INITIALIZE MULTIPLICATION FACTOR
0015. ** FLOOP THOUGH BYTES
0016. HLD = BIT($A,68) (LIT) $
0017. IF HLD LS 27 $
0018. **3*#BEGIN
0019. VAL = VAL * 10 $
0020. TEST $
0021. **3*#END
0022. IF HLD LS 37 $
0023. **4*#BEGIN
0024. HLD = HLD - 27 $
0025. GOTO COMP $
0026. **4*#END
0027. IF HLD LS 48 $
0028. **5*#BEGIN
0029. VAL = VAL * 10 $
0030. TEST $
0031. **5*#END
0032. IF HLD LS 58 $
0033. **6*#BEGIN
0034. HLD = HLD - 48 $
0035. GOTO COMP $
0036. **6*#END
0037. VAL = VAL * 10 $
0038. TEST $
0039. DTOB = DTOB + HLD * VAL $
0040. VAL = VAL * 10 $
0041. **2*#END
0042. **1*#END
COMP.
TERM $

```

DT000100  
DT000200  
#DT000300  
#DT000400  
#DT000500  
#DT000600  
#DT000700  
#DT000800  
#DT000900  
#DT001000  
#DT001100  
#DT001200  
DT001400  
DT001500  
DT001600  
DT001700  
DT001800  
DT001900  
DT002000  
DT002100  
DT002200  
DT002300  
DT002400  
DT002500  
DT002700  
DT002800  
DT002900  
DT003000  
DT003200  
DT003300  
DT003400  
DT003500  
DT003600  
DT003700  
DT003800  
DT003900  
DT004000  
DT004100  
DT004200

DT0B is an integer function  
subprogram. The result is the  
value of integer variable DT0B  
when control returns to the  
calling program.

| LINE   | LOCATION               | OCTAL | LABEL   | OP                     | OPERANDS |
|--------|------------------------|-------|---------|------------------------|----------|
| 00000  | 55555555555555555555   |       | LIT     | DATA 10A               |          |
| 00005  | 04241702555555000006 + |       | DT08    | VFD 42/710T08 ,1M/DT08 |          |
| 00006  | 33743342355534353335   |       |         | VFD 60/10L01072 1202   |          |
| 00007  | 6120000002 +           |       |         | S82 VAL                |          |
| 00008  | 6140000001 +           |       |         | S83 HLD                |          |
| 00009  | 6160777771             |       |         | S84 DT08               |          |
| 00010  | 6150000000 +           |       |         | S85 LIT                |          |
| 00011  | 7160000001             |       |         | S86 -6                 |          |
| 00012  | 43700                  |       |         | SX6 1                  |          |
| 00012. | 56620                  |       |         | MX7 0                  |          |
| 00013  | 6110000066             |       |         | SA6 B2                 |          |
| 00013. | 56740                  |       |         | S81 54                 |          |
| 00014  | 46000                  |       |         | SA7 B4                 |          |
| 00015  | 46000                  |       |         | NO                     |          |
| 00016  | 46000                  |       |         | NO                     |          |
| 00017  | 46000                  |       |         | NO                     |          |
| 00018  | 43006                  |       | G000014 | MX0 6                  |          |
| 00019  | 56150                  |       |         | SA1 B5                 |          |
| 00020  | 7120777744             |       |         | SX2 -27                |          |
| 00021  | 22311                  |       |         | LX3 B1,X1              |          |
| 00022  | 11703                  |       |         | BX7 X0*X3              |          |
| 00023  | 20706                  |       |         | LX7 6                  |          |
| 00024  | 36072                  |       |         | SA7 B3                 |          |
| 00025  | 0320000021 +           |       |         | IX0 X7+X2              |          |
| 00026  | 56320                  |       |         | PL X0,G000021          |          |
| 00027  | 20303                  |       |         | SA3 B2                 |          |
| 00028  | 36033                  |       |         | IX0 X3+X3              |          |
| 00029  | 20303                  |       |         | LX3 3                  |          |
| 00030  | 36630                  |       |         | IX6 X3+X0              |          |
| 00031  | 56620                  |       |         | SA6 B2                 |          |
| 00032  | 0400000041 +           |       |         | EQ G000041             |          |
| 00033  | 46000                  |       |         | NO                     |          |
| 00034  | 46000                  |       |         | NO                     |          |
| 00035  | 36170                  |       | G000021 | SX0 -37                |          |
| 00036  | 46000                  |       |         | IX1 X7+X0              |          |
| 00037  | 7100777732             |       |         | NO                     |          |
| 00038  | 0321000024 +           |       |         | PL X1,G000024          |          |
| 00039  | 36772                  |       |         | IX7 X7+X2              |          |
| 00040  | 56730                  |       |         | SA7 B3                 |          |
| 00041  | 0400000035 +           |       |         | EQ COMP                |          |
| 00042  | 46000                  |       |         | NO                     |          |
| 00043  | 46000                  |       |         | NO                     |          |
| 00044  | 56430                  |       | G000024 | SA4 B3                 |          |
| 00045  | 7100777717             |       |         | SX0 -48                |          |
| 00046  | 36140                  |       |         | IX1 X4+X0              |          |
| 00047  | 0321000030 +           |       |         | PL X1,G000030          |          |
| 00048  | 56520                  |       |         | SA5 B2                 |          |
| 00049  | 36155                  |       |         | IX1 X5+X5              |          |
| 00050  | 20503                  |       |         | LX5 3                  |          |
| 00051  | 36651                  |       |         | IX6 X5+X1              |          |
| 00052  | 56620                  |       |         | SA6 B2                 |          |
| 00053  | 0400000041 +           |       |         | NO                     |          |
| 00054  | 46000                  |       |         | EQ G000041             |          |

| LINE   | LOCATION | OCTAL        | LABEL    | OP  | OPERANDS    |
|--------|----------|--------------|----------|-----|-------------|
| 00027  |          | 46000        |          | NO  |             |
| 00027  |          | 46000        |          | NO  |             |
| 00029. |          | 7110777705   | G0000030 | SX1 | -58         |
| 00030  |          | 36241        |          | IX2 | X4+X1       |
| 00030  |          | 46000        |          | NO  |             |
| 00031  |          | 0322000033 + |          | PL  | X2,G0000033 |
| 00031  |          | 36740        |          | IX7 | X4+X0       |
| 00031  |          | 56730        |          | SA7 | B3          |
| 00032  |          | 0400000035 + |          | EQ  | COMP        |
| 00032  |          | 46000        |          | NO  |             |
| 00032  |          | 46000        |          | NO  |             |
| 00033  |          | 56120        | G0000033 | SA1 | B2          |
| 00033  |          | 36011        |          | IX0 | X1+X1       |
| 00033  |          | 20103        |          | LX1 | 3           |
| 00033  |          | 36610        |          | IX6 | X1+X0       |
| 00034  |          | 56620        |          | SA6 | B2          |
| 00034  |          | 0400000041 + |          | EQ  | G0000041    |
| 00034  |          | 46000        |          | NO  |             |
| 00035  |          | 56220        | COMP     | SA2 | B2          |
| 00035  |          | 56330        |          | SA3 | B3          |
| 00035  |          | 27003        |          | PX0 | B0+X3       |
| 00035  |          | 36122        |          | IX1 | X2+X2       |
| 00036  |          | 56440        |          | SA4 | B4          |
| 00036  |          | 10522        |          | BX5 | X2          |
| 00036  |          | 27602        |          | PX6 | B0+X2       |
| 00036  |          | 42206        |          | DX2 | X0+X6       |
| 00037  |          | 20503        |          | LX5 | 3           |
| 00037  |          | 36751        |          | IX7 | X5+X1       |
| 00037  |          | 26002        |          | UX0 | X2          |
| 00037  |          | 36640        |          | IX6 | X4+X0       |
| 00040  |          | 56720        |          | SA7 | B2          |
| 00040  |          | 56640        |          | SA6 | B4          |
| 00040  |          | 46000        |          | NO  |             |
| 00040  |          | 46000        |          | NO  |             |
| 00041  |          | 66116        | G0000041 | SBI | B1+B6       |
| 00041  |          | 0610000014 + |          | GE  | B1,G0000014 |
| 00041  |          | 46000        |          | NO  |             |
| 00042  |          | 5110000001 + |          | SA1 | DT08        |
| 00042  |          | 10611        |          | BX6 | X1          |
| 00042  |          | 46000        |          | NO  |             |
| 00043  |          | 0400000006 + |          | EQ  | DT08        |
| 00043  |          | 46000        |          | NO  |             |
| 00043  |          | 46000        |          | NO  |             |
| 00043  |          | 46000        |          | END |             |

\*\*PROGRAM LENGTH IS 0000448 WORDS\*\*

\*\*\*\*\* THIS COMPILATION REQUIRED 0605058 WORDS OF CORE ----- .875 SECONDS TO COMPILE \*\*\*\*\*

---LAME---LOAD CODE ---LAME---TABLES 052741  
 FWA LOADER 054756 FWA TABLES 052741  
 --PROGRAM-----ADDRESS--  
 COEMO 000100  
 BDEC 000432  
 DT08 000517  
 GETBA 000563  
 SYSTEM\$ 000602  
 BACKSP\$ 001605  
 OUTPTB\$ 002132  
 REWINH\$ 002404  
 JOVIC\$ 002454  
 SIO\$ 003047  
 INPUTB\$ 004462  
 ---ENTRY-----ADDRESS--  
 COEMO. 000100  
 CDEMO 000364  
 OUTPUT= 000100  
 BDEC. 000432  
 BDEC 000453

---LAME---COMMON---

BDFC\$A 000433  
 BDEC\$B 000434  
 BDEC\$C 000432  
 DT0B. 000517  
 DT0B 000525  
 DT0B\$A 000517  
 GETBA 000564  
 QANTRY. 000606  
 END. 000622  
 EXIT\$ 000644  
 STOP. 000652  
 ABNORM. 000663  
 SYSTEME 000700  
 SYSTEM\$ 000724  
 SYSTEM. 000730  
 SYSTEMI 000765



000373  
 000415  
 000372  
 000414  
 000371  
 000413  
 000374  
 000416

REFERENCES

Entry points for passing parameters between subprogram BDEC and the calling programs. The suffixes A, B, or C are assigned from left to right for the formal parameters specified in the subprogram declaration. The relative address within the subprogram is dependent upon the order in which the data declarations for the formal parameters occur within the body of the subprogram. For example, in the subprogram declaration for BDEC, the order of parameters is HOL, NUMB and BCDP; thus, HOL has suffix A, NUMB suffix B and BCDP suffix C. The order of data declarations within the subprogram is BCDP, HOL and NUMB, hence the difference in the address order and parameter suffix order.



000400  
 000417

Entry point for passing the input parameter to function subprogram DT0B.

BACKSP\$ 001614  
 OUTPTB\$ 002154  
 REWINH\$ 002413  
 SIO\$ 003322  
 INPUTB\$ 004523

For additional information, refer to the SCOPE Operating System Reference Manual (Loader Map Section) listed in the preface of this manual.





|         |        |          |        |
|---------|--------|----------|--------|
| POF1.   | 004302 | PAKCP3   | 002037 |
| MVMS.   | 004432 | REINH\$  | 002836 |
| SYERR.  | 004443 | OUTPTB\$ | 002275 |
| JOV.    | 004513 | OUTPTB\$ | 002240 |
| LINKH.  | 004514 | INPUTB\$ | 004730 |
| IPUTBI. | 004520 | JOVIO\$  | 002477 |
| INPUTB. | 004557 | JOVIO\$  | 002456 |
|         |        | JOVIO\$  | 002521 |
|         |        | JOVIO\$  | 002571 |
|         |        | JOVIO\$  | 002641 |
|         |        | JOVIO\$  | 002643 |
|         |        | JOVIO\$  | 002644 |
|         |        | JOVIO\$  | 002705 |
|         |        | JOVIO\$  | 002715 |

REFERENCES

----UNSATISFIED EXTERNALS-----

- 1
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

Program Output

01/07/72 \*\*SCOPE\*\*3.3 PSR284 01/05/72  
 12.01.57.CDEMO33  
 12.01.57.CDEMO,CM6100,177.  
 12.01.57.JOVIAL(A=CMPL,N=CMPL)  
 12.02.03.JOVIAL,O=C=CMPL,N=CDEMO,B=CDEMO.  
 12.02.35.CDEMO.  
 12.02.39.END CDEMO  
 12.02.39.CP 004.217 SEC.  
 12.02.39.PP 024.945 SEC.  
 12.02.39.IO 004.653 SEC.

Dayfile

## SAMPLE PROGRAMS

I

---

This appendix contains three sample programs which combined with the sample listings in Appendix H, demonstrate the usage of several types of JOVIAL code.

Sample program one demonstrates the object time compatibility between JOVIAL and FORTRAN Extended routines. A JOVIAL program calls a FORTRAN subroutine which then calls a JOVIAL subprogram. Both JOVIAL and the FORTRAN routines write on the file, declared in the JOVIAL program.

Sample program two is a utility program used to punch a sequence numbered deck. Appendix M demonstrates how this routine may be placed on the system.

Sample program three is a utility program used to punch UPDATE correction cards. The program scans the compiled file for a specified bit pattern. If a match is found, the field is replaced by the desired bit pattern. A \*DELETE card with the required IDENT and sequence number is punched along with the corrected card. While this may produce some cards which have no purpose, it is more efficient to sort them out than to scan a large program by hand to find specific cards. The output shown is partial output from a change to the compiler to eliminate an entry point conflict between the JOVIAL library and the FORTRAN Extended 3.0 library.

# PROGRAM 1

```

JVFTN,CM70000,T77.
JOVIAL{A=CMP4,N=CMP4,XR}
JOVIAL{M,C=CMP4,XR}
FTN.
MAP{OFF}
LGO.
REWIND{TABEQ}
COPYBF{TAPE9,OUTPUT}
789
START *
 'COMPOOL FOR JOVIAL FORTRAN COMMUNICATION SERIES'
 DEFINE WS '60' *
 '*****'
 'COMPOOL FOR JOVCOM'
 '*****'
 PROC FORCOM {INTGR,INCR,WL,ARAY} *
 PROC JOVPRC {FINTGR,FINCR,ARAY} *
 BEGIN
 ITEM FINTGR I WS S *
 ITEM FINCR F *
 ARRAY ARAY 10 I WS S *
 ITEM FWL F P 10 *
 ITEM MARAY I WS S *
 END
 TERM *
 789
 START
 'JOVIAL FORTRAN COMMUNICATION SERIES'
 'JOVIAL EXTERNAL PROCEDURE CALLED BY FORTRAN SUBPROGRAM'
 PROC JOVPRC {FINTGR,FINCR,ARAY} *
 DEFINE WS '60' *
 ARRAY FINT 1 I WS S * ARRAY FINC 1 I WS S *
 DEFINE FINTGR 'FINT{#0#}' * DEFINE FINCR 'FINC{#0#}' *
 ARRAY ARAY 10 I WS S * 'ARAY FR JOV MAIN PROG'
 ITEM FWL I WS S P 10 *
 ITEM MARAY I WS S *
 ITEM MSG H 6 P 6H{ } *
 MONITOR MSG *
 BEGIN
 MSG = 6H{JOVPRC} *
 FOR I=0,1,FWL-1 *
 BEGIN
 IF ARAY{#I#} N0 FINTGR+2+I * 'CHECK FORTRAN SUBROUTINE'
 BEGIN 'COMPUTATION'
 MSG = 6H{ERR 01} *
 MONITOR MARAY *
 MARAY = ARAY{#I#} *
 END
 ARAY{#I#} = ARAY{#I#} - 1 * 'DECREMENT ARRAY ELEMENTS'
 END
 MSG = 6H{EOJPRC} *
 RETURN *
 END
 TERM *
 START *
 'JOVIAL-FORTRAN COMMUNICATION SERIES'

```

} Control card file

```

''JOVIAL CALLING FORTRAN SUBROUTINE AND RETURN''
 DEFINE WS '60' *
ITEM MSG H b P bHC } *
ITEM INTGR I WS S P 0 *
ITEM INCR F WS S P 1 *
ITEM WL I WS S P 10 *
ARRAY ARAY 10 I WS S *
ITEM MARRAY I WS S P 0 *
 FILE BAD H 20 V 200 V{OPN} V{EOF} V{LER} TAPE9 *
JFCTST.
 MONITOR MSG *
 MSG = bH{JOVCOM} *
 FORCOM {INTGR,INCR,WL,ARAY} *
 FOR I=0,1,WL-1 *
 BEGIN
 IF ARAY{#I#} NQ I+1 *
 BEGIN
 MSG = bH{ERR 03} *
 MONITOR MARRAY *
 MARRAY = ARAY{#I#} *
 END
 END
 MSG=bH{E0JJOV}*
 TERM JFCTST *
789
C JOVIAL FORTRAN COMMUNICATION SERIES
C FORTRAN CALLING JOVIAL SUBPROGRAM AND RETURN
 SUBROUTINE FORCOM {INTGR,INCR,WL,ARAY}
C FORTRAN SUBROUTINE CALLED BY JOVIAL MAIN PROGRAM
C DYNAMICALLY PRESET AN ARRAY WITH A CONSECUTIVE SET OF INTEGERS
 INTEGER FINTGR
 REAL FINCR
 INTEGER WL
 INTEGER INTGR
 REAL INCR
 INTEGER ARRAY{WL}
 DO 5 I=1,WL,1
5 ARRAY{I}=INTGR+INCR+I
 FINTGR = INTGR
 FINCR = INCR
 CALL JOVPRC {FINTGR,FINCR,ARAY}
 DO 7 I=1,WL,1
 IF {ARRAY{I}.EQ.{FINTGR+I}} GO TO 7
 WRITE{9,4}ARRAY{I}
7 CONTINUE
 4 FORMAT {22HFPROC ERR 010 ARRAY{I},I8}
 WRITE{9,10}
10 FORMAT{90H FORTRAN SUBR FORCOM WRITING ON UNIT 9, DEFINED IN JOVIA
 *L CALLING PROGRAM JOVCOM AS TAPE9.}
 RETURN
 END
6789

```

**DATA INPUT**

No input.

**DATA OUTPUT TO PRINTER**

```
*** MONITORED HOLLERITH DATA MSG = JOVCOM
*** MONITORED HOLLERITH DATA MSG = JOVPRC
*** MONITORED HOLLERITH DATA MSG = EOJPRC
*** MONITORED HOLLERITH DATA MSG = EOJJOV
FORTRAN SUBR FORCOM WRITING ON UNIT 9, DEFINED IN JOVIAL CALLING PROGRAM
JOVCOM AS TAPE9.
```

## PROGRAM 2

```

SEQD,CM70000,T100.
JOVIAL{F,N=SEQD}
LG0.
789
} Control Cards
START * ' READ A DECK, PUNCH A NEW DECK WITH COLUMNS 73 AND 74 OF THE
 FIRST CARD GANGPUNCHED IN THE NEW DECK AND COLUMNS 75-80
 SEQUENCE NUMBERED BY 100 '
ITEM BASE I 18 S P 100 *
ITEM FIX A 60 S P 100 *
ITEM GP H 2 *
ITEM CARD H 80 *
ITEM DUMMY H 90 *
ITEM BLANK H 10 *
OVERLAY DUMMY = BLANK , CARD *
FILE IN H 32000 R 129 V{OK} V{EOF} INPUT *
FILE PRT H 0 R 129 V{OK} OUTPUT *
FILE OUT H 32000 R 129 V{OK} PUNCH *
OPEN INPUT IN *
OPEN OUTPUT OUT *
OPEN OUTPUT PRT *
OUTPUT PRT 10H{L } *
INPUT IN CARD *
IF IN EQ V{EOF} *
GOTO QUIT *
GP = BYTE{#72,2*}{CARD} *
NEW. BYTE{#72,2*}{CARD} = GP *
 BYTE{#74,6 *}{CARD} = BDEC{FIX} *
OUTPUT OUT CARD *
OUTPUT PRT DUMMY *
FIX = FIX + BASE *
INPUT IN CARD *
IF IN NQ V{EOF}*
QUIT. GOTO NEW *
SHUT INPUT IN *
SHUT OUTPUT OUT*
SHUT OUTPUT PRT *
STOP *
PROC BDEC{NUMB} *
 '1'BEGIN
 ITEM BDEC H 10 *
 ITEM NUMB A 60 S *
 ITEM REM A 60 S *
 ITEM QU0 A 60 S *
 FOR A = 9, -1, 0 *
 '2'BEGIN
 QU0 = NUMB/10 *
 BIT{#6*A,6*}{BDEC} = 27 + NUMB - QU0 * 10 *
 NUMB = QU0 *
 '2'END
 '1'END
TERM *
789

```

**DATA INPUT**

```
START * ' TEST OF INTEGER TRUNCATION FOR INDEX SWITCH'
MODE F *
MONITOR ZERO. , ONE. ,TWO. *
FLT = 0.1 *
GOTO NUMB{* FLT + 1.1 *} *
ZERO.
STOP *
ONE.
STOP *
TWO.
STOP *
SWITCH NUMB ={ZERO,ONE,TWO} *
TERM *
L789
```

IS  
└─ Column 73

**DATA OUTPUT TO PRINTER AND PUNCH**

```
START * ' TEST OF INTEGER TRUNCATION FOR INDEX SWITCH' IS000100
MODE F * IS000200
MONITOR ZERO. , ONE. ,TWO. * IS000300
FLT = 0.1 * IS000400
GOTO NUMB{* FLT + 1.1 *} * IS000500
ZERO. IS000600
STOP * IS000700
ONE. IS000800
STOP * IS000900
TWO. IS001000
STOP * IS001100
SWITCH NUMB ={ZERO,ONE,TWO} * IS001200
TERM * IS001300
```

Column 73 └─

### PROGRAM 3

```
INFL,CM70000,T77
ATTACH(OLDPL,PF4,LD=JOV)
UPDATE(Q,L=0)
JOVIAL.
LGO(COMPILE,PUNCH)
789
*COMPILE SYMIO
789
START *
```

} Control Cards

```
" THIS PROGRAM -
 SCANS UPDATE COMPILE FILE FOR 'PRINT'
 REPLACES WITH ' PRNT'
 BUILDS '*DELETE' CARD
 OUTPUTS CHANGED CARD AND DELETE CARD
FILE IN H O R 1024 V{OK} V{EOF} CT * '*COMPILE FILE INPUT'
FILE OUT H O R 129 V{OK} CRD * '*CARD OUTPUT'
ITEM CPL H 90 * '*COMPILE FILE 90 COLUMN IMAGE'
ITEM CARD H 80 * '*CARD IMAGE'
OVERLAY CARD = CPL *
ITEM PA I 60 S P 0{2022111624} * '*PRINT'
ITEM DEL H 80 * '*DELETE CARD'
 BYTE{#0,7*}{DEL} = 7H{*DELETE} *
 OPEN INPUT IN *
LOOP. OPEN OUTPUT OUT *
 INPUT IN CPL *
 IF IN EQ V{EOF} *
 '1'BEGIN
 SHUT INPUT IN *
 SHUT OUTPUT OUT *
 STOP *
 '1'END
 FOR A = 0,6,420 * '*SCAN FOR 'PRINT' '
 '2'BEGIN
 IF BIT{#A,30*} {CPL} NQ PA *
 TEST A *
 BIT{#A,30*} {CPL} = 0{55202211624} * '*PRNT'
 FOR Z = 73,1,79 * '*SCAN FOR END OF IDENT '
 '3'BEGIN
 IF BYTE{#Z*} {CPL} NQ 1H{ } *
 TEST Z *
 BYTE{#8,Z-73*} {DEL} = BYTE{#73,Z-73*} {CPL} *
 BYTE{#8+Z-73*} {DEL} = 1H{.} *
 FOR Y = Z,1,89 * '*SCAN FOR NUMBER'
 '4'BEGIN
 IF BYTE {#Y*} {CPL} EQ 1H{ } *
 TEST Y *
 BYTE{#9+Z-73,89-Y*} {DEL} = BYTE{#Y,89-Y*} {CPL} *
 OUTPUT OUT DEL *
 OUTPUT OUR CARD *
 BYTE{#7,13*} {DEL} = 1H{ } *
 GOTO LOOP *
 '4'END
 '3'END
 '2'END
 GOTO LOOP*
TERM *
6789
```




**DATA INPUT**

Program SYMIO

**PROGRAM OUTPUT TO PUNCH**

|                       |                    |  |        |
|-----------------------|--------------------|--|--------|
| *DELETE SYMIO.3       |                    |  |        |
| ENTRY PRNT            |                    |  | SYMIO  |
| *DELETE SYMIO.6       |                    |  |        |
| ENTRY PRNTFL          |                    |  | SYMIO  |
| *DELETE VB0825.99     |                    |  |        |
| ENTRY PRNT*           |                    |  | VB0825 |
| *DELETE VB0825.102    |                    |  |        |
| PRNT* BSS 0           |                    |  | VB0825 |
| *DELETE SYMIO.7       |                    |  |        |
| PRNT BSSZ ↓           |                    |  | SYMIO  |
| *DELETE SYMIO.9       |                    |  |        |
| SA5 PRNT              | GET RETURN ADDRESS |  | SYMIO  |
| *DELETE SYMIO.11      |                    |  |        |
| SA7 PRNTFL            | IN PRINTFL ENTRY   |  | SYMIO  |
| *DELETE SYMIO.13      |                    |  |        |
| PRNTFL BSSZ ↓         |                    |  | SYMIO  |
| *DELETE SYMIO.32      |                    |  |        |
| E0 PRNTFL             |                    |  | SYMIO  |
| *DELETE SYMIO.36      |                    |  |        |
| TRACE. VFD 60/7L PRNT | TRACE INFO         |  | SYMIO  |

Column 74 

## FIXED-POINT SCALING

J

---

Integers used in computations with fixed-point items are treated as if the decimal point were at the right of the low-order digit.

If the operands are integer and fixed-point, both operands and the result have three attributes in common. These attributes (with a two-letter mnemonic) are:

- I — the number of integer bits required to contain the value. Represented for operands 1 and 2 and the result by  $I_1$ ,  $I_2$ ,  $I_R$  respectively.
- A — the precision or number of fractional bits. Represented by  $A_1$ ,  $A_2$ ,  $A_R$  respectively.
- M — the number of bits required to contain the absolute minimum magnitude of the operand or result. Represented by  $M_1$ ,  $M_2$ ,  $M_R$  respectively.

Other subscripts which may be used in place of 1, 2, or R include:

- N — numerator
- D — denominator
- A — fixed operand
- I — integer operand

Examples:

- $3 * 5.25A1$   $A_1 = 0, I_1 = 2, M_1 = 2$   
 $A_2 = 1, I_2 = 3, M_2 = 4$
- ITEM AA 17 U 25 ... 100 \$  $A = 0, I = 7, M = 5$   
If the range had not been specified, M would equal 1. The value assumed is 0, which requires one bit for representation.
- ITEM BB A 7 S 2 0.5A2...14 \$  $A = 2, I = 4 (7 - 2 - 1 \text{ for sign}), M = 2$

The rules for computing these attributes for results are given in the following pages.

## ADDITION AND SUBTRACTION

If both operands are integer, the result is integer, i. e. ,  $A_R = 0$ ; otherwise the result is fixed-point.

$$I_R = 1 + \max (I_1, I_2)$$

If  $A_1 = A_2$ , then

$$A_R = A_1$$

If one operand is integer and  $A_A \geq 0$ , then

$$A_R = A_A;$$

otherwise

$$A_R = 1 + \min (A_1, A_2)$$

If the operation is addition and both operands are unsigned, positive constants or absolute values, then

$$M_R = A_R + \max (M_1 - A_1, M_2 - A_2)$$

otherwise

$$M_R = 1$$

## MULTIPLICATION

If both operands are integer, the result is integer; otherwise the result is fixed-point.

$$I_R = I_1 + I_2$$

If both operands are integer,

$$M_R = M_1 + M_2 - 1$$

If both operands are fixed-point,

$$A_R = A_1 + A_2 + 1 - \max (M_1, M_2)$$

$$M_R = \min (M_1, M_2)$$

otherwise

$$A_R = A_A + 1 - M_I$$

$$M_R = M_A$$

## DIVISION

If both operands are integer,

$$I_R = I_N + 1 - M_D$$

$$A_R = 48 - I_R$$

otherwise

$$I_R = I_N + A_D + 1 - M_D$$

If the numerator is fixed,

$$A_R = I_D + A_N$$

otherwise

$$A_R = 2 * I_D + A_D - M_N$$

The result is always fixed-point.

$$M_R = \max(1, M_N - M_D)$$

## EXPONENTIATION

The result is floating-point unless the base (B) is not floating, the exponent (E) is an integer constant, and  $(\text{exponent} * \log_2(\text{base})) < 48$ .

The result scaling is:

For an integer base

for  $E > 0$

$$I_R = E * I_B$$

$$A_R = 0$$

$$M_R = E * M_B - E + 1$$

for E = 0

$$I_R = 1$$

$$A_R = 0$$

$$M_R = 1$$

for E < 0

$$I_R = 1 - [E] * (M_B - 1)$$

$$A_R = 47 + [E] * (M_B - 1)$$

$$M_R = 1$$

For a fixed base

for E > 0

$$I_R = E * I_B$$

$$A_R = E * (A_B - M_B + 1) + M_B - 1$$

$$M_R = M_B$$

for E = 0

$$I_R = 1$$

$$A_R = 0$$

$$M_R = 1$$

for E < 0

$$I_R = [E] * (A_B - M_B + 1) + 1$$

$$A_R = [E] * (2 * I_B + A_B - M_B + 1) + M_B - 2$$

$$M_R = 1$$

## NUMERIC BIT PATTERNS

K

---

This appendix shows examples of the CHAR and MANT functional modifiers discussed in Section 2 and gives examples of numeric assignment and numeric exchange statements discussed in Section 4.

### CHAR EXAMPLES

```
ITEM FLOAT F P 123.456 $
ITEM INT I 60 S $
MONITOR FLOAT INT $
FLOAT = FLOAT $
INT = CHAR(FLOAT) $
CHAR(FLOAT) = CHAR(FLOAT) + 1 $
CHAR(FLOAT) = 15 $
CHAR(FLOAT) = -80 $
```

|                               | <u>Decimal</u>        | <u>Octal</u>              |
|-------------------------------|-----------------------|---------------------------|
| MONITORED REAL DATA<br>FLOAT  | = .12345600000000E+03 | = O(17267556457065176763) |
| MONITORED INTEGER<br>DATA INT | = -41                 | = O(777777777777777726)   |
| MONITORED REAL DATA<br>FLOAT  | = .24691200000000E+03 | = O(17277556457065176763) |
| MONITORED REAL DATA<br>FLOAT  | = .88959423295464E+19 | = O(20177556457065176763) |
| MONITORED REAL DATA<br>FLOAT  | = .22456515580416E-09 | = O(16577556457065176763) |

In the above examples, item FLOAT is set to a floating point value of 123.456 and item INT is set to a integer value of 60. In line 4, the contents of the floating-point variable (123.456) is assigned to the receiving variable FLOAT. In line 5, the integer item INT is set to the value representing the power of two by which the fractional part of FLOAT is multiplied. The biased value, which is  $1726_8$ , is converted to  $-51_8$  or  $-41_{10}$ .

Line 6 increments FLOAT by one. It raises the exponent by a power of two and doubles the value of the floating-point variable. In line 7, the CHAR of FLOAT is set to  $15_{10}$  ( $17_8$ ).

Thus, the exponent bits of FLOAT are set to  $2000_8$  plus  $17_8$  or a total of  $2017_8$ . Next, the CHAR of FLOAT is set to  $-80_{10}$  ( $-120_8$ ). The exponent bits are set to  $1657_8$ .

## MANT EXAMPLES

```

FLOAT = 123.456 $
INT = MANT(FLOAT) $
MANT(FLOAT) = MANT(FLOAT) + 100000000000 $
MANT(FLOAT) = 0(7556457065176763) $
MANT(FLOAT) = -1234567890123 $
INT = MANT(FLOAT) $

```

|                                 | <u>Decimal</u>      | <u>Octal</u>            |
|---------------------------------|---------------------|-------------------------|
| MONITORED REAL DATA =<br>FLOAT  | .12345600000000E+03 | O(17267556457065176763) |
| MONITORED INTEGER =<br>DATA INT | 271482615037427     | O(00007556457065176763) |
| MONITORED REAL DATA =<br>FLOAT  | .12350147473509E+03 | O(17267560030122762763) |
| MONITORED REAL DATA =<br>FLOAT  | .12345600000000E+03 | O(17267556457065176763) |
| MONITORED REAL DATA =<br>FLOAT  | .56141647752293E+00 | O(60517756021601175464) |
| MONITORED INTEGER =<br>DATA INT | -1234567890123      | O(77777756021601175464) |

In the above examples, FLOAT is assigned a preset to the floating-point value of 123.456. Next INT is set to the MANT of FLOAT. The decimal point is located to the right of the 48 bits which represent the mantissa of the floating-point variable. The integer variable has an identical bit pattern.

On line 3, the MANT of FLOAT is incremented. However, no change in the exponent bits is made as the result of the integer addition of the original value of MANT and the constant added to it is positive. Line 4 sets the MANT of FLOAT to the octal value it originally contained. Thus, the variable is returned to its original value.

Next, the MANT of FLOAT is set to a negative number. It causes the sign bit to be set and makes available the one's complement of the exponent. The mantissa bits are set to the octal constant of -1234567890123. On line 6, the integer variable, INT, is set to the MANT of FLOAT, and because it is negative, the value is sign extended.

## NUMERIC EXCHANGE STATEMENT EXAMPLES

### FLOATING-POINT AND INTEGER ITEM EXCHANGE

```
ITEM FLOAT F P 14.8 $
ITEM INT I 18 S P 5 $
INT == FLOAT $
```

In this example, a floating-point item FLOAT, with a value of 14.8, and an integer item INT, with a signed value of 5, will be exchanged. FLOAT will be set to a floating-point value of 5, and item INT will be set to an integer value of 14. The fractional portion of item FLOAT was lost in the exchange.

### FIXED-POINT SIMPLE ITEM EXCHANGE

```
ITEM BFIX A 7 U 4 P 4.3125 $
ITEM FIXD A 7 U 3 P 11.125 $
FIXD == BFIX $
```

In this example, two fixed-point items with differing integer and fractional bit precision are exchanged causing precision adjustment to be required for both variables. The items may be simple, array, or non-packed table items. Left-truncation will occur for simple items only when the word boundary is exceeded. The word containing item BFIX is set to a value which exceeds that specified in its specification. It was declared with only 3 integer bits which can represent a maximum of 7 bits, but the word containing it is now set to 11. This value would be used in any further computations. Listed below are the decimal values of the items, and the octal and binary values of the words before and after the exchange.

|        | <u>Item</u> | <u>Decimal</u> | <u>Octal</u> | <u>Bit Pattern</u> |
|--------|-------------|----------------|--------------|--------------------|
| Before | { BFIX      | 4.3125         | 105          | 00100.0101         |
|        | { FIXD      | 11.125         | 131          | 001011.001         |
| After  | { BFIX      | 11.125         | 262          | 01011.0010         |
|        | { FIXD      | 4.25           | 42           | 000100.010         |

### PACKED TABLE FIXED-POINT ITEM EXCHANGE

```
TABLE TAB V 5 1 $
BEGIN
ITEM FILL1 I 18 S 0 0 $ BEGIN 0 END
ITEM TBFIX A 7 U 4 0 18 $ BEGIN 4.3125 END
ITEM FILL2 I 20 S 0 25 $ BEGIN 0 END
ITEM TFIXD A 7 U 3 0 45 $ BEGIN 11.125 END
ITEM FILL3 I 7 S 0 52 $ BEGIN 0 END
END
TFIXD(0) == TBFIX(0)
```



In this example, two fixed-point items in a packed table (defined or dense) with differing integer and fractional bit precision are exchanged causing precision adjustment to be required for both variables. Left-truncation will occur when the word boundary of a item is exceeded. Listed below are the entry values (in octal), the decimal values of the items, and the octal and binary values of the words before and after the exchange.

|        | <u>Item</u> | <u>Decimal</u> | <u>Octal</u>            | <u>Bit Pattern</u> |
|--------|-------------|----------------|-------------------------|--------------------|
| Entry0 |             |                | 00000 10500 00013 10000 |                    |
| Before | { TBFIX     | 4.3125         | 105                     | 100.0101           |
|        | { TFIXD     | 11.125         | 131                     | 1011.001           |
| After  | { TBFIX     | 3.125          | 62                      | 011.0010           |
|        | { TFIXD     | 4.25           | 42                      | 0100.010           |
| Entry0 |             |                | 00000 06200 00004 20000 |                    |

---

Direct code assembly language is the subset of the COMPASS assembly language which can be introduced into the JOVIAL language with

DIRECT-JOVIAL brackets<sup>†</sup>

Certain assumptions are made by the compiler about direct code:

- Compiler-generated instructions remain unmodified
- No branches to regions coded in JOVIAL occur
- No registers or data are saved; as a result optimization is severely degraded

## COUNTERS

Counters are maintained by direct code to define the location of code and the current position within a word. There are three types of counters: origin, location, and position.

### ORIGIN

The origin counter is maintained by the JOVIAL compiler to indicate the location where instructions will be placed by the loader. It is incremented by one for each completed word of assembled data.

### LOCATION

The location counter has a value identical to that of the origin counter. It provides definition for location symbols.

### POSITION

This counter maintains a position within a 60-bit assembly word. As each code-generating instruction is encountered, the position counter is updated to reflect the next available bit position.

---

<sup>†</sup> See Section 4 for a description of the DIRECT-JOVIAL brackets.

## SOURCE STATEMENTS

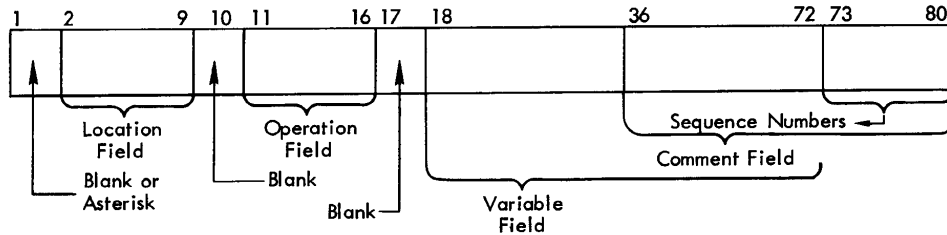
Direct code consists of a sequence of symbolic statements. Each statement contains a maximum of four fields in the order listed below:

- Location Field – Must begin in column 1 or 2
- Operation Field – May begin in columns 3 to 35
- Variable Field – Must begin before column 36
- Comment Field – Begins after termination of variable field, or after column 35 if variable field is empty. Comment may continue through column 80.

These fields are separated by one or more blanks. Blanks are always interpreted as field separators unless embedded in the comment field. Columns 73-80 may contain a comment, but generally are used for sequencing.

Each statement is either a comment or an instruction. Column 1 is used to distinguish a comment statement from an instruction statement. If it contains an asterisk, the remainder of the line is comment; any other character including the blank indicates the start of an instruction.

Examples of a standard format for source lines:



## COMMENT

A comment statement is introduced by an asterisk in column 1. The comment is printed in the output listing but does not generate any code. A comment may consist of any combination of characters extending from column 2 through 80.

## INSTRUCTION

The elements of an instruction correspond to the four fields of a statement which are:

- Location Field – Blank; or may contain one of the following:
  - +
  - 
  - symbol
- Operation Field – Required; may contain one of the following:
  - central processor operation code
  - pseudo instruction
- Variable Field – Depends on the operation code; for direct code machine instructions, the variable field is 1, 2, or 3 subfields separated by commas. Each subfield may contain register names separated by the operators + - \* /. These operators determine the octal value of the instruction and may not be substituted.
- Comment Field – Optional; may contain any combination of characters including the blank.

## SYMBOLS

A symbol is a string of 1 to 8 alphanumeric characters representing a value. The first character may be numeric. When a symbol is specified in the location field of a machine or pseudo instruction, it is assigned the current value of the location counter. When a relocatable symbol is specified, the value assigned is relative to the program's base address. A symbol in an assembly may not be An, Bn, or Xn, where n is a single digit between 0 and 7; these represent the registers.

All symbols used in direct code statements, except in the ASSIGN pseudo instruction, must be defined in that direct code block.

## REGISTERS

Register names are symbolic representations of the 24 operating registers of the computer. The names are predefined in direct code, and may not be redefined in the program. Registers are represented by An, Bn, or Xn, where n is a digit between 0 and 7. Any other value for n will cause the name to be interpreted as a symbol rather than a register name.

## ADDRESS EXPRESSIONS

An address expression may appear as a subfield of the variable field of an instruction statement. An address expression consists of one of the following:

- Symbol
- Symbol  $\pm$  integer-constant
- Integer-constant (optionally signed)

## FORCING UPPER

Assembled data is packed sequentially into a 60-bit word in bytes of 15, 30, or 60 bits. If there is not room in a partially filled 60-bit word for the instruction or data currently being evaluated, the remainder of that word is filled with 15-bit no-operation instructions ( $46000_8$ ), and the current instruction is assigned the first position in the next word.

Upper is also forced when any of the following occurs:

- A symbol or  $+$  appears in the location field of the current statement
- Current instruction is RE, WE, or XJ, unless there is a minus sign in the location field
- Current instruction is BSS or BSSZ

Forcing upper is automatic after JP, RJ, PS, and an EQ or ZR with a single address (the unconditional EQ or ZR). The ECS instructions WE and RE must appear in the upper 30 bits of an instruction; and when executed successfully, execution continued at the beginning of the next 60-bit word. The lower half of the WE or RE word presumably contains a jump to an error routine to be taken if the WE or RE is rejected.

Automatic forcing upper after JP, RJ, PS, EQ, and ZR can be negated by using a minus sign in the location field of the next instruction. When a minus sign appears, the current line will be assembled into the next position large enough to contain it.

## PSEUDO INSTRUCTIONS

Direct code provides two types of pseudo instructions: storage allocation instructions and JOVIAL directives.

## STORAGE ALLOCATION

There are two storage allocation pseudo instructions: BSS and BSSZ. They both cause adjustment of the location and origin counters, and both force upper.

### BSS

This instruction reserves an area of storage. The form is:

| <u>Location</u> | <u>Operation</u> | <u>Variable</u> |
|-----------------|------------------|-----------------|
| symbol or blank | BSS              | constant        |

A symbol in the location field is defined as the current value of the location counter. The location and origin counters are incremented by the value of the constant. The area between the symbol and the constant is reserved and is zero-filled. BSS 0 does not allocate storage, but it does force upper.

### BSSZ

BSSZ also reserves an area of zero-filled storage. The specification of this instruction is exactly like BSS and its effect is identical.

## JOVIAL DIRECTIVES

There are three pseudo instructions that direct the JOVIAL compiler: DIRECT, JOVIAL, and ASSIGN.

### DIRECT

Instructs the compiler that all lines up to and including the JOVIAL pseudo instruction are direct code. DIRECT causes full word alignment in the object code.

### JOVIAL

Instructs the direct code processor of the JOVIAL compiler that this is the last line of direct code. JOVIAL code resumes in column 1 of the next line. The remainder of the card is not examined.

## ASSIGN

Permits transfer of values between JOVIAL variables and direct code registers by loading from or storing into a simple non-indexed JOVIAL variable.

The following forms of ASSIGN are permitted; in each case, var is a simple JOVIAL variable and reg is a register specification. The \$ terminator must not be placed in column 72.

|                                         |                                                                                           |
|-----------------------------------------|-------------------------------------------------------------------------------------------|
| ASSIGN A(reg) = var \$                  | reg may be X1, X2, X3, X4, or X5                                                          |
| ASSIGN A( ) = var \$                    | the register is assumed to be X5                                                          |
| ASSIGN A(reg, i <sub>1</sub> ) = var \$ | reg may be X1, X2, X3, X4, or X5. i <sub>1</sub> is an optionally signed integer constant |
| ASSIGN A(i <sub>1</sub> ) = var \$      | register is assumed to be X5. i <sub>1</sub> is an optionally signed integer constant     |
| ASSIGN var = A(reg) \$                  | reg may be X6 or X7                                                                       |
| ASSIGN var = A( ) \$                    | the register is assumed to be X6                                                          |
| ASSIGN var = A(reg, i <sub>2</sub> ) \$ | reg may be X6 or X7. i <sub>2</sub> is an optionally signed integer constant              |
| ASSIGN var = A(i <sub>2</sub> ) \$      | the register is assumed to be X6. i <sub>2</sub> is an optionally signed integer constant |

The machine instructions generated by ASSIGN loads the A register which corresponds to the specified or assumed X register with the variable address.

If i<sub>1</sub> is specified and is not equal to zero, an algebraic left shift i<sub>1</sub> places (right shift if i<sub>1</sub> is negative) follows the load of the A register. If i<sub>2</sub> is specified and is not equal to zero, the A register load is preceded by an algebraic right shift i<sub>2</sub> places (left shift if i<sub>2</sub> is negative). Where i<sub>1</sub> or i<sub>2</sub> is not specified, the effect is as though the operation were on a floating-point value.

## **CENTRAL PROCESSOR OPERATION CODES**

The complete set of central processor operation codes for use with direct code are given in Table L-1.

Each operation is defined by listing the mnemonic, each subfield of the variable field, the octal representation, and the instruction length in bits.

Instructions are listed in the order of their octal value; an entry is given for the octal value of each permissible variable field format.

In the mnemonic (operation code) field and the variable field notations, the following symbology is used:

|                 |                                                                                          |
|-----------------|------------------------------------------------------------------------------------------|
| $X_i, X_j, X_k$ | X register symbols (the number of the register is placed in the $i, j,$ or $k$ portion). |
| $A_i, A_j, A_k$ | A register symbols                                                                       |
| $B_i, B_j, B_k$ | B register symbols                                                                       |
| K               | Address expression (18 bits)                                                             |
| n               | Integer-constant (96 bits)                                                               |

TABLE L-1. CENTRAL PROCESSOR OPERATION CODES

| Length | Mnemonic | Variable Field | Octal                   |
|--------|----------|----------------|-------------------------|
| 30     | PS       |                | 0000 000000             |
| 30     | RJ       | K              | 0100 K                  |
| 30     | RE       | $B_j+K$        | 011j K                  |
| 30     | WE       | $B_j+K$        | 012j K                  |
| 60     | XJ       |                | 0130 000000 46000 46000 |
| 30     | JP       | K              | 0200 K                  |
| 30     | JP       | $B_j+K$        | 020j K                  |
| 30     | ZR       | $X_j, K$       | 030j K                  |
| 30     | NZ       | $X_j, K$       | 031j K                  |
| 30     | PL       | $X_j, K$       | 032j K                  |
| 30     | NG       | $X_j, K$       | 033j K                  |
| 30     | IR       | $X_j, K$       | 034j K                  |
| 30     | OR       | $X_j, K$       | 035j K                  |
| 30     | DF       | $X_j, K$       | 036j K                  |
| 30     | ID       | $X_j, K$       | 037j K                  |
| 30     | ZR       | K              | 0400 K                  |
| 30     | EQ       | K              | 0400 K                  |
| 30     | EQ       | $B_i, K$       | 04i0 K                  |
| 30     | ZR       | $B_i, K$       | 04i0 K                  |
| 30     | EQ       | $B_i, B_j, K$  | 04ij K                  |
| 30     | NZ       | $B_i, K$       | 05i0 K                  |
| 30     | NE       | $B_i, K$       | 05i0 K                  |

(Continued)



TABLE L-1. CENTRAL PROCESSOR OPERATION CODES (Cont'd)

| Length | Mnemonic | Variable Field           | Octal   |
|--------|----------|--------------------------|---------|
| 30     | NE       | $B_i, B_j, K$            | 05ij K  |
| 30     | PL       | $B_i, K$                 | 06i0 K  |
| 30     | GE       | $B_i, K$                 | 06i0 K  |
| 30     | GE       | $B_i, B_j, K$            | 06ij K  |
| 30     | LE       | $B_j, K$                 | 060j K  |
| 30     | LE       | $B_j, B_i, K$            | 06ij K  |
| 30     | NG       | $B_i, K$                 | 07i0 K  |
| 30     | LT       | $B_i, K$                 | 07i0 K  |
| 30     | LT       | $B_i, B_j, K$            | 071ij K |
| 30     | GT       | $B_j, K$                 | 070j K  |
| 30     | GT       | $B_j, B_i, K$            | 07ij K  |
| 15     | BXi      | $X_j$                    | 10ijj   |
| 15     | BXi      | $X_j * X_k$              | 11ijk   |
| 15     | BXi      | $X_j + X_k$              | 12ijk   |
| 15     | BXi      | $X_j - X_k$              | 13ijk   |
| 15     | BXi      | $-X_k$                   | 14ikk   |
| 15     | BXi      | $-X_k * X_j$             | 15ijk   |
| 15     | BXi      | $-X_k + X_j$             | 16ijk   |
| 15     | BXi      | $-X_k - X_j$             | 17ijk   |
| 15     | LXi      | n                        | 20i n   |
| 15     | AXi      | n                        | 21i n   |
| 15     | LXi      | $X_k$                    | 22i0k   |
| 15     | LXi      | $B_j, X_k$ or $X_k, B_j$ | 22ijk   |
| 15     | AXi      | $B_j, X_k$ or $X_k, B_j$ | 23ijk   |
| 15     | NXi      | $X_k$                    | 24i0k   |
| 15     | NXi      | $B_j, X_k$ or $X_k, B_j$ | 24ijk   |
| 15     | ZXi      | $X_k$                    | 25i0k   |
| 15     | ZXi      | $B_j, X_k$ or $X_k, B_j$ | 25ijk   |
| 15     | UXi      | $X_k$                    | 26i0k   |
| 15     | UXi      | $B_j, X_k$ or $X_k, B_j$ | 26ijk   |
| 15     | PXi      | $B_j, X_k$ or $X_k, B_j$ | 27ijk   |
| 15     | FXi      | $X_j + X_k$              | 30ijk   |
| 15     | FXi      | $X_j - X_k$              | 31ijk   |
| 15     | DXi      | $X_j + X_k$              | 32ijk   |
| 15     | DXi      | $X_j - X_k$              | 33ijk   |

(Continued)

TABLE L-1. CENTRAL PROCESSOR OPERATION CODES (Cont'd)

| Length | Mnemonic | Variable Field              | Octal  |
|--------|----------|-----------------------------|--------|
| 15     | RXi      | $x_j + X_k$                 | 34ijk  |
| 15     | RXi      | $X_j - X_k$                 | 35ijk  |
| 15     | IXi      | $X_j + X_k$                 | 36ijk  |
| 15     | IXi      | $X_j - X_k$                 | 37ijk  |
| 15     | FXi      | $X_j * X_k$                 | 40ijk  |
| 15     | RXi      | $X_j * X_k$                 | 41ijk  |
| 15     | DXi      | $X_j * X_k$                 | 42ijk  |
| 15     | MXi      | n                           | 43i n  |
| 15     | FXi      | $X_j / X_k$                 | 44ijk  |
| 15     | RXi      | $X_j / X_k$                 | 45ijk  |
| 15     | NO       |                             | 46000  |
| 15     | CXi      | $X_k$                       | 47ikk  |
| 30     | SAi      | $A_j + K$                   | 50ij K |
| 30     | SAi      | K                           | 51i0 K |
| 30     | SAi      | $B_j + K$                   | 51ij K |
| 30     | SAi      | $X_j + K$                   | 52ij K |
| 15     | SAi      | $X_j$                       | 53ij0  |
| 15     | SAi      | $X_j + B_k$ or $B_k + X_j$  | 53ijk  |
| 15     | SAi      | $A_j$                       | 54ij0  |
| 15     | SAi      | $A_j + B_k$ or $B_k + A_j$  | 54ijk  |
| 15     | SAi      | $A_j - B_j$ or $-B_j + A_j$ | 55ijk  |
| 15     | SAi      | $B_j$                       | 56ij0  |
| 15     | SAi      | $B_j + B_k$                 | 56ijk  |
| 15     | SAi      | $-B_k$                      | 57i0k  |
| 15     | SAi      | $B_j - B_k$ or $-B_k + B_j$ | 57ijk  |
| 30     | SBi      | $A_j + K$                   | 60ij K |
| 30     | SBi      | K                           | 61i0 K |
| 30     | SBi      | $B_j + K$                   | 61ij K |
| 30     | SBi      | $X_j + K$                   | 62ij K |
| 15     | SBi      | $X_j$                       | 63ij0  |
| 15     | SBi      | $X_j + B_k$ or $B_k + X_j$  | 63ijk  |
| 15     | SBi      | $A_j$                       | 64ij0  |
| 15     | SBi      | $A_j + B_k$ or $B_k + A_j$  | 64ijk  |
| 15     | SBi      | $A_j - B_k$ or $-B_k + A_j$ | 65ijk  |

(Continued)

TABLE L-1. CENTRAL PROCESSOR OPERATION CODES (Cont'd)

| Length | Mnemonic | Variable Field  | Octal  |
|--------|----------|-----------------|--------|
| 15     | SBi      | Bj              | 66ij0  |
| 15     | SBi      | Bj+Bk           | 66ijk  |
| 15     | SBi      | -Bk             | 67i0k  |
| 15     | SBi      | Bj-Bk or -Bk+Bj | 67ijk  |
| 30     | SXi      | Aj+K            | 70ij K |
| 30     | SXi      | K               | 71i0 K |
| 30     | SXi      | Bj+K            | 71ij K |
| 30     | SXi      | Xj+K            | 72ij K |
| 15     | SXi      | Xj              | 73ij0  |
| 15     | SXi      | Xj+Bk or Bk+Xj  | 73ijk  |
| 15     | SXi      | Aj              | 74ij0  |
| 15     | SXi      | Aj+Bk or Bk+Aj  | 74ijk  |
| 15     | SXi      | Aj-Bk or -Bk+Aj | 75ijk  |
| 15     | SXi      | Bj              | 76ij0  |
| 15     | SXi      | Bj+Bk           | 76ijk  |
| 15     | SXi      | -Bk             | 76i0k  |
| 15     | SXi      | Bj-Bk or -Bk+Bj | 77ijk  |

---

Programs that exceed available memory may be divided into independent parts which may be called and executed as needed. Such programs can be divided into segments and overlays.

Segments are groups of subprograms that are loaded in relocatable form when requested, giving the user the explicit control over established interprogram links. An overlay is a program combined with its subprograms which is converted to absolute form and written on mass storage prior to execution. During execution, overlays are called into memory and executed as requested.

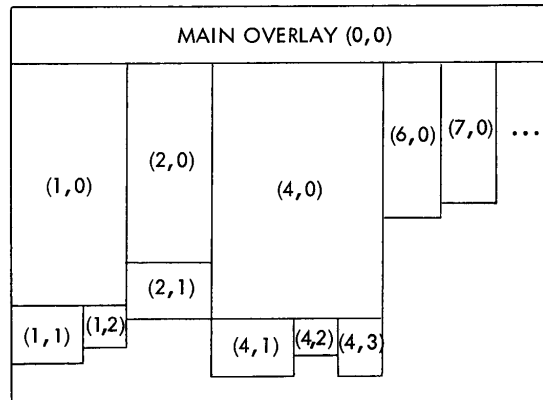
## OVERLAYS

Overlay processing allows programs to be divided into independent parts which may be called and executed as needed. Each part (overlay) must consist of a single main program and any necessary subprograms.

Each overlay is numbered with an ordered pair of numbers (I, J), each in the range 0-77<sub>8</sub>. I denotes the primary level, J, the secondary level. An overlay with a nonzero secondary level is called a secondary overlay. It is associated with and subordinate to the overlay which has the same primary level and a zero secondary level, called the primary overlay. The initial or main overlay which always remains in memory has levels (0, 0). The significance of this distinction appears in the order in which overlays are loaded.

Overlay level numbers (0, 1), (0, 2), (0, 3) and so forth, are illegal. Primary overlays all have their origin at the same point immediately following the main overlay (0, 0). The origin of secondary overlays immediately follows the primary overlay. For any given program execution, all overlay identifiers must be unique. The loading of any primary overlay destroys any other primary overlay. For this reason, no primary overlay may load other primary overlays. Secondary overlays may be loaded only by the associated primary overlay or main overlay; thus, two levels of overlays are available to the programmer. An overlay may reference subprograms in its own overlay, in the main overlay, or in its associated primary overlay.

Example:



Overlays (1, 1) and (1, 2) are secondary to overlay (1, 0)

Overlay (2, 1) is secondary to overlay (2, 0)

Overlay (2, 1) may not be called from (1, 0) or (1, 1) or (1, 2) but only from (2, 0) or (0, 0)

Overlays (1, 0), (2, 0), (4, 0) etc., may be called only from the main overlay (0, 0)

### CREATING JOVIAL OVERLAYS

JOVIAL overlays are created by one or subsequent overlay control cards that are inserted prior to the START card in the program. Each overlay control card must begin in column one and be of the following form:

```
OVERLAY (lfn, l1, l2, Cnnnnnn)
```

where:

lfn = The file name on which overlay is to be written. The first overlay card must have a named lfn. Subsequent cards may omit it, indicating that the overlays are related and are to be written in the same lfn. A different lfn on subsequent cards results in generation of overlays to the new lfn.

l1 = The primary level number in octal.

12 = The secondary level number in octal. 11, 12 for the first overlay card must be 0, 0.

Cnnnnn = An optional parameter consisting of the letter C and a six-digit octal number. If this parameter is present, the overlay is loaded nnnnnn words from the start of blank common. This provides a method of changing the size of blank common at execution time. Cnnnnn cannot be included on the overlay 0, 0 loader directive. If this parameter is omitted, the overlay is loaded in the normal manner.

When the compiler recognizes the card or cards they are transferred directly into the binary output file. No error checking is performed during the transfer.

When overlays other than the main overlay (0, 0) are compiled, the overlay transfer parameter, E, must be specified on the JOVIAL control card, as described in Section 10. When compiling only the main overlay (0, 0), any normal JOVIAL control card may be used.

#### **LOADING OVERLAYS**

The primary JOVIAL program overlay must be loaded by a SCOPE control card, as would any JOVIAL program to be executed.

Secondary JOVIAL overlays (as specified on the overlay control cards) are loaded and executed by calling the library procedure OVRLOD. The calling sequence and execution are described on page G-7.

#### **PROGRAM OVERLAY EXAMPLES**

This section of Appendix M contains two examples of program overlays. The complete listings were too long to be shown here, therefore, only the source input and a portion of the load map are printed for each example.

##### **EXAMPLE 1**

This example installs the utility program SEQD shown on page I-5, which is used to punch a sequentially numbered deck of cards. Once SEQD is installed on the system, it may be used by any user.

```

SEQD,CM70000,T100. } Entry Point Name SEQD
JOVIAL{N=SEQD}
MAP{PART}
LOAD{LGO}
NOGO.
REWIND{SEQD}
EDITLIB.
RETURN{SEQD}
SEQD.
789
OVERLAY{SEQD,0,0} ← Overlay Control Card

```

(SEQD Program as on page I-5)

```

789
READY{SYSTEM}
RPADD{* ,SEQD}
COMPLETE.
789
START * ≠ TEST OF INTEGER TRUNCATION FOR INDEX SWITCH ≠
MODE F *
MONITOR ZERO. , ONE. ,TWO. *
FLT = 0.1 *
GOTO NUMB{ * FLT + 1.1 * } *
ZERO.
STOP *
ONE.
STOP *
TWO.
STOP *
SWITCH NUMB = {ZERO,ONE,TWO} *
TERM *
6789

```

SEQD Partial Load Map

```

CORE MAP 14.20.58. OVERLAY 00.00 CONTROL
--TIME--LOAD MODE --L1-L2--TYPE--
FMA LOADER 063756 FMA TABLES 061370
-PROGRAM-----ADDRESS-
SEQD 000101
SYSTEM$ 001202
JOVSM$ 002205
JOVIO$ 002430
BACKSP$ 003023
OUTPTB$ 003350
REWIN$ 003622
SIO$ 003672
INPUTB$ 005305
GETBA 005574
----UNSATISFIED EXTERNALS-----
--Labeled---COMMON--
--USER--CALL--
000100 005613 000000 000000
-----FMA LOAD--LMA LOAD--BLNK COMM--LENGTH--

```

REFERENCES

LIST OF PROGRAMS REPLACED

```

SEQD DELETED } EDITLIB Output
SEQD ADDED }

```

Dayfile

```

12/22/71 **SCOPE**3.3 PRR284 11/30/71
14.20.46.SEQD04H
14.20.46.SEQD.CM70000.T100.
14.20.46.JOVIAL(N=SEQD)
14.20.55.MAP(PART)
14.20.55.LOAD(LGO)
14.20.59.NOGO.
14.20.59.REMIND(SEQD)
14.20.59.EDITLIB.
14.21.01. READY(SYSTEM)
14.21.01. RPA00(+,SEQD)
14.21.03. COMPLETE.
14.21.13.GO.
14.21.15.RETURN(SEQD) → Return local file SEQD
14.21.15.SEQD. → Execute SEQD on system
14.21.15.STOP → End of execution
14.21.16.CP 010.879 SEC.
14.21.16.PP 009.625 SEC.
14.21.16.IO 002.104 SEC.

```



EXAMPLE 2

This example demonstrates the use of a main overlay with two second level overlays. The second level overlays are loaded by a call to the library routine OVRLOD. The flow of control may be traced by the printouts from the overlays indicating execution.

```

OVL D,CM70000,T77.
JOVIAL(N=OVER,B=HOLD,M)
JOVIAL(N=OVER,B=HOLD,M,E)
MAP{PART}
LOAD{HOLD}
NOGO.
OVER.
789
 } Control Cards

OVERLAY{OVER,0,0} ← Overlay Control Card
START $
 ARRAY LIT 3 H 10 $
 BEGIN 10H{ OVERLAY 0} 10H{,0 IS IN 0}
 10H{PERATION } END
 PRINT{6H{{3A10}}} $
 FOR A = 0,1,2 $
 LIST{LIT{#A#}} $
 ENDL $
 OVRLOD {10H{OVER },1,0} $
 PRINT{6H{{3A10}}} $
 FOR A = 0,1,2 $
 LIST{LIT{#A#}} $
 ENDL $
 OVRLOD {10H{OVER },2,0} $
 PRINT{6H{{3A10}}} $
 FOR A = 0,1,2 $
 LIST{LIT{#A#}} $
 ENDL $
TERM $
789
OVERLAY{OVER,1,0}
START $
 ARRAY LIT 3 H 10 $
 BEGIN 10H{ OVERLAY 1} 10H{,0 IS IN 0}
 10H{PERATION } END
 PRINT{6H{{3A10}}} $
 FOR A = 0,1,2 $
 LIST{LIT{#A#}} $
 ENDL $
TERM $
OVERLAY{OVER,2,0}
START $
 ARRAY LIT 3 H 10 $
 BEGIN 10H{ OVERLAY 2} 10H{,0 IS IN 0}
 10H{PERATION } END
 PRINT{6H{{3A10}}} $
 FOR A = 0,1,2 $
 LIST{LIT{#A#}} $
 ENDL $
TERM $
6789
 }

```

Main Overlay; compiled by:  
 JOVIAL(N=OVER, B=HOLD, M)  
 N=OVER - Entry point OVER  
 B=HOLD - Binary to file HOLD  
 M - Monitor option specified

Secondary Overlays; compiled by:  
 JOVIAL(N=OVER, B=HOLD, M, E)  
 N=OVER - Entry point OVER  
 B=HOLD - Binary to file HOLD  
 M - Monitor option specified  
 E - Overlay transfer parameter specified

Overlay (0,0) Partial Load Map

```

CORE MAP 13.40.22. OVERLAY 00.00 CONTROL 000100 005022 000000 000000
---TIME---LOAD MODE --L1--L2-----TYPE-----
FMA LOADER 063742 FMA TABLES 061500 -----FMA LOAD--LMA LOAD--BLNK COMN--LENGTH--
-PROGRAM-----ADDRESS-
OVER 000101
SYSTEM$ 000423
SYNIO 001426
OVERL$ 001466
OVERLOD 001610
OUTPTC$ 001656
SIO$ 001752
GETBA 003365
KODER$ 003404
-----UNSATISFIED EXTERNALS-----
REFERENCES
--LABELED---COMMON--
USER-----CALL-----

```

Overlay (1,0) Partial Load Map

```

CORE MAP 13.40.24. OVERLAY 01.00 CONTROL 005022 005055 000000 000000
---TIME---LOAD MODE --L1--L2-----TYPE-----
FMA LOADER 063742 FMA TABLES 061453 -----FMA LOAD--LMA LOAD--BLNK COMN--LENGTH--
-PROGRAM-----ADDRESS-
OVER 005023
-----UNSATISFIED EXTERNALS-----
REFERENCES
--LABELED---COMMON--
USER-----CALL-----

```

Overlay (2,0) Partial Load Map

```

CORE MAP 13.40.26. OVERLAY 02.00 CONTROL 005022 005055 000000 000000
---TIME---LOAD MODE --L1--L2-----TYPE-----
FMA LOADER 063742 FMA TABLES 061453 -----FMA LOAD--LMA LOAD--BLNK COMN--LENGTH--
-PROGRAM-----ADDRESS-
OVER 005023
-----UNSATISFIED EXTERNALS-----
OVERLAY 0,0 IS IN OPERATION
OVERLAY 1,0 IS IN OPERATION
OVERLAY 0,0 IS IN OPERATION
OVERLAY 2,0 IS IN OPERATION
PROGRAM OUTPUT
REFERENCES
--LABELED---COMMON--
USER-----CALL-----

```

Dayfile

12/17/71 \*\*SCOPE\*\*3.3 PSR20E 12/10/71  
13.39.30.OVL001H  
13.39.30.OVLD,CM7000,T77. 26  
13.39.30.  
13.39.30.JOVIAL(N=OVER,B=HOLD,M)  
13.39.48.JOVIAL(N=OVER,B=HOLD,M,E)  
13.40.14.MAP(PART)  
13.40.14.LOAD(HCLD)  
13.40.26.NOGO.  
13.40.27.OVER.  
13.40.29.END OVER  
13.40.29.CP 002.903 SEC.  
13.40.29.PP 026.631 SEC.  
13.40.29.IC 005.223 SEC.

## SEGMENTS

A segment is a group of subprograms (possibly only one) which are loaded together when specified by the programmer. Segments are loaded at levels from 0 to 77<sub>8</sub>. Level zero is reserved for the initial or main segment. Level zero, which must contain a program, remains in memory during execution.

The following definitions apply to segmentation:

- **Entry Point** — A named location within a subprogram that can be referenced by another program; created by a program or subprogram. The entry point for a program is the name specified by the N=NAME on the JOVIAL control card; if none is specified, the default is MAINPG. The entry point for subprograms is the same as the subprogram name (see page 8-6).
- **External reference** — A reference within a program or subprogram to the entry point of some other subprogram; created by explicit call statements, function references, I/O statements, implicit functions, etc.
- **Link** — The connection established between an external reference and an entry point when the programs are loaded into memory.
- **Unsatisfied external** — An external reference for which no matching entry point can be found, and therefore no link established.

When the segment is loaded, external references will be linked to entry points in previously loaded segments (those at a lower level). Similarly, entry points in the segment are linked to unsatisfied external references in previously loaded segments. Unsatisfied external references in the segment remain unsatisfied; subsequent segment loading may include entry points to satisfy the external references. Unsatisfied external references may be satisfied, if possible, from the system library.

If a segment is to be loaded at a requested level which is less than or equal to the level of the last loaded segment, all segments at levels down to and including the requested level are removed or delinked. Delinking a segment at a given level requires that the linkage of external references in lower levels to entry points in the delinked segment be destroyed so that the external references are again unsatisfied.

Once the delinking is complete, the segment is loaded. Only one occurrence of a given subprogram or entry point is necessary since all levels may eventually link to the subprogram. However, a user may force loading of a subprogram by explicitly naming it in another segment at a higher level. Thereafter, all external references in higher levels are linked to

the new version. In this manner, a subprogram and/or entry point can effectively replace an identical one already loaded at a lower level. However, once a linkage is established, it is not destroyed unless the segment containing the entry point is removed.

Example:

The SINE routine is loaded in a segment at level 1. The user wishes to try an experimental version of SINE. He loads a segment containing the new SINE at level 2. Segments loaded at level 3 or higher will now be linked to SINE at level 2 until a new level 2 or a new SINE is loaded.

Common blocks may be loaded with any segment. Labeled common may not be cross-referenced in segments. Maximum blank common length is established in the first segment which makes use of blank common.

#### CREATING JOVIAL SEGMENTS

JOVIAL segments are created by segment control cards that are inserted prior to the START card in the program. Each control card must begin in column one. The segment control cards are SECTION, SEGZERO, and SEGMENT.

The SECTION control card defines a section within a segment. Segments are loaded by user calls during execution or by MTR during initial load. This card has the form:

```
SECTION (sname, pn1, pn2, ..., pnn)
```

where:

sname = Name of section (seven-character maximum).

pn<sub>1</sub> = Names of subprograms in the section. If more than one card is necessary to define a section, additional cards with the same sname may follow consecutively.

All subprograms within a section are loaded whenever the named section is loaded. All section cards must appear prior to the SEGMENT cards which refer to the named sections.

A SEGZERO card is always required prior to the binary text for all programs requiring segmentation loading. This card has the form:

```
SEGZERO (sn, pn1, pn2, . . . , pnn)
```

where:

sn = Segment name.

pn<sub>1</sub> = Names of subprograms or section names which make up main or zero level segments. Defining other segments in a similar manner reduces the list of subprograms in the loader call.

To define an entire segment, the following control card is used:

```
SEGMENT (sn, pn1, pn2, . . . , pnn)
```

The parameters are defined as in SEGZERO. In a segment, all programs must reside on the same file. A segment defined in the user's program need not be defined by a SEGMENT card; however, a SEGZERO card is always required.

When the compiler recognizes the card or cards, they are transferred directly into the binary output file. No error checking is performed during the transfer.

#### LOADING SEGMENTS

JOVIAL segments (specified on the control cards) are loaded by calling the library procedure SEGLD routine. The calling sequence and execution are described on page G-9. Once the named segments or subprograms are loaded, control returns to the statement following the call to SEGLD. The programmer is free to call on the loaded subprogram as desired.

## JOVIAL/INTERCOM INTERFACE

N

---

When a program is entered at an INTERCOM control point, INTERCOM associates INPUT and OUTPUT files of the program with the user's remote terminal device, and all references to these files are directed to the terminal. With calls to subprograms, the user may specify other files to be associated with the terminal.

When a program is executed at an INTERCOM control point after being compiled in batch mode, INPUT and OUTPUT files are not automatically associated with the user's terminal. The user must associate the files with the terminal using the statement:

```
CONNEC (lfn) $
```

where:

lfn = Any full word item with the device name for the file. The word is left-justified and the remainder of the word is zero-filled.

The CONNEC statement can also be used to associate any logical files in the user's program with the terminal.

If a file is already connected, the request is ignored. If the file has been used already, but not connected, this request will clear the file's buffer, write an end-of-file, and backspace over it before the connection is performed.

A file is disconnected by:

```
DISCON (lfn) $
```

where:

lfn = The same as CONNEC.

This request will be ignored if the file is not connected. After a disconnect, the file is re-associated with its former device.

Any files defined within the program or in a COMPOOL used during compilation may be connected or disconnected during program execution. An attempt to connect or disconnect an undefined file will result in a fatal execution time error, and the job is terminated.

CONNEC and DISCON calls are ignored when programs are not executed through an INTERCOM control point.

Example:

```
ITEM LFNA H 10 P {0611114050100000000000} #
"OCTAL VALUE OF FILEA"
```

In this program example, item LFNA has a file name of FILEA (octal value) which is left-justified, and the remainder of the word is zero-filled. FILEA may be associated with the user's remote terminal by the statement:

```
CONNEC (LFNA) $
```

or disconnected by

```
DISCON (LFNA) $
```



## INDEX

---

- ABS: 2-4
- Address Expressions: L-4
- ALL: 2-4
- AND: 3-7
- Arithmetic Operators: 2-2
- Arrays: 5-10, 6-10
  - Data Forms: 6-10
  - Declaration: 6-11
  - Constant List: 5-12
  - Referencing: 5-16
- ASSIGN: 2-4, 4-27
- Assignment Operator: 2-4
- Binary Output
  - Control Card Parameter: 10-2
- BIT: 2-4, 2-20
- Bit Patterns, Numeric: K-1
  - CHAR: K-1
  - Exchange Statements: K-3
  - MANT: K-2
- Brackets: 2-5
- Boolean Constant: 2-14
  - Assignment Statement: 4-9
  - Data: 2-9
  - Declarations: 5-7
  - Exchange Statement: 4-11
  - Logical Formula: 3-7
  - Relational Formula: 3-4, 3-5, 3-6
  - Variable: 2-25
- Buffer Size
  - File Declaration: 6-4
- BYTE: 2-4, 2-23
- Calling Sequence: E-1
- Carriage Control
  - JOVIAL Files: 6-2
- CHAR: 2-4, 2-22
  - Examples: K-1
- Characters: 2-1
  - Basic Elements: 2-1
  - Sets: F-1
- Clause
  - FOR: 4-16
  - FOR ALL: 4-23
  - IF: 4-13
- Closed Forms: 7-6
  - CLOSE: 7-6
  - Exit: 7-17
  - Functions: 7-16
  - Procedures: 7-8
- Codes, CPU Operation: L-6
- Comments
  - JOVIAL: 3-10
  - Direct Code: L-2
- Common Declaration: 8-2
  - COMMON: 8-2
- COMPOOL: 8-1
  - Assembly Parameter: 10-4
  - Common Declaration: 8-2
  - Control Card Parameter: 10-2
  - Creation: 8-3
  - Data Declaration: 8-1
  - Examples: 8-6
  - Program Structure: 8-6
  - Reference: 8-4, 8-5
  - Specification: 8-1
  - Subprogram Declaration: 8-2
  - Subprogram Reference: 8-5
- Constant List
  - Arrays: 5-12
  - Tables: 5-35

Constants: 2-9  
     Boolean: 2-14  
     Literal: 2-13  
     Numeric: 2-10  
     Status: 2-14

Control Card: 6-19  
     Execution: 6-19  
     JOVIAL: 10-1  
     Parameters: 10-1

Counters: L-1  
     Location: L-1  
     Position: L-1  
     Origin: L-1

Data Allocation: 5-41  
     Overlays: 5-41

Data Forms: 6-9  
     Arrays: 6-10  
     Examples: 6-11  
     Parallel Items: 6-10  
     Simple Items: 6-10  
     Table Entries: 6-11  
     Table Items: 6-10  
     Tables: 6-11  
     Variable Items: 6-10

Data Type Properties: 2-7  
     Boolean: 2-9  
     Literal: 2-9  
     Numeric: 2-8  
     Status: 2-9

Debugging Aids: 9-1  
     Monitor Statement: 9-1  
     Run-Time Error Monitor: 9-8

Decks, Sample: B-1

Declarations, Data: 5-1, 8-1  
     Array: 5-10  
     Common: 8-2  
     COMPOOL: 8-1  
     Files: 6-2, 6-3, 6-4, 6-5  
     Implicit Items: 5-7  
     Index Switch: 7-1  
     Item: 5-1  
     Item Switch: 7-3  
     Mode: 5-8  
     Overlay: 5-41  
     Procedures: 7-8  
     Subprogram: 8-2  
     Tables: 5-17, 5-25

Declarators: 2-6

DEFINE Directives: 3-10

Define Items: 5-32

Define Table: 5-18, 5-30  
     Define Item: 5-32  
     String Item: 5-33

Dense Packing: 5-22

Descriptors: 2-6

DIRECT: 2-5, 4-27

Directives: 2-6  
     DEFINE: 3-10  
     JOVIAL: L-5

Diagnostic Messages: A-1

Direct Code: L-1  
     Address Expression: L-4  
     CPU Operation Codes: L-6  
     Counters: L-1  
     Forcing Upper: L-4  
     JOVIAL Directives: L-5  
     Pseudo Instructions: L-4  
     Registers: L-3  
     Source Statements: L-2  
     Symbols: L-3

Elements, Basic: 1-2, 2-1  
     Characters: 2-1  
     Symbols: 2-1

END: 2-4, 5-40

ENDL: 6-19

ENTRY: 2-4, 5-40

Entry  
     Assignment Statement: 4-9  
     Exchange Statement: 4-12  
     Relational Formula: 3-6  
     Variable: 2-26

Error Messages: A-1  
     Source Diagnostic: A-1  
     Terminate Messages: A-13

Error Tracing: E-1

Exit From Closed Forms: 7-17

## Files

Declaration: 6-2.2  
Input/Output: 6-2.1  
I/O Statement: 6-11, 6-12, 6-14  
JOVIAL: 6-1  
Operators: 2-3  
Organization: 6-8  
Positioning: 6-7  
Status: 6-5

## Fixed-Point

Constant: 2-13  
Declaration: 5-2  
Exchange Example: K-3  
Numeric Data: 2-8  
Scaling: J-1, J-2, J-3

## Floating-Point

Constant: 2-12  
Declaration: 5-4  
Exchange Examples: K-3  
Numeric Data: 2-8, 2-9

FOR: 2-3, 4-17

Forcing Upper: L-4

FOR Clause: 4-16  
FOR ALL: 4-32

FORTRAN Formatted Output: 6-19  
Examples: 6-21

FOR Statement: 4-16  
Nested: 4-22  
One-Component: 4-18  
Parallel: 4-21  
Three-Component: 4-20  
Two-Component: 4-19

## Formulas: 3-1

Boolean: 3-4  
Comments: 3-10  
DEFINE: 3-10  
Functions: 3-9  
Literal: 3-3  
Logical: 3-7  
Numeric: 3-1  
Sequence of Evaluation: 3-8

## Functional Modifiers: 2-4

BIT: 2-20  
BYTE: 2-23  
CHAR: 2-22  
Constant: 2-11  
LOC: 2-11  
MANT: 2-22  
NENT: 2-11, 2-21  
NWDSN: 2-11

ODD: 2-25  
POS: 2-19  
Variable: 2-18

Functions: 7-16  
Call: 7-16  
Formulas: 3-9

Global Scope: 2-27

GOTO: 2-3, 4-12, 7-2

## Hints

Compiler Usage: D-1

IF: 2-3, 4-13

IFEITH: 2-3, 4-14

Index Declaration: 7-1  
Switch Call: 7-2

INPUT: 6-12

Input/Output: 6-1  
Execution: 6-19  
File Declaration: 6-2.2  
File Positioning: 6-7  
FORTRAN Formatted Output: 6-19  
JOVIAL Files: 6-1  
Organization of Data: 6-8  
Short Forms: 6-16  
Statements: 6-11, 6-13, 6-15

## Instruction

Codes: L-6  
Source Statement: L-2, L-3

INTERCOM Interface: N-1

## Integer: 2-8

Constants: 2-10  
Decimal: 2-10  
Declarations: 5-1  
Exchange Example: K-3  
Functional Modifier: 2-11  
Numeric Data: 2-8  
Octal: 2-10

Introduction: 1-1

Item Declarations: 5-1  
Boolean: 5-7

Descriptions: 5-9  
 Fixed-Point: 5-2  
 Floating-Point: 5-4  
 Integer: 5-1  
 Literal: 5-5  
 Status: 5-6  
 Switch: 7-3

JOVIAL: 2-5, 4-27  
 Card Parameters: 10-1  
 Control Card: 10-1  
 Defined Symbols: 2-2  
 Directives: L-5  
 Files: 6-7  
 Primitives: 2-15  
 Pseudo Instruction: 4-27

Language: 1-1  
 Like Table: 5-18, 5-34  
 Limitation: C-1  
 LIST: 6-19

List  
 Control Card Parameters: 10-2

Listings, Sample: H-1

Literal  
 Assignment Statement: 4-6  
 Constant: 2-13  
 Data: 2-9  
 Declarations: 5-5  
 Exchange Statement: 4-10  
 Formulas: 3-4  
 LOC: 2-4, 2-11  
 Variable: 2-23

Local Scope: 2-27

Logical Operators: 2-3

MANT: 2-4, 2-22  
 Examples: K-2

Messages: A-1

Mode Declaration: 5-8

Modifiers  
 Functional: 2-4

MONITOR: 9-1  
 Examples: 9-5

Monitor  
 Control Card Parameter: 10-3

Named Variables: 2-17

Names: 2-14  
 Loop Variables: 2-16  
 Prime-Prefixed Primitives: 2-15  
 Primitives: 2-15

NENT: 2-4, 2-11, 2-21, 5-40

No Packing: 5-21

NOT: 3-7

Notation: 1-3  
 Shorthand: 3-8

Numeric Constants: 2-10  
 Data: 2-8  
 Variable: 2-19

Numeric Formulas: 3-1  
 Mode of Results: 3-3  
 Sequence of Operation: 3-2

Numeric Statements  
 Assignment: 4-3  
 Bit Patterns: K-1  
 Exchange: 4-9

NWDSSEN: 2-4, 2-11, 5-40

ODD: 2-4, 2-25

OPEN INPUT: 2-4, 6-12

OPEN OUTPUT: 2-4, 6-12

Operation Codes, CPU: L-6

Operators  
 Arithmetic: 2-2  
 Assignment: 2-4  
 File: 2-3  
 Logical: 2-3  
 Relational: 2-2  
 Sequential: 2-3

Optimization  
 Control Card Parameter: 10-3

Optional Prime  
Control Card Parameter: 10-4  
OR: 3-7  
Ordinary Table: 5-18, 5-27  
ORIF: 2-3, 4-14  
OUTPUT: 6-14  
Overlay Transfer Address  
Control Card Parameter: 10-5

Parallel  
FOR Statements: 4-21  
Items: 6-10  
Table Structure: 5-19

Parameters, Control Card: 10-1  
Binary Output: 10-2  
COMPOOL: 10-2  
COMPOOL Assembly: 10-4  
List: 10-2  
Monitor: 10-3  
Optimization: 10-3  
Optional Prime: 10-4  
Overlay Transfer Address: 10-5  
Program Name: 10-5  
Single Statement Scheduling: 10-5  
Source Input: 10-1  
Terminate Compilation: 10-4

PAUSE (See STOP)

POS: 2-4, 2-19, 6-7

Presets: 2-6

Primitives: 2-15  
Prime: 2-1  
Prime-Defined: 2-15

PRINT: 6-19

PRINTF: 6-19

PROC: 7-9, 7-16, 8-3

Procedure: 7-8  
Call: 7-14  
Declaration: 7-8  
Parameter Passing: 7-13

Processing Declaration: 7-1  
Closed Forms: 7-6  
Switches: 7-1

Program  
Description: 1-1  
Sample: I-1, I-2, I-5, I-7  
Structure: 8-6, I-1  
Programmer Packing: 5-24  
Program Name  
Control Card Parameter: 10-5  
Pseudo Instructions  
ASSIGN: 4-27, L-6  
BSS: L-5  
BSSZ: L-5  
DIRECT-JOVIAL: 2-5, 4-27  
Storage Instruction: L-5

Registers  
Direct Code: L-1, L-3

Relational Operators: 2-2

Restriction: C-1

RETURN: 2-3, 7-17

Run-Time Error Monitor: 9-8

Scaling, Fixed-Point: J-1  
Additional: J-2  
Division: J-3  
Exponentiation: J-3  
Multiplication: J-2  
Subtraction: J-2

SCOPE File Name: 6-3

Separators: 2-5

Sequential Operators: 2-3

Serial Table: 5-20

SHUT INPUT: 2-4, 6-14

SHUT OUTPUT: 2-4, 6-16

Simple Item: 6-10  
Exchange Example: K-3

Single Statement Scheduling  
Control Card Parameter: 10-5

Source Input  
Control Card Parameter: 10-1

Statements: 4-1  
Assignment: 4-2  
Complex: 4-1  
Compound: 4-1  
Control: 4-12  
DIRECT-JOVIAL: 4-26  
Exchange: 4-9, K-3  
FOR: 4-17  
Forms: 4-1, 6-16  
Input/Output: 6-11, 6-13, 6-15  
Labels: 4-2  
Loop: 4-16  
Messages: A-1  
MONITOR: 9-1  
Named: 4-1  
PAUSE (See STOP)  
Program Control: 4-25  
Simple: 4-1  
Source: L-2  
STOP: 4-25

Status  
Assignment Statements: 4-8  
Constant: 2-14  
Data: 2-9  
Declarations: 5-6  
Exchange Statements: 4-11  
Relational Formula: 3-6  
Variable: 2-26

STOP: 2-3, 4-25

String Item: 5-33

Structure, Program  
Main: 8-6  
Parallel Table: 5-20  
Serial Table: 5-20  
Subprogram: 8-7

Subprogram  
Declaration: 8-2  
Reference: 8-5  
Structure: 8-7

Switches: 7-1  
Index Switch: 7-1, 7-2  
Item Switch: 7-3, 7-4

Symbols: 2-1, L-3  
JOVIAL Defined: 2-2  
User Defined: 2-7  
Scope of Definition: 2-27

Tables: 5-17, 6-11  
Constant List: 5-35  
Declaration: 5-25, 5-26  
Defined: 5-18, 5-30  
Dense Packing: 5-22  
Entries: 6-11  
Items: 6-10  
Like: 5-18, 5-34  
Modifiers: 5-40  
No Packing: 5-21  
Ordinary: 5-18, 5-27  
Overlay Utilization: 5-25  
Packing: 5-21  
Programmer Packing: 5-24  
Referencing: 5-39  
Rigid-Length: 5-20  
Size: 5-19  
Structure: 5-18, 5-20  
Types: 5-18  
Variable-Length: 5-20

Terminate Compilations  
Control Card Parameter: 10-4

TEST: 2-3, 4-24

User-Defined Symbols: 2-7  
Constants: 2-9  
Data Type Properties: 2-7  
Names: 2-14  
Scope of Definition: 2-27  
Variables: 2-16

Variable  
Boolean: 2-25  
Definitions: 2-17  
Entry: 2-26  
Functional Modifier: 2-18  
Indexed: 2-17  
Items: 6-10  
Literal: 2-23  
Loop: 2-16, 2-17  
Named: 2-17  
Numeric: 2-19  
Simple: 2-17  
Status: 2-26  
Usage: 2-19

# READER SURVEY FORM

**MANUAL TITLE** JOVIAL Compiler Reference Manual

**PUBLICATION NO.** 17302500

**REVISION** A

**FROM:** NAME: \_\_\_\_\_

BUSINESS  
ADDRESS: \_\_\_\_\_

HOW DO YOU USE THIS PUBLICATION ?

CDC EMPLOYEE ?

As a reference source \_\_\_\_\_

Yes \_\_\_\_\_

As a classroom text \_\_\_\_\_

No \_\_\_\_\_

As a self-study text \_\_\_\_\_

As a user's guide \_\_\_\_\_

BASED ON YOUR OWN EXPERIENCE, RATE THIS PUBLICATION

As a reference source:

\_\_\_\_ very good    \_\_\_\_ good    \_\_\_\_ fair    \_\_\_\_ poor    \_\_\_\_ very poor

As a text for learning:

\_\_\_\_ very good    \_\_\_\_ good    \_\_\_\_ fair    \_\_\_\_ poor    \_\_\_\_ very poor

As a user's guide:

\_\_\_\_ very good    \_\_\_\_ good    \_\_\_\_ fair    \_\_\_\_ poor    \_\_\_\_ very poor

WHAT IS YOUR OCCUPATION ? \_\_\_\_\_

DID THE MANUAL MEET YOUR NEEDS ?    \_\_\_\_ Yes    \_\_\_\_ No

WHAT SPECIFIC THINGS WOULD BE OF USE TO YOU AS A USER, TEACHER, OR STUDENT ? \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

IS THE MANUAL EASY TO READ AND UNDERSTAND ?    \_\_\_\_ Yes    \_\_\_\_ No

IS IT WELL ORGANIZED ?    \_\_\_\_ Yes    \_\_\_\_ No

WE WOULD APPRECIATE YOUR SPECIFIC COMMENTS; PLEASE GIVE PAGE AND LINE REFERENCES WHERE APPROPRIATE. IF YOU WISH A REPLY, BE SURE TO INCLUDE YOUR NAME AND ADDRESS.

**NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.**

STAPLE

STAPLE

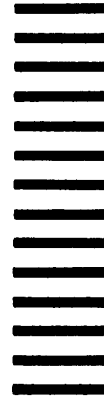
FOLD

FOLD

FIRST CLASS  
PERMIT NO. 8241  
MINNEAPOLIS, MINN.

**BUSINESS REPLY MAIL**  
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY  
**CONTROL DATA CORPORATION**  
Systems Publications  
215 Moffett Park Drive  
Sunnyvale, California 94086



CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE



## COMMENT SHEET

**MANUAL TITLE** JOVIAL Compiler Reference Manual

**PUBLICATION NO.** 17302500 **REVISION** A

**FROM:** NAME: \_\_\_\_\_

BUSINESS  
ADDRESS: \_\_\_\_\_

### COMMENTS:

This form is not intended for use as an order blank. Your evaluation of this manual is welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be noted below. Please include page number references and fill in the publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer Engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN U.S.A.

A43419 REV. 11/69

**NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.**

STAPLE

STAPLE

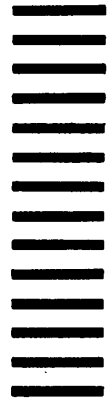
FOLD

FOLD

FIRST CLASS  
PERMIT NO. 8241  
MINNEAPOLIS, MINN.

**BUSINESS REPLY MAIL**  
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY  
**CONTROL DATA CORPORATION**  
Systems Publications  
215 Moffett Park Drive  
Sunnyvale, California 94086



CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE