NASP–9288–A5f11
Multiple Virtual Storage (MVS)
IBM JOVIAL Language
User's Manual
Model A5f1.1

PAGE CONTROL CHART

The following pages contain technical and/or editorial changes for the A5f1.1 update, dated 25 August 2000. Pages that have not changed for this update will maintain their previous levels and dates.

Pages:

Cover
i
iii

NASP–9288–A5f11

NATIONAL AIRSPACE SYSTEM

En Route

MULTIPLE VIRTUAL STORAGE (MVS)

IBM JOVIAL Language

User's Manual

Model A5f1.1

25 August 2000

This manual (JOVIAL) contains information to help programmers write JOVIAL language programs.

Operational Support
National En Route Automation Division, AOS–300
Federal Aviation Administration
William J. Hughes Technical Center
Atlantic City International Airport, New Jersey 08405

CHANGE HISTORY

Level	*Date*	*Case File or PTR No.*	*Change/Comment*
02H	07 May 1986		A4e0.0 HCS Update.
02H	14 October 1986		A4e0.0 HCS Update.
02H	15 August 1995		Reissued for conformity of standards and format, and CD−ROM delivery.
A5f11	25 August 2000		A5f1.1 publication. HOCSR phase 2 enhancements.

PREFACE

This manual contains the information needed to write JOVIAL language programs. It is intended primarily as a reference document, for use by programmers with some familiarity with the JOVIAL language. It may be used as a source document to teach JOVIAL; however, the organization of the manual is not intended to provide a course outline for such a class, nor is it designed as a self-instruction guide. This manual is intended for programmers who have experience using at least one compiler programming language, such as PL/1 or FORTRAN.

Following a general introduction to the JOVIAL language, the statement formats and rules for writing data descriptions and operative statements are presented. Appendixes contain additional reference material, including a list of language formats.

Use of the JOVIAL direct code facility permits parts of programs to be coded in assembly language. Familiarity with the Data Processing System: Basic Assembly Language User's Manual (BALASM) is necessary for using direct code.

NOTATION USED IN THIS PUBLICATION

The following notation is used in the language statement formats throughout this publication:

1. Upper-case words and letters must appear exactly as given in the format.

2. Lower-case words and letters represent material to be supplied by the programmer. Usually the type of material is indicated. For example, if the word "item-name" appeared in a format, the programmer would supply the name of an item.

3. Braces { } indicate a choice. One and only one of the items enclosed in braces is to be used.

4. Brackets [ ] indicate optional material. The programmer decides whether the item enclosed in brackets is to be used or omitted. If both braces and brackets could be used; i.e., a choice is to be made of optional material, the braces are omitted from the format. The items from which a choice is to be made are listed one under the other to avoid any possibility of confusion.

5. Three dots (an ellipsis) indicate that the last complete unit is to be repeated one or more times. A unit is a word (or words) enclosed in braces or brackets.

6. Punctuation, where shown, is required.

7. Compound words consisting of several words separated by hyphens are used to designate elements of the JOVIAL Language which cannot be adequately described by simple English words. Examples of this are:

   floating-point

   table-item-name

   closed-compound-procedure-name.

This document supersedes NASP−9288−02H, dated 14 August 1995 and incorporates changes for the Host Computer System (HCS).

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0 INTRODUCTION

The JOVIAL language is designed to be used in both scientific and commercial applications. A JOVIAL language statement is flexible, so that it is not cumbersome for scientific applications, and the variety of data formats makes it useful for commercial applications.

The JOVIAL language is composed of data declaration statements that assign names to and describe the forms of data items to be used in the program, and operative statements that specify the operations to be performed to solve the problem.

Some of the important features of the JOVIAL language are:

1. A method of maintaining independent, externally stored lists of data declaration statements that can be referred to in any JOVIAL program. A list of these independent data declaration statements is called a compool. If the use of a compool is requested in a JOVIAL program, the data in the compool can be referred to just as though it were defined in the JOVIAL program. Rules for creating and maintaining compools are given in the publication IBM Data Processing System: Compool Edit User's Manual (CMPEDT).

2. A facility for incorporating sections of coding written in IBM Basic Assembler Language into JOVIAL programs. The part of the program written in Basic Assembler Language (BAL) is called direct code. Any machine instructions and any assembly instructions can be used in direct code. Conventions have been established to permit direct code references to JOVIAL defined data. The use of direct code is explained in Section 7.

3. A method of segmenting programs through the use of defined procedures. A defined procedure, which consists of operative statements and, possibly, data declaration statements, is given a name and can be called when needed. The five kinds of defined procedures are: closed compound procedures, functions, procedures, library routines, and closed programs. The form and the provisions for data communication between regions varies with the type of defined procedure. A region is a function, a procedure, or a program excluding functions and procedures. Defined procedures are explained in Section 6.

1.1 IBM 3083 JOVIAL LANGUAGE

IBM 3083 JOVIAL language programs are translated by the IBM JOVIAL Compiler into Basic Assembler Language which is further translated by the Assembler into loader text. The compiler, which is a component of the Utility Programming System for the IBM Data Processing System, is described in the IBM Data Processing System: Compool Edit User's Manual (CMPEDT).

JOVIAL language programs contain two types of statements:

1. Data declaration statements used by the compiler to determine the form in which data is to be stored.

2. Operative statements used by the compiler to determine the operations that must be performed to solve the problem.

1.1.1 JOVIAL Character Set

The JOVIAL character set is composed of 48 characters:

26 letters — A through Z

10 numbers — 0 through 9

12 special characters — characters: + – = ' (,) . $ / * blank

In a source program the character set can be represented in either EBCDIC or BCD card image codes except in character type constants. EBCDIC constants can include any EBCDIC character and must be represented in EBCDIC card image code. ASCII constants can include a subset of the ASCII character set and must be represented in EBCDIC card image code.

1.1.2 JOVIAL Symbolic Names

A JOVIAL symbolic name is an alphanumeric character string used to identify an element of the user's program. It must consist of two to six characters, of which the first must be alphabetic and the remainder alphabetic or numeric. Symbolic names are used for data declarations, statement labels, procedure and function declarations, etc. JOVIAL reserved words (see Appendix A) may not be used as symbolic names.

1.1.3 Data for JOVIAL Programs

Data to be used in a JOVIAL program can be read into storage from a peripheral device or can be entered directly by specifying the value in the program. The JOVIAL language does not contain any input/output statements. Input and output operations can be specified in direct code or in JOVIAL procedure call statements to library routines. An explanation of the library routines and their associated procedure call statements is given in the publication Library Subroutines (LIBRARY) User's Manual.

Data items can be constants or variables. A constant is an unchanging value. For example, 32 is a constant. The description of a constant is implicit in its representation, so no data declaration statement need be given. Constants can appear in operative statements.

A variable is an element of data whose value may change during program execution. For example, if BB is a variable, the value of BB could be changed from 32 to 18 to 10 during program execution. The initial values of some variables can be specified by constants or read into storage from peripheral devices. In either case, an explicit description of the variable must be given in a data declaration statement. The description includes the name of the variable and a description of the variable field format. Variables are referred to by name in operative statements. A reference to the name of a variable refers to the current value of the variable.

Variables can be combined to form tables and arrays. A table is a 2-dimensional structure composed of a series of repeated entries. An entry is composed of one or more variables. Usually each entry will have the same format, although the values of the variable fields will differ from entry to entry. Information in a table is referred to in an operative statement by subscripting the name of a variable in the table to specify the entry.

An array is a structure of one or more dimensions. Each variable field in the array has the same structure, but the number of fields in each dimension can be different. An element in an array can be referred to in an operative statement by subscripting the name of the array, one subscript for each dimension.

1.1.4 Operations in JOVIAL Programs

Operative statements in JOVIAL programs specify arithmetic, logical, and control operations. They can refer to variables by the name of the variable, to constants by the value of the constant, and to other operative statements by statement labels that can be assigned to operative statements to identify them.

Three of the most frequently used operative statements in a JOVIAL program are the Assignment statement, the GOTO statement, and the IF statement.

The Assignment statement is used to perform arithmetic operations or to set a variable field to the value of another field. For example, the statement:

```
CHECK. AA = 37 + BB $
```

is an Assignment statement, labeled CHECK. When this statement is executed, the current value of the variable field BB is added to 37 and the sum is stored in the variable field AA. The $ is terminal punctuation for all JOVIAL statements.

The GOTO statement is used to transfer from one part of a program to another. For example, the statement:

```
GOTO CHECK $
```

is a GOTO statement. When this statement is encountered, control transfers to the statement labeled CHECK.

The IF statement is used to evaluate a condition. A condition is a comparison of an expression to a constant or another expression. The condition will be true or false when the statement is executed depending upon the current value of the expressions. If the condition is true, the statement after the IF statement is executed. If the condition is false the second statement after the IF statement is executed. For example,

```
IF AA EQ BB $

GOTO CHECK $

AA = BB - CC $
```

is an IF statement whose true exit is the GOTO statement GOTO CHECK $ and whose false exit is the Assignment statement AA = BB−CC $. If the value of the variable field AA is equal to the value of the variable field BB when the statement is executed, the true exit is taken. If not, the false exit is taken.

Some of the other operative statements can be used to exchange the values of two variable fields, can defined procedures, and perform integer division providing a quotient and remainder.

1.2 PROGRAM ORDER

A JOVIAL program has a relatively free format. A few rules have been imposed to ensure efficient compiler translation. These are:

1. A JOVIAL program begins with a START statement followed by data declaration and operative statements, and is terminated by a TERM statement.

2. Data declaration statements and operative statements can be mixed in the program, but data must be declared before it can be referred to in an operative statement.

3. Coding may begin in any column and must not extend beyond column 66. Terms, which make up statements, must not be split from one line to the next. A new statement need not begin on a new line. Statements can be continued for as many lines as necessary as long as the statement (excluding comments) does not exceed 2000 characters and the number of terms does not exceed 256. A list of JOVIAL terms is given in Appendix A.

4. A blank (or blanks) is used to separate terms. Terms must not contain embedded blanks.

5. A dollar sign ($) terminates all statements. A dollar sign must not appear in column 1, or the statement will be misinterpreted as a system control statement.

6. Every program should have a STOP statement unless it is a library program.

1.3 COMMENTS

Comments are narrative statements that may be written within or between JOVIAL statements. Anywhere a blank appears or is permitted in a JOVIAL program, a comment may be inserted. Comments are enclosed in two pairs of single apostrophe marks "...". Any JOVIAL characters may be used in a comment except consecutive apostrophe marks, or the sequence dollar-blank.

1.4 PROGRAM REGIONS

A single JOVIAL program may be divided by the compiler into several Regions in order to facilitate processing. Often it is necessary to be aware of the Region structure in order to produce a validly operating program. The following definitions constitute program regions.

1. Any procedure or function not containing nested procedures or functions.
2. That portion of a main or CLOSE program excluding procedures or functions.
3. The outer PROC in a LIBE program excluding nested procedures or functions.
4. Compool.

The regions can most easily be distinguished by the assigned prefixes. Categories 2 and 3 above always have the prefix pair A0/A1, and may be referred to as "Main" regions.

1.4.1 Accessible Regions

Data definitions in one region may or may not be usable in code in another region. In order that they be usable, the region of definition must be "accessible" to the region of use. The following rules define accessibility:

1. Any region is accessible to itself.
2. Compool, if used, is accessible to any region.
3. The "Main" region is accessible to any procedure or function region.
4. A procedure or function region is not accessible to any region except itself.

1.5 DATA TYPES

The following list gives the types of constants and variables that can be used in a JOVIAL program.

Integer

Fixed-Point

Floating-Point

Hexadecimal

EBCDIC

ASCII

Status value

Table Address

The following sections give the formats for the different types of data. There is a special format for each type of constant and each type of variable.

A constant that is written in one of the formats appear directly in an operative statement. A data declaration statement is not necessary to describe the field format of the constant, because the compiler can determine the storage format from the form of the constant.

Variables, however, must be described in data declaration statements. The variable field format descriptions given in the following sections are used as part of several different kinds of data declaration statements. In addition to specifying the variable field format, data declaration statements also name the variables and indicate whether they are stored as single items or stored in tables or arrays. Data declaration statements are described in Section 2.

1.5.1 Integer Type

An integer is a whole number. For example, 23, 467, −3, 0, and 511 are integers. Integers must not contain decimal points.

When integers end with a large number of zeros (e.g., 56,000,000 or −20,000,000,000), it is easier to write them as an integer number times a multiple of 10 than to write out all the zeros. For example, 56,000,000 is equal to 56 x 10^6; −20,000,000,000 is equal to −2 x 10^{10} (or −20 x 10^9, etc.).

1.5.1.1 Integer Constant Format. The format of integer constant is:

{ integer
integerE+n
integerEn }

---

integer	1 to 10 decimal digits, optionally preceded by a sign.
E	Required to indicate that the number following is an exponent (power) of 10. This is just short representation of the symbols “x 10” in the examples given above. For example, 26 x 10^3 is written in the JOVIAL format as 26E3.
n	Positive integer representing the power to which 10 is raised before it is multiplied by the integer. If the plus sign is omitted, it is assumed.

RESTRICTIONS: The following restrictions apply to integer constants.

1. An integer consists of from 1 to 10 decimal digits.
2. The absolute value of an integer must not exceed $2^{31}-1$. If it exceeds $2^{31}-1$, it is set to 0 and a warning message is issued.
3. When an integer is used as a subscript (described in the section “Referring to Data”), it is treated modulo 2^{24}.

1.5.1.2 Integer Variable Field Format. Descriptions of variable fields to contain integer data have the form:

I bits [sign]

I — Specifies integer type.

bits — Total number of binary bits required to represent the largest number anticipated. It must include one bit for the sign if the item will contain a signed value. The total number of bits, including the sign bit, must not exceed 32. Appendix D contains information about the number of binary bits required for a decimal number.

sign — The letter S or U. S indicates that the values will be signed. They may be positive or negative. U indicates that the values will be unsigned. If the sign designator is omitted, it will be assumed to be signed.

RESTRICTIONS: The following restrictions apply to integer variable fields.

1. If an integer value is too large for a field, the high-order bits are truncated. For example, a 3-bit field will be set to 1(001) if the value 9(1001) is assigned to it.

2. If an integer value does not fill a field, it is right justified and the field is filled out with leading zeros.

EXAMPLES OF INTEGER DATA: In the following examples, the constants would be permissible values for the corresponding variable field.

Variable Description			Constant
I	2	U	3
I	6	S	+25
I	14	S	−4096
I	32	U	2147483647
I	15	U	25E3
I	32	S	−36E7
I	32	U	60E+7

1.5.2 Fixed-Point Type

A fixed-point number is a real number. It consists of a whole number and a fraction, which is separated from the number by a decimal point. For example, 32.5, −1.6,.032, and 6. are fixed-point numbers.

Although the customary way to write fixed-point numbers is a whole number and a fraction, fixed-point numbers can be represented as a number times a power of 10. This is permitted whenever it is convenient, including, for example, 6.10^{6},.7x10$-^{6}$, and $77.63x10^{1}$ to represent 6,000,000,.0000007, and 776.3, respectively.

In the JOVIAL format the location of the binary point as well as the decimal point must be given. The binary point location tells how much significance (how many binary digits) are to be allocated to the fractional part of the fixed-point number. The hexadecimal-decimal conversion table in Appendix D can be used to aid in determining how many binary bits are required to store the fractional part of a decimal (base 10) number.

1.5.2.1 Fixed-Point Constant Format. The format of a fixed-point constant is:

{ fixed-pointAm
fixed-pointAmEn
fixed-pointAmE+n
fixed-pointAmE−n
fixed-pointEnAm
fixed-pointE+nAm
fixed-pointE−nAm }

---

fixed-point	Number consisting of from 1 to 10 decimal digits and an explicit decimal point. It is optionally preceded by a plus or minus sign. If the sign is omitted, the number is assumed to be positive.
A	Required to indicate that the number following is a binary point specification.
m	An unsigned integer specifying the scale factor (number of bits to the right of the binary point). It must not exceed 31. It must always be specified even if the value is 0, indicating that the constant is an integer.
E	Required to indicate that the number following is a power of 10.
n	Integer representing the power to which 10 is raised before it is multiplied by the fixed-point number. If the sign is omitted, it is assumed to be plus.

RESTRICTIONS: The following restrictions apply to fixed-point constants.

1. The absolute value of a fixed-point constant must not exceed $2^{31} - 1$. That is, it must not be less than −2,147,483,647 nor greater than +2,147,483,647.
2. If the scale factor is too large, if the number exceeds 10 digits, or if a 10-digit number is outside the permissible range, the number is cleared to 0 and a warning message is issued.
3. If the scale factor is not large enough to accommodate the fractional portion, the fractional portion is truncated, and the integer portion is not changed. A warning message is issued.

1.5.2.2 Fixed-Point Variable Field Format. The format of variable fields that will contain fixed-point data is:

A bits [sign] [scale]

A	Specifies fixed-point type.
bits	Total number of binary bits required to represent the largest number anticipated. It must include one bit for the sign if the values are signed. The total number of bits, including the sign bit, must not exceed 32. See Appendix D.
sign	The letter S or U. S indicates that the values will be signed. They may be positive or negative. U indicates that the values will be unsigned. If the sign designator is omitted, it will be assumed to be signed.

scale — Indicates the number of binary digits to the right of the binary point specification, thus specifying the accuracy desired. The scale must not exceed 31 bits and must not exceed the total number of magnitude bits. If the scale is omitted, it is assumed to be 0.

RESTRICTIONS: The following restrictions apply to fixed-point variable fields:

1. If a fixed-point number is too large for the field, it is aligned by the binary point specification, and low-order fraction bits and high-order integer bits are truncated.

2. If a fixed-point number does not fill the field, it is aligned by the binary point specification and extra high-order and/or low-order bits are set to 0.

EXAMPLES OF FIXED−POINT DATA: In the following examples, the constants would be permissible values for the corresponding variable field.

	Variable Description			Constant
A	6	U	3	5.2A3
A	3	U		4.A0 (treated as integer 4)
A	32	S	1	−032145.A1
A	16	U	3	5.2A3E2
A	16	S	2	+6.8E3A2

1.5.3 Floating-Point Type

A floating-point number is also a real number. It consists of a whole number and a fraction (which is separated from the whole number by a decimal point) times a power of 10. For example, 21.2 x 10^4, −3.6 x 10^2,4. x 10^{-6}, and .32 x 10^2 are all floating-point numbers.

Floating-point numbers can be written as just a whole number and a fraction, and the power of 10 is assumed to be 0. For example, 3.26 is equivalent to 3.26 x 10^0.

The major difference between fixed-point and floating-point numbers is in how they are stored in the computer.

A fixed-point number is stored, in binary, as a whole number and a fraction separated by a binary point.

A floating-point number is stored, in binary, as an exponent and a fraction. The exponent occupies bits 1−7 and the fraction occupies the remainder of the word including bit 0, which contains the sign of the fraction. The quantity expressed by this number is the product of the fraction and the number 16 raised to the power of the exponent.

Very large numbers should be stored in floating-point format since they can be manipulated with less possibility of a loss of accuracy than fixed-point numbers.

1.5.3.1 Floating-Point Constant Format. The format of floating-point constants is:

```
{ floating-point          }
{ floating-pointE+n       }
{ floating-pointEn        }
{ floating-pointEn        }
```

floating-point	Number consisting of from 1 to 8 decimal digits and an explicit decimal point. It is optionally preceded by a plus or minus sign. If the sign is omitted, it is assumed to be positive.
E	Required to indicate that the number following is a power of 10.
n	Integer representing the power to which 10 is raised before it is multiplied by the floating-point number. If the sign is omitted, it is assumed to be plus.

RESTRICTIONS: The following restrictions apply to floating-point constants.

1. The largest permissible floating-point constant is $\pm.72{,}370{,}051 \times 10^{76}$; the smallest is $\pm.53{,}976{,}054 \times 10^{-78}$.
2. If the entire floating-point number (sign, decimal digits, decimal point, E, and exponent) exceeds 15 characters, the number is set to 0 and a warning message is issued.
3. If the number of decimal digits exceeds 8, the excess low-order digits are ignored, but the exponent is adjusted. No warning message is issued.
4. If the number is greater than the maximum permissible number, there is a loss of accuracy in the low-order digit. No message is issued.
5. If the number is less than the minimum permissible number, the number is set to 0, but no message is issued.
6. If the exponent is greater than 76, the number is set to the maximum positive floating-point number ($72{,}370{,}051 \times 10^{76}$) and a warning message is issued.
7. If the exponent is less than -78, the number is set to 0. No message is issued.

1.5.3.2 Floating-Point Variable Field Format. Floating-point fields are used when the number of decimal digits in the values is either large or unpredictable. Floating-point numbers are stored alone in full computer words. The format of variable fields that will contain floating-point data is:

```
F
```

F	Indicates floating-point type data.

RESTRICTIONS: The following restriction applies to floating-point variable fields.

Floating-point numbers must not exceed the limits $\pm 7 \times 10^{75}$ and $\pm 5 \times 10^{-79}$. It is the programmer's responsibility to ensure this; otherwise, results are unpredictable.

EXAMPLES OF FLOATING-POINT DATA: In the following examples, the constants would be permissible values for the corresponding variable field.

Variable Description	Constant
F	9.321
F	+99.012
F	−5000.99
F	00012.592 (leading zeros ignored)
F	3.2E+6
F	−31.5E+7
F	.4
F	6.

1.5.4 Hexadecimal

Hexadecimal data is represented to the base 16. It is composed of from 1 to 16 alphanumeric characters chosen from the numbers 0 through 9 and the letters A through F. It is unsigned. Hexadecimal numbers are manipulated as though they were unsigned integers and may be used wherever an integer value may be used. The usual use of a hexadecimal constant is to set up a bit pattern in a field. Appendix D contains decimal to hexadecimal conversion tables.

1.5.4.1 Hexadecimal Constant Format. The format of a hexadecimal constant is:

X(hexadecimal-number)

X	indicates hexadecimal
hexadecimal-number	Unsigned number to the base 16 composed of from 1 to 16 hexadecimal characters.

EXAMPLES OF HEXADECIMAL CONSTANTS: Hexadecimal variable fields are not permitted. The following are examples of hexadecimal constants.

Constant
X(7AC)
X(23FD0C98A)

1.5.5 EBCDIC Type

EBCDIC data is made up of from one to eight characters chosen from the 256-character EBCDIC character set.

1.5.5.1 EBCDIC Constant Format. The format of EBCDIC constants is:

---

pH(characters)

P	Positive number indicating the number of characters enclosed in parentheses. It must not exceed eight.
H	Specifies EBCDIC type.
characters	From one to eight EBCDIC characters depending upon "P."

1.5.5.2 EBCDIC Variable Field Format. The format of variable fields that will contain EBCDIC data is:

H limit

H	Indicates EBCDIC
limit	Maximum number of characters in the largest EBCDIC value. It must not exceed eight.

RESTRICTIONS: The following restrictions apply to EBCDIC variable fields.

1. If EBCDIC data is too large to fit in the field, high-order bits are truncated.
2. If EBCDIC data does not fill the field, it is right-justified and preceded by leading zeros.

EXAMPLES OF EBCDIC DATA: In the following examples, the constants would be permissible values for the corresponding variable field.

Variable Description		Constant
H	6	6H(JOVIAL)
H	3	3H(5 /b 3)
H	5	5H($1.25)
H	8	7H(SMALLER)
H	6	7H(TOO /b BIG)

1.5.6 ASCII Type

ASCII data is composed of from one to eight characters chosen from the 128-character ASCII character set. However, since JOVIAL source programs are coded in EBCDIC, only the subset of ASCII in EBCDIC may be used in ASCII constants. EBCDIC constants will be converted to their ASCII equivalent in storage. Other characters will be converted to ASCII null (code 00).

1.5.6.1 ASCII Constant Format. The format of ASCII constants is:

---

pC(characters)

---

p	Positive number indicating the number of ASCII characters enclosed in parentheses. It must not exceed eight.

C — Specifies ASCII type.

characters — From one to eight ASCII characters, depending upon "p."

1.5.6.2 ASCII Variable Field Format. The format of variable fields that will contain ASCII data is:

C limit

C — Indicates ASCII type.

limit — Positive number specifying maximum number of characters in the largest ASCII value. It must not exceed eight.

RESTRICTIONS: The following restrictions apply to ASCII variable fields.

1. If ASCII data is too large for the field, high-order bits are truncated.
2. If ASCII data does not fill the field, it is right-justified and preceded by leading zeros.

EXAMPLES OF ASCII DATA: In the following examples, the constants would be permissible values for the corresponding variable field.

Variable Description		Constant
C	1	1C( )
C	4	4C(2 ␣␣ 4)
C	3	3C(RAL)
C	5	5C($1.25)
C	8	7C(SMALLER)
C	6	7C(TOO ␣ BIG)

1.5.7 Status Value Type

Status value variables are a special class of unsigned integer variable, each of whose values is assigned a symbolic name (status value constant) by the programmer. Status value constants may be used only when their relationship to a variable can be unquestionably construed from the context. This is necessary because identical status value constants may represent different numeric values when assigned to different status value variables and the symbolic names may be the same as other symbolic names used in a program.

This data type is used for information which, while it must be reduced to numeric representation within a computer, is logical in nature and can be more meaningfully described by a symbolic name. For example, in describing color, words like "red", "blue", or "green" are more convenient than the numbers "0", "1", or "2".

While the numeric representation of a status value constant is not normally of interest, it may be determined from the constant's position in the variable data declaration because the status value constants are assigned the values 0, 1, 2,... in order of appearance in the statement.

1.5.7.1 Status Value Constant Format. The format of status value constants is:

V(status-name)

V	Indicates status value.
status-name	One to six alphanumeric characters, the first of which is alphabetic.

1.5.7.2 Status Value Variable Field Format. The format of variable fields that will contain status values is:

S V(status-name) V(status-name) [V(status-name)] ...

S	Indicates status value.
V(status-name)	Status value constant.

EXAMPLES OF STATUS VALUE DATA: In the following examples, the constants used in the variable field description are the only values the variable field can assume.

Variable Description	Constant		
S V(YES) V(NO)	V(YES)	=	0
	V(NO)	=	1
S V(RED) V(BLUE)	V(RED)	=	0
	V(BLUE)	=	1

1.5.8 Table Address Type

Subject to the following restrictions, a table name or unsubscripted array name can be used as a constant whose value is the beginning storage address of the table or array. These constants are 24-bit unsigned integers.

1.5.8.1 Table Address Constant Format. The format of a table address constant is:

---

symbolic-name

---

symbolic-name	Name of a table or array.

RESTRICTIONS: The following restrictions apply to table address constants.

1. Table address constants may be used only as input parameters in function and procedure calls. (See “Defined Procedures”.)

2. Table address constants may be used as single parameters only and may never be combined with other operands in an expression.

2.0 DESCRIPTION OF DATA

Data items used in a JOVIAL program must be described in data declaration statements before they can be manipulated in operative statements. The description names the data items and indicates their forms.

Data used in a JOVIAL program can be constant or variable. Constant data is entered by specifying the value directly in the program. The compiler sets up a storage area in the proper format.

The programmer describes the fields for storing variable items. He specifies the general format of the variable field (type, size, etc.) and assigns a name to the variable. The values in the field can be changed during program execution by referring to the variable by name. The section "Referring to Data" explains how variable information is used in operative statements.

2.1 DATA DECLARATION STATEMENTS

Data declaration statements specify whether variables are stored as single independent items, are combined with other variables to form a table, or stored as an array of one or more dimensions. In either case, a description the type of variable (fixed-point, integer, etc.) must be given.

If variables are combined to form a table, data declaration statements specifying the size and structure of the table must also be given.

The following statements are used to describe data fields.

Statement	Purpose
ITEM	Describes field format for single variables and for variables to be combined to form a table.
TABLE	Specifies the number of entries in a table and gives the entry structure. ITEM and STRING statements are used to describe the variables that make up an entry.
STRING	Describes a special type of variable used in a table entry.
ARRAY	Describes the format of an array by giving the number of dimensions and size of each dimension and the variable field format of the elements in the array.
EQUATE	Allows two or more single items, tables, or arrays to use the same storage area.

2.2 ITEM DESCRIPTIONS

There are three types of ITEM statements.

1. The parameter ITEM statement assigns names to constants.
2. The compiler-allocated ITEM statement describes variables unrelated to other variables (single items) and variables grouped to form compiler-allocated tables (table items). With this type of statement, the compiler determines how the items are to be allocated in computer words.
3. The programmer-allocated ITEM statement describes table items in programmer-allocated tables. The programmer specifies how the items are to be located in computer words. Because programmer-allocated ITEM statements are used only to describe entries in programmer-allocated tables, this type of statement is explained in the section "Programmer-Allocated TABLE Statement".

2.2.1 Parameter ITEM Statement

The parameter ITEM statement is used to assign a symbolic name to a constant value, so that the symbolic name can be used instead of the constant in operative statements. The primary advantage of named constants is that if the constant is changed from one compilation of the program to another, only the constant value in the parameter ITEM statement need be changed.

It is not necessary to specify a field format in the statement since the compiler determines storage allocation from the constant value.

The format of a parameter ITEM statement is:

```
ITEM item-name constant-value $
```

item-name	A programmer-assigned symbolic name. This name is used to refer to the constant value.
constant-value	Any constant written in one of the formats shown in Figure 2−1. A complete description and examples of constant formats are given in Section 1, under "Data Types".

RESTRICTIONS: The following restrictions apply to parameter ITEM statements.

1. The value of the constant can not be changed during program execution.
2. Status value and table address constants must not be used in parameter ITEM statements.
3. A parameter ITEM statement cannot be used instead of compiler-allocated or programmer allocated ITEM statements to describe a table entry.
4. Whenever an integer constant can be used in a data declaration statement (e.g., number of words in an entry), an integer parameter item may be substituted, provided that the parameter item is defined in an accessible region.

EXAMPLES: The examples in Figure 2−2 show parameter ITEM statements.

2.2.2 Compiler-Allocated ITEM Statement

The compiler-allocated ITEM statement is used to describe single items or table items in a compiler-allocated table.

Type of Constant	Format
Integer	integer integerE+n integerEn
Floating-point	floating-point floating-pointE+n floating-pointEn floating-pointE−n
Fixed-point	fixed-pointAmE±n fixed-pointE±nAm fixed-pointAm fixed-pointAmEn fixed-pointEnAm
Hexadecimal	X(hexadecimal)
EBCDIC	pH(EBCDIC-characters)
ASCII	pC(ASCII-characters)
Explanation of format	
integer	1 to 10 decimal digits, optional sign.
floating-point	1 to 8 decimal digits, explicit decimal point, optional sign.
fixed-point	1 to 10 decimal digits, explicit decimal point, optional sign.
hexadecimal	1 to 16 alphanumeric characters, no sign.
EBCDIC-characters	1 to 8 characters from complete EBCDIC character set.
ASCII-characters	1 to 8 characters from the ASCII character set.
n	Exponent. Integral power of 10 by which number is multiplied.
m	Binary point specification. Unsigned integer specifying number of bits to right of binary point.
p	Positive number indicating maximum number of characters in EBCDIC or ASCII constant.

FIGURE 2−1. CONSTANTS FORMATS

Example	Meaning
ITEM EDIT 4H(EDIT) $	The name EDIT refers to the EBCDIC constant EDIT.
ITEM NTGR −64 $	The name NTGR refers to the integer constant −64.
ITEM EX 2.7183 $	The name EX refers to the floating-point constant 2.7183.
ITEM EXFIX 2.7183A16 $	The name EXFIX refers to the fixed-point constant 2.7183 (which has 16 binary bits to represent .7183).

FIGURE 2−2. PARAMETER ITEM STATEMENTS

The format of the compiler-allocated ITEM statement is:

```
ITEM item-name field-format $
```

item-name — A programmer-assigned symbolic name. When this name is used in an operative statement, it refers to the current value of the item.

field-format — The variable field format of the item. Figure 2−3 gives the codes used to specify field formats. A detailed description of variable field formats is given in Section 1, under “Data types”.

RESTRICTIONS: The following restrictions apply to single items described by compiler-allocated ITEM statements.

1. Fixed-point values are aligned by the binary point when they are stored in the field by operative statements. Leading and trailing zeros are supplied for values not large enough to fill the fields. High-order integer bits and/or low-order fractional bits are truncated if the values are too big for the field.
2. If binary point alignment is not required, values are right-justified in the field and preceded by leading zeros, if necessary. High-order integer bits are truncated if the values are too big for the field.
3. Floating-point numbers are stored alone in a full-word.

Rules for compiler-allocated table items are given in Section 2, under “Compiler-Allocated TABLE Statement”.

EXAMPLES: The following examples show compiler-allocated ITEM statements.

Example	Meaning
ITEM FF F $	Item FF contains floating-point data.
ITEM II I 5 S $	Item II contains signed integer data with a maximum size of four binary bits and a sign.
ITEM AA A 8 U 4 $	Item AA contains unsigned fixed-point data whose maximum size is eight bits, four bits of which are to the right of the binary point.
ITEM HH H 5 $	Item HH contains EBCDIC data whose maximum size is five characters.
ITEM CC C 5 $	Item CC contains ASCII data whose maximum size is five characters.

Type of Item	Format
Integer	I bits [sign]
Floating-point	F
Fixed-point	A bits [sign] [scale]
EBCDIC	H limit
ASCII	C limit
Status	S V(status-name) V(status-name) [V(status-name)]...
Explanation of format	
bits	Total number of bits required to store largest integer or fixed-point number. The number of bits must not exceed 32, including the sign.
sign	The letters S or U. S represents signed items. U represents unsigned items. If the sign designator is omitted, items are assumed to be signed.
scale	Number of bits to the right of the binary point. If it is omitted, it is assumed to be zero. Scale must not exceed 31.
limit	Number of characters in EBCDIC or ASCII item. The number of characters must not exceed 8.
V(status-name)	Status value constant.

FIGURE 2–3. FIELD FORMATS FOR COMPILER–ALLOCATED ITEM STATEMENTS

2.3 TABLE DESCRIPTIONS

A table is a list of information. The rows of the table are called entries. The columns of the table are called table items or items in the entry. All of the items in one column have the same variable field format. For example, consider a population table composed of 50 entries, one for each state. Each entry contains two items: the name of the state and the population of that state. It is not necessary to specify the format of each of the 50 entries. A description of one entry is all that is necessary. It would consist of an EBCDIC (or ASCII) item to contain the state name and an integer item big enough to contain the largest state population. A TABLE statement is used to specify that this entry format is to be repeated 50 times.

A description of a table has two major parts. The first is the TABLE statement specifying the general format of the table. The second is one or more ITEM statements specifying the entry format. An entry may not exceed 4096 bytes.

There are two categories of tables: programmer-allocated tables and compiler-allocated tables. The programmer specifies the arrangement of the entries in computer words for programmer-allocated tables by using programmer-allocated ITEM statements to describe the entry format. The compiler determines storage allocation for compiler-allocated tables.

Subscripts are used to refer to particular entries in a table or to items in an entry. Modifiers can be used to determine special kinds of information about a table; e.g., number of words in an entry and number of entries in a table. They can also be used to refer to an entire table or particular pieces of data in a table and to modify a table structure. Rules for subscripts and modifiers are given in Section 3, “Referring to Data”.

There are some special statements that make table descriptions easier. The STRING statement is used to repeat variable field formats within an entry to form variable-length entries. The duplicating TABLE statement is used to repeat the format of an entire table.

The operative statement FOR is particularly useful for manipulating information in tables (see Section 5, “Operative Statements”).

2.3.1 Compiler-Allocated TABLE Statement

The compiler determines how items in compiler-allocated tables are to be located in computer words. The programmer can specify in the TABLE statement whether the items are to be stored in fullwords, half-words, or bytes, and whether the table is to be fixed or variable in length. If he wishes to provide more detailed instructions for storage allocation, he must use the programmer-allocated TABLE statement. Compiler-allocated ITEM statements follow the TABLE statement to specify the entry format.

The format of a compiler-allocated TABLE statement, followed by compiler-allocated ITEM statement is:

TABLE table-name

{ V / R } entry-limit [{ N / M / D / B }] $

BEGIN item-statement

[item-statement]...END

---

table-name — Programmer-assigned symbolic name used to refer to the table.

V — Indicates variable-length table.

R	Indicates fixed-length table.
entry-limit	Number of entries in fixed-length table and maximum number of entries in variable-length table.
N	Specifies that each item is stored by itself in a fullword if it will fit, or in two consecutive fullwords, right-justified.
M	Specifies that each item is stored by itself in a halfword if it will fit, in a fullword, or in two consecutive fullwords, right-justified.
D	Specifies that each item is stored in the smallest number of bytes required to hold it.
B	Same as D.
	If N, M, B or D is not specified, N-type packing is automatically provided.
BEGIN	JOVIAL opening bracket. Indicates that the ITEM statements that follow describe the table entry.
item-statement	Compiler-allocated ITEM statement describing an item in the entry.
END	JOVIAL closing bracket. Indicates the end of the entry description.

RESTRICTIONS: The following restriction applies to compiler-allocated tables.

If M is given as the packing factor, the maximum number of information bits exclusive of sign that can be stored in a halfword is 15. That is, 2-character EBCDIC or ASCII items, or 16-bit unsigned integer or fixed-point items will be stored in fullwords.

EXAMPLE: The following statements are used to describe a compiler-allocated table containing state names and state populations.

```
TABLE      STPOP     R  50  N                    $
  BEGIN
   ITEM    STATE     C   8                       $
   ITEM    POP       F                           $
  END
```

The fixed-length Table STPOP contains 50 entries. Each entry contains two items; an ASCII item for the state name and a floating-point item for the population. The floating-point items are stored separately in fullwords, the ASCII items are stored separately in two consecutive words.

2.3.2 Programmer-Allocated TABLE Statement

In programmer-allocated tables, the programmer can specify the allocation of storage for items within entries of the table. He does this by describing the entry format with programmer-allocated ITEM statements and/or STRING statements that specify the word and bit location for the item. A table may be declared without items or strings, simply to allocate storage. The BEGIN and END brackets are still required.

The format of a programmer-allocated TABLE statement followed by programmer-allocated ITEM statements or STRING statement is:

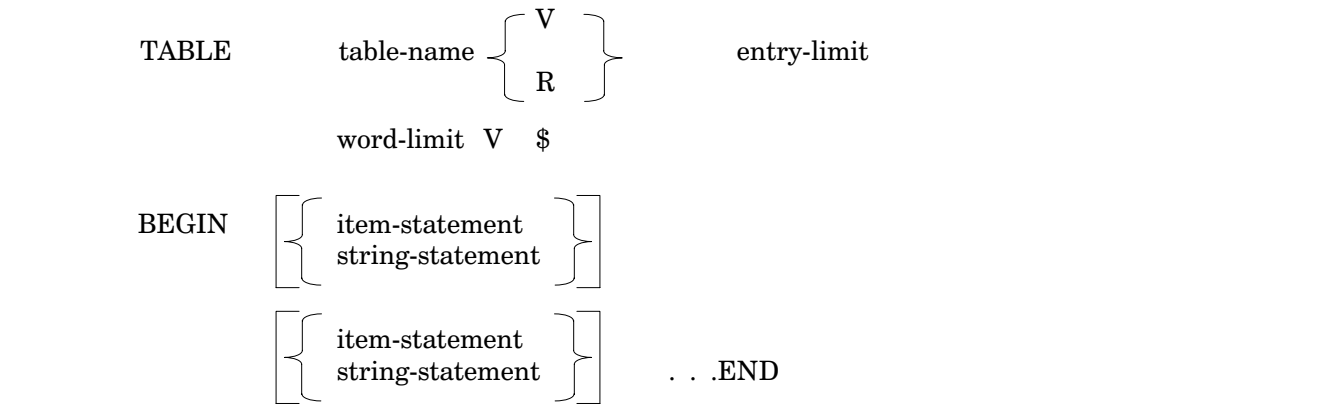
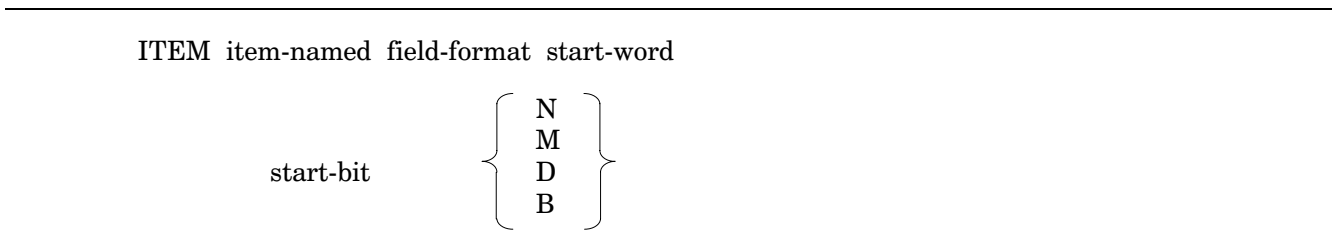


table-name	Programmer-assigned symbolic name.
V	Variable-length table.
R	Fixed-length table.
entry-limit	Number of entries for fixed-length tables and maximum number of entries for variable-length tables.
word-limit	Number of words per entry. (When variable length entries are to be used, as discussed under "Variable Length Table Entries", the word limit should be 1 and the entry limit should be the entire number of words allocated to the table.) This value may not exceed 1024.
V (preceding $ sign)	Variable-entry-length table. Suppresses diagnostic warning "WORD NUMBER IN THIS DECLARATION INCONSISTENT WITH ENTRY LENGTH."
BEGIN	JOVIAL opening bracket. Indicates that the statements that follow are related and describe the table entry.
item-statement	A programmer-allocated ITEM statement; explained under "Programmer-Allocated ITEM Statement."
string-statement	A STRING statement. Explained under "STRING Statement."
END	JOVIAL closing bracket. Indicates end of entry format.

2.3.3 Programmer-Allocated ITEM Statement

The programmer-allocated ITEM statement allows the programmer to specify the allocation of storage for items in a table. It is used only to define items in programmer-allocated tables.

The format of the programmer ITEM statement is:



item-name	Programmer-assigned symbolic name
field-format	The variable field format of the item. Figure 2−4 gives the codes used to specify field-format.
start-word	Word that the item will occupy in the table entry. Word count starts with 0.
start-bit	Starting bit position for item in the computer word. Bit numbering starts with 0.
N	Code that indicates item is alone in a fullword or two fullwords and is right-justified.
M	Code that indicates item is alone and right-justified in a halfword. A maximum of 15 magnitude bits are permitted; i.e., I 15 U and I 16 S are legal, but I 16 U is not.
D	Code that indicates item occupies only as many bits as necessary to hold it.
B	Code that indicates item is unsigned and alone and right-justified within a byte.

Type of Item	Format
Integer	I bits [sign]
Floating-point	F
Fixed-point	A bits [sign] [scale]
EBCDIC	H limit
ASCII	C limit
Status	S V(status-name) V(status-name) [V(status-name)]...
Explanation of format	
bits	Total number of bits required to store largest integer or fixed-point number. Total number of bits required to store number representing number of status values in programmer-allocated items. The number of bits must not exceed 32.
sign	The letters S or U. S represents signed items. U represents unsigned items. Unsigned items are assumed to be positive.
scale	Number of bits to the right of the binary point. If it is omitted, it is assumed to be zero. Scale must not exceed 31.
limit	Maximum number of characters in EBCDIC or ASCII item. The number of characters must not exceed 8.
V(status-name)	Status value constant.

FIGURE 2–4. FIELD FORMATS FOR PROGRAMMER–ALLOCATED ITEM STATEMENTS

RESTRICTIONS: The following restrictions apply to programmer-allocated ITEM statements.

1. Integer and fixed-point items must not cross a word boundary.
2. Floating-point items require fullwords, and must be declared as N-packed.
3. EBCDIC or ASCII items of four or fewer characters must not cross a word boundary.
4. EBCDIC or ASCII items of more than four characters must not cross two word boundaries.
5. M cannot be specified as the packing factor for 2-character EBCDIC and ASCII items, or 16-bit unsigned integer or unsigned fixed-point items.
6. B cannot be specified as the packing factor for signed items.

EXAMPLE: The following example is a description of a programmer-allocated table.

Statement	Comments
TABLE AA R 1000 3 $	Table AA has 1000 entries. Each entry occupies 3 words.
BEGIN	Opening bracket.
ITEM BB I 15 U 2 3 D $	Unsigned integer field BB begins in bit 3 of word 2 and occupies 15 bits.
ITEM CC H 2 0 16 N $	EBCDIC field CC begins in bit 16 of word 0 and is alone and right-justified in a fullword.
ITEM DD I 2 U 2 18 D $	Unsigned integer field DD begins in bit 18 of word 2 and occupies 2 bits.
ITEM EE S 2 V (DRY) V(DAMP) V(MOIST) V(WET) 1 6 B $	Status field EE begin in bit 6 of word 1. It occupies 2 bits because the number 3 representing the last status value V(WET) can be stored in a 2-bit field.
END	Closing bracket.

Figure 2−5 gives the storage format for an entry in Table AA.

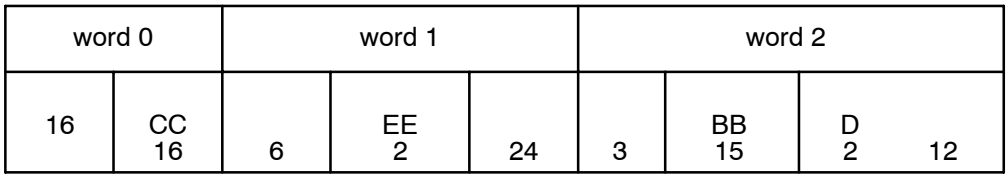


FIGURE 2−5. STORAGE FORMATS

2.3.4 STRING Statement

The STRING statement is used to specify the repetition of variable field formats in programmer allocated table entries. Each repetition in a string is called a bead. The number of beads can vary (under programmer control) from entry to entry. This is explained under “Variable-Length Table Entries.”

Several beads of a string can appear in one computer word and then a word or words (possibly containing single items or beads from another string) can be skipped and several beads can appear in the next computer word.

The STRING statement specifies the number of beads in a computer word, their location in the word, and the number of words before the appearance of the next bead in the same string. The format of the STRING statement is:

```
STRING  string-name  field-format  start-word

                      { N }
start-bit             { M }          skip  bead-limit     $
                      { D }
                      { B }
```

string-name	Programmer-assigned symbolic name.
field-format	Variable field format of the beads in the string. Codes for variable field formats are given in Figure 2–4.
start-word	Computer word that the first bead will occupy in an entry. Word numbering starts with 0.
start-bit	Starting bit for first bead in each computer word that contains beads of the string. Bit numbering starts with 0.
N	Bead is alone and right-justified in a fullword.
M,	All beads are alone and right-justified in a halfword.
D	Bead occupies only as many bits as declared.
B	All beads are alone and right-justified in a byte.
skip	Number of computer words to the next word that contains beads. For example, if words 3, 5, and 7 of a entry contain beads, skip would be 2.
bead-limit	Maximum number of beads in one computer word.

RESTRICTIONS: The following restrictions apply to strings.

1. Strings can be used only in programmer-allocated tables.
2. Initial values cannot be assigned to beads in a string.
3. A bead must not cross a word boundary.
4. If more than one bead is assigned to a single computer word, following beads are positioned, with no space, immediately to the right of the previous bead.
5. M cannot be specified as the packing factor for 2-character EBCDIC and ASCII beads, or 16-bit unsigned integer or unsigned fixed-point beads.
6. B cannot be specified as the packing factor for signed beads.

EXAMPLE: The following example shows a programmer-allocated table whose entries are described with programmer-allocated ITEM statements and STRING statements.

```
TABLE      ZZ          R1000      1                          $
  BEGIN
   ITEM    AA          I  32  U   0   0  N                   $
   ITEM    BB          I  32  U   1   0  N                   $
   STRING  CC          A  10  U   4   2  0  D   2   3       $
   STRING  DD          F   3  0                  N   2   1   $
  END
```

1. Fixed-length Table ZZ has 1000 1-word entries.
2. The ITEM and STRING statements describing the entry format in Table ZZ are enclosed in BEGIN and END brackets.
3. The 32-bit unsigned integer field AA starts in bit 0 of word 0. It is alone and right-justified in a fullword.
4. The 32-bit unsigned integer field BB starts in bit 0 of word 1. It is alone and right-justified in a fullword.
5. String CC is composed of 10-bit unsigned fixed-point beads with a binary point specification of four. The first bead in an entry starts in bit 0 of word 2. One computer word separates each word containing beads; i.e., words 2, 4, 6, etc., may contain beads. There may be three beads in each computer word. They are neither alone nor right-justified in the words.
6. String DD contains floating-point beads. The first bead starts in bit 0 of word 3. Two computer words separate words containing beads. There is one bead per computer word. It is alone and right-justified in the word.
7. The programmer must control the number of computer words in each entry that contains beads. "Variable-Length Table Entries" explains how this is done.

2.3.5 Duplicating TABLE Statement

The duplicating TABLE statement is used to describe a new table whose entry format is identical to another table in the same region. (A region is a function or a procedure or a program excluding functions and procedures. Functions and procedures are explained in Section 6, "Defined Procedures.") The values, the number of entries, and the specification of whether the table is fixed or variable in length for the new table may differ from the original table. The original table is not affected by the duplication. The format of the duplicating TABLE statement is:

TABLE modified-table-name $\left[\left\{ \begin{matrix} V \\ R \end{matrix} \right\} \text{ entry-limit} \right]$ L$

---

modified-table-name	The name of the original table modified by the addition of a 1-character alphanumeric suffix. The new name must be a permissible symbolic name.
V	Indicates variable length.
R	Indicates fixed length.
entry-limit	Maximum number of entries for variable-length table and number of entries for fixed-length table.

L Indicates duplicating TABLE statement.

RESTRICTIONS: The following restrictions apply to duplicating tables.

1. The original table must be declared in a region accessible to the region in which the duplicating table statement appears, or the original must be in compool.
2. If the program is recompiled with a new description for the original table, the duplicated table will be changed accordingly.
3. Any preset contains in the original table will not be duplicated in the new table.
4. If the original table and the new table are to share the same storage area, an EQUATE statement must be used.
5. All items in the new tables are referred to by the item-names in the original table modified by the 1-character alphanumeric suffix. For this reason, the original table and table item names must not exceed five characters.
6. If the length specification (variable or fixed) is omitted, it is assumed to be the same for the new table as for the original table.
7. If either the entry-limit or the length specification is included in the declaration, both must be included. If only one or the other specification is present, a serious diagnostic will be issued.

EXAMPLE: In the example in Figure 2−6, Table AAB duplicates Table AA. Note that Table AA is variable in length and Table AAB is fixed in length; Table AA has 100 entries and Table AAB has only one entry; Table AA IS 200 words long and Table AAB is 2 words long.

Original Table	Statement	New Table
TABLE AA V 100 $ BEGIN ITEM AAA F $ ITEM BBB I 16 U $ END	TABLE AAB R 1 L $	TABLE AAB R 1 $ BEGIN ITEM AAAB F $ ITEM BBBB I 16 U $ END
Note: The statements given to describe Table AAB do not actually appear in the program.		

FIGURE 2−6. DUPLICATED TABLE

2.3.6 Variable Length Table Entries

Programmer-allocated tables can contain variable-length entries if STRING statements are used in the description of the entry format. Because the number of beads in the string is not specified in the STRING statement, the number can vary from entry to entry under programmer control.

When the input is prepared or the table is built during execution, the programmer must provide means of knowing the position of each entry in the table and the number of beads in each string in each entry. The compiler does not provide bookkeeping, but merely accesses data in accordance with the item and string declarations and indexes according to the word limit declared in the table statement. Appropriate programmer control varies with circumstances because the number of beads per entry may be fixed may be indicated by a specially coded terminating bead, or may be indicated by control items in the same or related table.

EXAMPLE: Assume that a variable-length table of the form shown in Figure 2−7 is to be read into storage from a peripheral device.

STOCK							
BINNO	ITEM	PARTA	PARTB	PARTC	PARTD	***	PARTN
36	0						
37	3	050	060	070			
38	6	051	152	153	154	***	
*	*	*	*	*	*	***	*
*	*	*	*	*	*	***	*

FIGURE 2–7. INPUT INFORMATION

The following statements should be given to describe the table.

```
TABLE      STOCK     V2500      1                    $
  BEGIN
   ITEM    BINNO     I   12  U  0   0  D            $
   ITEM    ITUM      I    6  U  0  12  D            $
   STRING  PART      I    6  U  1   0  D  1   5     $
  END
```

ITUM is used as a control item to indicate the number of beads in String PART for an entry.

Figure 2–8 shows the structure of the first three logical entries in table STOCK. Because the number of words per logical entry varies, the number of words per entry was given in the TABLE statement as 1 in order to make every word in the table accessible. There are six actual entries in the first three logical entries of the table.

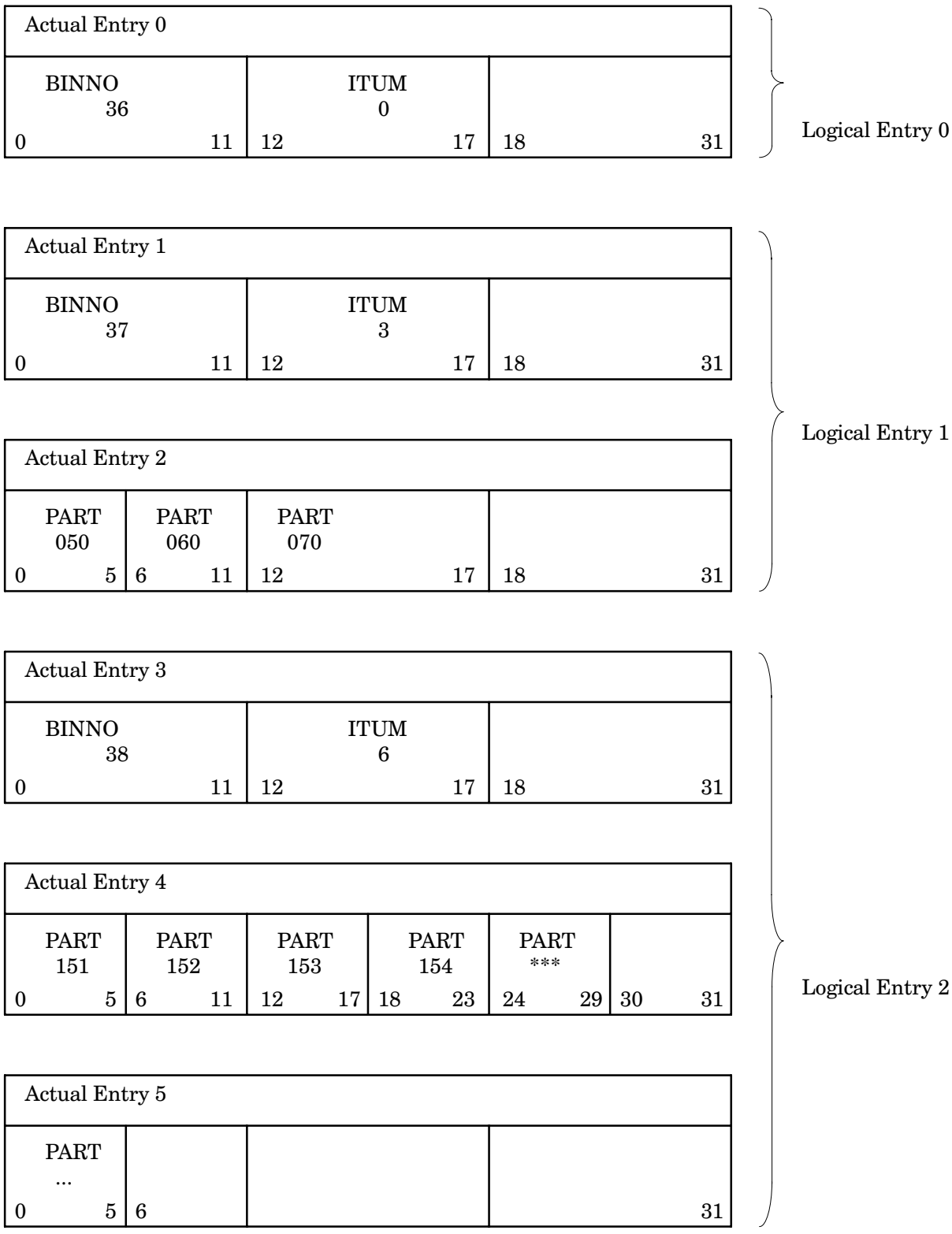


FIGURE 2–8. VARIABLE–LENGTH TABLE ENTRIES

2.3.7 Variable Structure Table Entries

In programmer-allocated tables, the structure of the entries can be variable. That is, the items in successive entries can vary in name, size, field-format, etc. The programmer must provide some method of determining what kind of items are in the entry he is processing.

When the input is prepared or the table is built during execution, the programmer must provide means of knowing which items vary. The compiler does not provide bookkeeping but merely accesses the data according to the item and table declaration. Appropriate control varies with circumstances because the type of item may be indicated by control items in the same or a related table or by distinctive characteristics of the item.

EXAMPLE: The following statements describe a table with variable structure entries. The item ACTYPE (aircraft type) is used as a control item. When the value of ACTYPE is TRNS (transport), item 3 is TONAGE. When the value of ACTYPE is PASS (passenger), item 3 is CAP (capacity).

```
TABLE       DESCR      R   4   1   2              $
  BEGIN
   ITEM     ACTYPE     H   4       0    0   N         $
   ITEM     MAXALT     I   7   U   1    0   D         $
   ITEM     MAXSPD     I  11   U   1    7   D         $
   ITEM     TONAGE     I  12   U   1   20   D         $
   ITEM     CAP        I   8   U   1   18   D         $
  END
```

If ACTYPE has a value of TRNS in entry 0 and a value of PASS in entry 1, Figure 2–9, shows the structure of the first two entries of Table DESCR.

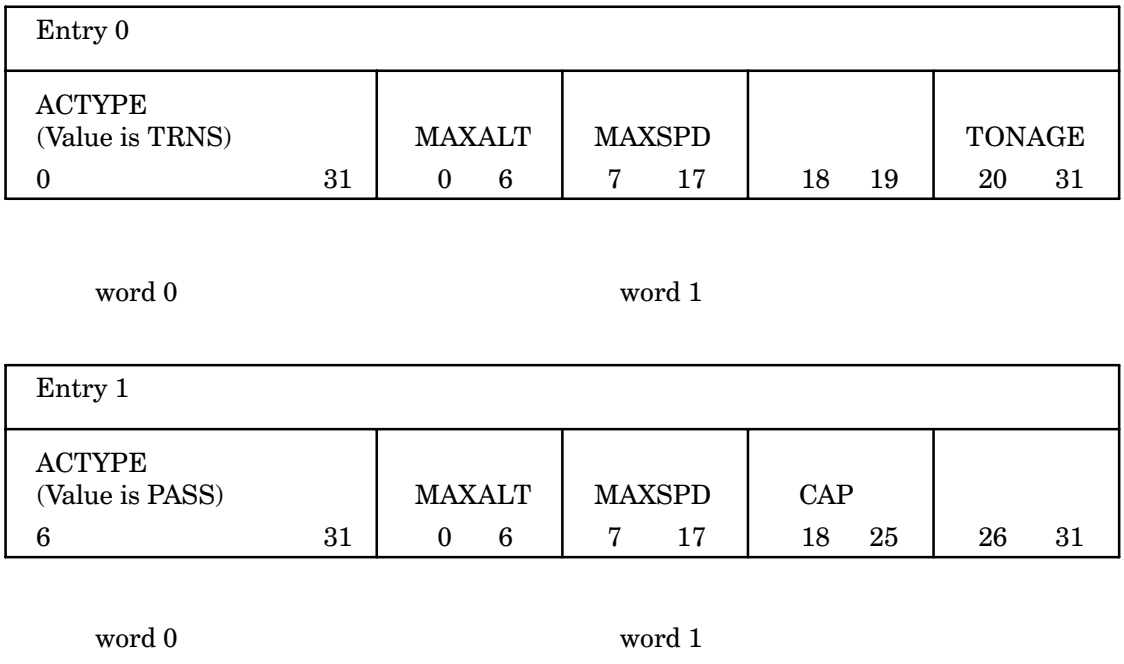


FIGURE 2−9. VARIABLE−STRUCTURE TABLE ENTRIES

In compiler-allocated tables, the EQUATE statement can be used to form entries with variable structures. An example is given in the discussion of the EQUATE statement.

2.3.8 Initial Values for Table Items

Initial values can be assigned to table items in compiler-allocated or programmer-allocated tables by placing a list of constants for the item immediately after the ITEM statement. The list of constants is enclosed in the BEGIN and END brackets. The first constant in the list is the initial value of the item in the 0th entry; the second, the initial value of the item in the 1st entry etc. Since the items are named and the field formats are given, the values of the item can be changed during the program execution. The constants must be permissible values for the field type specified. Figure 2−10 gives permissible constant formats. Constant formats are explained in detail in Section 1, under “Data Types”.

Type of Constant	Format
Integer	integer integerE+n integerEn
Floating-point	floating-point floating-pointE+n floating-pointEn floating-pointE−n
Fixed-point	fixed-pointAm fixed-pointAmEn fixed-pointEnAm fixed-pointAmE±n fixed-pointE±nAm
Hexadecimal	X(hexadecimal)
EBCDIC	pH(EBCDIC-characters)
ASCII	pC(ASCII-characters)
Explanation of format	
integer	1 to 10 decimal digits, optional sign.
floating-point	1 to 8 decimal digits, explicit decimal point, optional sign.
fixed-point	1 to 9 decimal digits, explicit decimal point, optional sign.
hexadecimal	1 to 16 alphanumeric characters, no sign.
EBCDIC-characters	1 to 8 characters from complete EBCDIC character set.
ASCII-characters	1 to 8 characters from complete JOVIAL character set.
n	Exponent. Integral power of 10 by which number is multiplied.
m	Binary point specification — Unsigned integer specifying number of bits to right of binary point.
p	Positive number indicating maximum number of characters in EBCDIC or ASCII constant.

FIGURE 2−10. CONSTANT FORMATS

EXAMPLE: The following statements assign initial values to the first item in each entry of the compiler allocated table STPOP.

```
TABLE     STPOP     R  50  D                    $
  BEGIN
   ITEM   STATE     C   8                       $
  BEGIN   8C(ALABAMA)  8C(ALASKA)...END
  END
```

2.4 ARRAY STATEMENT

An array is a data organization of one or more dimensions. The number of dimensions necessary is determined by the number of reference points needed to uniquely locate a piece of information (element) in the data organization.

One reference point is needed to located an element in a 1-dimensional array. For example, consider a 1-dimensional array formed by listing the letter A through Z. Then B is uniquely referred to by specifying the second entry in the list.

Two reference points are needed to locate an element in a 2-dimensional array. For example, consider a map of elevations. If a grid of horizontal and vertical lines is laid over the map so that one elevation is recorded in each square of the grid, then any elevation can be uniquely referred to by specifying the horizontal and vertical entries.

In a similar manner, in an n-dimensional array, n reference points are needed to uniquely locate an element. In JOVIAL, all of the elements must have the same variable field format and a unique reference to an element is made by subscripting the name of the array, one subscript for each dimension. Subscripting is explained in Section 3 “Referring to Data”.

An array is described with an array statement. The format of an ARRAY statement is:

ARRAY array-name d_1 d_2 ...d_n field-format $

---

array-name	Programmer-assigned symbolic name.
d_i	Integer value representing the size of dimension.
field format	Variable field format of elements in the array. Codes for field formats are given in Figure 2−4.

RESTRICTIONS: The following restrictions apply to the ARRAY statement.

1. Only one variable field format is given for the entire array.
2. Status value fields are not permitted in arrays.
3. See Section 7, “DIRECT Code,” for information on arrangement of elements in storage and space required per element.

EXAMPLE: The following statement describes a 3-dimensional array which has 64 elements (floating-point variable fields). The size of the first dimension is 8; the size of the second dimension is 4; the size of the third dimension is 2.

```
ARRAY AA 8 4 2 F $
```

2.5 EQUATE STATEMENT

The EQUATE statement permits two or more items, table, or arrays to share the same storage area.

There are three major varieties of EQUATE statement: Item Equate, Structure Equate, and Dynamic Equate. The following paragraphs describe each of these.

2.5.1 Item Equate

The format of an Item Equate statement is:

```
EQUATE  symbolic-name=symbolic-name
          [=symbolic-name]  ...  $
```

---

symbolic-name Name of a single item or table item.

RESTRICTIONS: The following restrictions apply to Item Equates.

1. Single items may be equated only with other single items.
2. Fields of different data types can be equated. For example, a floating-point single item can be equated to an EBCDIC single item.
3. Table items may be equated only with other table items in the same table. When table items are equated, the EQUATE statement must precede the END bracket enclosing the item statements describing the entry format.
4. Programmer-allocated table items must not be equated.
5. Strings must not be equated.
6. Parameter items must not be equated.

2.5.2 Structure Equate

The Structure Equate permits two or more tables (or arrays) to partially or totally overlay one another. In the following description, the term "table" is intended to include arrays as well. The user need make no distinction between the two as used in Structure (or Dynamic) Equates.

The format of a Structure Equate is as follows:

```
EQUATE  series1=series2[=series3...]  $
```

---

series i is tablei [,tablei2,...].

When several Structure Equates have common elements, they are considered to be logically part of the same EQUATE, and they are processed together.

However, when a Structure Equate has a common element with a Dynamic Equate, it is considered "subordinate" and the Dynamic Equate rules apply.

RESTRICTIONS: The following restrictions apply to structure equates that are not subordinate to dynamic equates:

1. If tablei appears in more than one equate statement, it must be the only element of a series (i.e., not preceded or followed by a comma) in all but the first occurrence.

2. If a compool table appears in a structure equate, it must be alone in a series.

3. Tables whose initial values have been assigned must not be equated. This does not apply to a compool table appearing in a program-defined EQUATE statement.

4. There can be, at most, one compool-defined table or array in a program-defined EQUATE statement.

5. If two or more Structure Equate EQUATE statements form a chain, e.g., EQUATE AA = BB $ EQUATE BB = CC $, there can be, at most, one compool defined term in the entire program-defined Structure Equate chain.

6. If one table or array in a Structure Equate statement is defined in the compool, the location of the common storage area is in the compool-defined area.

7. The symbolic names used in the EQUATE statement must have been defined previously, and must all be in the same region, or in compool.

8. In any single Structure Equate statement, a maximum of one element may have appeared in a previous EQUATE.

2.5.3 Dynamic Equate

The Dynamic Equate allows the user to dynamically assign the location of a table or tables. Effectively, the "form" of the table (or of any structure of tables which can be defined using Structure Equates) is laid onto the storage area specified by the user.

The format of a Dynamic Equate is:

EQUATE item/series1 [=series2...] $

---

series i — is as for structure equate

item — must obey the following rules:

1. It must be a single item, table item, or procedure dummy item.

2. It must be I or A type, with at least 24 magnitude bit, and must be n-packed.

3. If it is a table item, the table containing it must be single entry, and must not itself appear in a dynamic equate.

When a dynamic equate is defined, the value in the "base item" is not preset. It must be set by the user. When a reference is made to a dynamic table (or its items), the word containing the base item is loaded and the low-order 24 bits used as an address. If the value of the item is changed by name, the base register will automatically be reset. Although the base item may be overlaid by item equates or table definition, references to overlaying items will not cause the value of an in-use base to change. (See also Section 8, Reload Pseudo-op.)

No storage will be allocated for an "internal" table which appears in a dynamic equate. Storage allocation for compool tables is dependent on the presence of other references to the segment. However, the compool table is, for all practical purposes, redefined internally. It is given an A1 or other local prefix, and the original location is lost. To regain the original location requires the use of direct code, possibly including a BAL format PSEG card image, or a .ƀPSEG card image, referencing the name of the segment or a non-dynamically-equated table therein. The dynamic equate statement must precede any references to tables (and their items) contained in it.

RESTRICTIONS: The following restrictions apply to Dynamic Equates and to subordinate Structure Equates.

1. If tablei appears in more than one EQUATE statement, it must be the only element of a series (i.e., not preceded or followed by a comma) in all but the first occurrence.

2. The symbolic names used in the EQUATE statement must have been defined previously, and must all be in an accessible region, or in compool.

3. No table appearing in a Dynamic Equate may contain an item used as the base for a Dynamic Equate (either itself or another Dynamic Equate).

4. No table may appear in more than one Dynamic Equate statement in a single program region. A table in a Dynamic Equate may, however, appear in one or more subordinate Structure Equates. A table may appear in a Dynamic Equate statement in the main program and also in Dynamic Equate statements in procedures and/or functions.

5. In any single subordinate Structure Equate, only one element may have appeared previously in a Dynamic Equate or subordinate Structure Equate.

6. No element in a Dynamic Equate may have appeared in any previous EQUATE; thus all subordinate Structure Equates must follow their Dynamic Equate Statements.

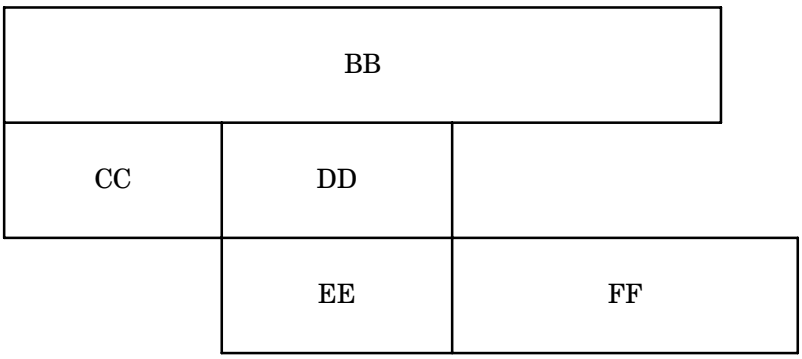
7. The total size of a dynamic structure (defined by a Dynamic Equate, with its subordinate Structure Equates, if any) must not exceed 65,536 bytes.

Example 1:

```
ITEM    AA    I    32    S    $
ITEM    BB    R    20    40   $BEGIN    END
ITEM    CC    R    4     6    $BEGIN    END
ITEM    DD    R    6     4    $BEGIN    END
ITEM    EE    R    10    20   $BEGIN    END
ITEM    FF    R    20    36   $BEGIN    END


EQUATE AA/BB=CC,DD $
EQUATE DD=EE,FF $
```

Resulting structure (no storage actually allocated):



Example 2:

In the following example, ACTUAL and EST (estimate) occupy the same word in an entry. If the field is referred to as ACTUAL, it is treated as floating-point; if it is referred to as EST, it is treated as an integer.

```
TABLE       MANY       V1000                          $
  BEGIN
   ITEM     ACTUAL     F                              $
   ITEM     EST        I    10   S                    $
   ITEM     CODE       H    6                         $
   EQUATE EST = ACTUAL                                $
  END
```

3.0 REFERRING TO DATA

3.1 SUBSCRIPTS

Single items can be referenced in operative statements by the name assigned to the item in the data declaration statement. Subscripts must be appended to data names to refer to items in tables, entries in tables, beads of a string, and elements of arrays. A subscript consists of one or more subscript expressions, separated by commas.

Subscripts are surrounded by dollar signs, and enclosed in parentheses, ($...$). There may be no blanks between the dollar sign and parentheses.

3.1.1 Subscript Expressions

Subscript expressions may be single items, constants, or arithmetic expressions. Because subscript expressions specify the number of the occurrence of the table item, table entry, bead, or array element (first occurrence, fifth occurrence, etc.), the subscript expression must be a positive integer. If the subscript expression contains a fractional part, it is truncated. All subscript expressions are used modulo 2^{24}.

3.1.2 Reference to Table Items

Individual items in a table can be referenced by specifying the name of the item, followed by a subscript expression indicating the number of the entry. The format of a subscripted table item name is:

---

table-item-name ($ subscript $)

Table-item-name	Programmer-assigned item name of ITEM statement associated with a compiler-allocated or programmer-allocated TABLE statement.
subscript	Subscript expression specifying the number of the entry that contains the table item. Entry numbering starts with 0. A subscript of 0 [e.g., table-item-name (0] need not be used, since an unsubscripted table item name refers to the table item in entry 0.

EXAMPLE: In the following example, a subscript is used to refer to the fourth occurrence of Item POP in Table STPOP.

Statement	Comments
TABLE STPOP 50 D $	Table statement.
BEGIN	
ITEM STATE C 8 $	Item statement.
ITEM POP F $	Item statement.
END	
IF POP ($ 3 $) GR 100000 $	Refers to Item POP in the fourth state.

Statement	Comments
GOTO LARGE $	True exit. Transfer to statement labeled LARGE.
GOTO SMALL $	False exit. Transfer to statement labeled SMALL.

3.1.3 Reference to Beads in a String

A bead in a string must be referenced by specifying the name of the string, followed by subscripts indicating the number of entry and the number of the bead in the entry. The format of a subscripted string name is:

---

string-name ($ bead, entry $)

---

string-name	Symbolic name of the string.
bead	A subscript expression specifying the number of the bead in the entry. Bead numbering starts with 0.
entry	A subscript expression specifying the number of the entry in the table. Entry numbering starts with 0.

The expression CC($0, 2$) refers to the first bead in third entry of a table containing String CC. Use of the subscripts is mandatory.

3.1.4 Reference to Elements in an Array

An element in an array must be referenced by specifying the array name, followed by subscripts indicating the location of the element in each dimension of the array. The format of a subscripted array name is:

---

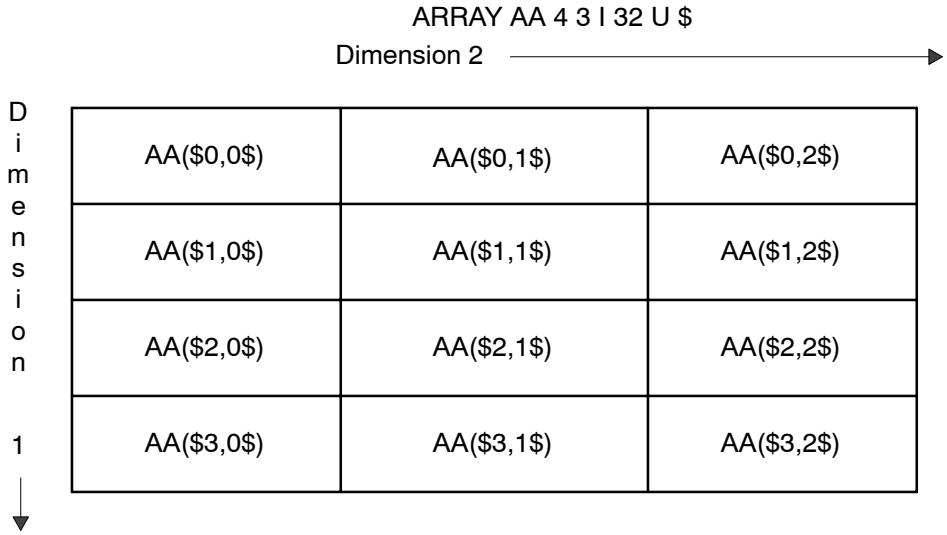
array-name ($$S_1$, S_2...S_n $)

---

array-name	Symbolic name of the array.
S_i	Subscript expression referring to dimension i. The number of subscript expressions specified must equal the number of dimensions. If the size of dimension S_i is n, the value of the subscript expression referring to dimension i can range from 0 through n-1.

Use of an array name without subscripts is treated as a “Table Address Constant” (See Section 1).

EXAMPLE: The following example shows a statement used to describe an array and the subscripts used to refer to items in the array. In the illustration, dimension 1 is vertical; dimension 2 is horizontal.



In Section 5, under "FOR Statement" there is an example of a FOR statement used to process an array.

3.2 MODIFIERS

Modifiers are used to obtain special kinds of information about data. The following list gives the types of modifiers and their purpose.

Modifier	Purpose
BIT	Refer to specific bits in an item.
BYTE	Refer to specific bytes in an item.
ENT	Refer to a table entry.
NENT	Refer to number of entries in a table.
NWDSEN	Refer to number of computer words in a table entry.

ALL is an additional modifier that is used only in FOR statements (see Section 5, under "FOR Statement").

Information about the modifier is enclosed in parentheses and follows the name of the modifier.

3.2.1 BIT Modifier

BIT modifiers are used to refer to specified bits in a single item, table item, array element, or bead of a string. They can be used in arithmetic expressions, IF, and Assignment statements. When this option is used, the bits are treated as a group of unsigned integer bits. The format of a BIT modifier is:

```
BIT ($ first-bit[,total-bits] $)
    (symbolic-name)
```

first-bit — A subscript expression specifying the first bit referenced. When the total-bits field is omitted, the reference is only to this one bit. Numbering begins with 0. If the item is signed, 0 refers to the sign bit.

total-bits	A subscript expression specifying the total number of bits referenced.
symbolic-name	Name of single item, table item, array, or string, subscripted if necessary.

EXAMPLE: In the following example, an ITEM statement describes the integer Item ROUTE, and a BIT modifier is used with IF and GOTO operative statements to determine whether the value of ROUTE is odd or even.

```
ITEM ROUTE I 6 U $
IF BIT ($5$) (ROUTE) EQ 1 $
GOTO ODD $
GOTO EVEN $
```

1. The last bit is referenced as bit number 5 because numbering begins with 0.

2. The first statement after an IF is the true exit; the second is the false exit.

3.2.2 BYTE Modifier

BYTE modifiers are used to refer to specified bytes of a single item, table item, array element, or bead of a string. They can be used in arithmetic expressions and IF and Assignment statements. When this option is used, the bytes are treated as a group of unsigned integer bits. The format of a BYTE modifier is:

```
BYTE ($ first-byte[, total-bytes] $
     (symbolic-name)
```

---

first-byte	A subscript expression specifying the first byte referenced. This is the only byte used if the total-bytes field is omitted. Numbering begins with 0.
total-bytes	A subscript expression specifying the total number of bytes referenced.
symbolic-name	Name of single item, table item, array, or string, subscripted if necessary.

EXAMPLE: In the following example, an ITEM statement describes the EBCDIC Item TYPE, and a BYTE modifier is used with IF and GOTO operative statements to determine whether the value of TYPE is SAMPLE.

```
ITEM TYPE H 8 S

IF BYTE ($2,6$) (TYPE)EQ 6H(SAMPLE) $

GOTO SHOW $

GOTO CALC $
```

3.2.3 ENT Modifier

The ENT (entry) modifier is used to set an entry of a table to zero, to the value of another entry, or to exchange entries.

The format of the ENT modifier used to set an entry or table item to zero is:

```
ENT(symbolic-name($ subscript $))=0 $
```

The format of the ENT modifier used to set an entry to the value of another entry is:

```
ENT(symbolic-name($ subscript $)) = ENT(symbolic-name($ subscript $))$
```

The format of the ENT modifier used to exchange entries is:

```
ENT(symbolic-name($ subscript $)) = = ENT(symbolic-name($ subscript $))$
```

symbolic-name	Name of a table or a table item or string within the table.
subscript	Subscript expression specifying the number of the entry in the table.

RESTRICTIONS: The following restrictions apply to ENT modifiers.

1. If the name of a table item or string is given, the effect is the same as if the name of the table containing the item or string were given, i.e., the complete entry is affected.
2. The three uses given for the ENT modifier are the only ways it can be used; an ENT modifier cannot appear in any other statement.
3. A table entry may only be set from or exchanged with a table entry of equal size.

3.2.4 NENT Modifier

The NENT (number of entries) modifier is a predefined name used to refer to the number of entries in a table. For fixed length tables it is set to the number of entries declared and, like a parameter item, cannot be changed. For variable length tables NENT is a special variable with the implied field description I 24 U. It is initially set to zero by the compiler, and may be altered as well as referenced by the programmer.

The format of the NENT modifier is:

```
NENT(symbolic-name)
```

symbolic-name	Name of table item, string, or table.

RESTRICTIONS: The following restrictions apply to the NENT modifier.

1. If the name of a table item or string is given, the reference is to the table containing the table item or string.
2. NENT modifiers can be used in arithmetic expression.

EXAMPLE: In the following example, TABLE and ITEM statements describe two tables; ENT modifiers manipulate the entries in the tables, and the NENT modifier tests the number of entries in the tables.

Statement	Comments
TABLE TABA V 20 N $	Variable-length, 20 entries, fullword packed.
BEGIN	Bracket.

Statement	Comments
ITEM ONEA H 4 $	4-character EBCDIC field.
ITEM TWOA 32 3 S $	32-bit signed fixed-point field.
END	Bracket.

Statement	Comments
TABLE TABA V 30 N $	Variable-length, 30 entries, fullword packed.
BEGIN	Bracket.
ITEM ONEB F $	Floating-point field.
ITEM TWOB I 32 S $	32-bit signed integer field.
END	Bracket.
.	
.	
.	
IF NENT (TABA) EQ 20 $	Number of entries in TABA equal to 20?
GOTO ROUT $	True. Go to statement labeled ROUT.
ENT(TABA(3)) = 0 $	False. Set entry 3 of TABA to zero.
ENT(ONEA(3)) = ENT(TABA(4)) $	Replace entry 3 of TABA containing (ONEA) with entry 4 of TABA.
ENT(ONEA(2)) = = ENT(TWOB(4)) $	Exchange the words in entry 2 of TABA with the corresponding words in entry 4 of TABB.

3.2.5 NWDSEN Modifier

The NWDSEN (number of words per entry) modifier is a predefined name of an integer constant referring to the number of computer words occupied by a table entry. It is assigned by the compiler for compiler-allocated tables and is determined by the compiler from TABLE statements for programmer-allocated tables. It can be used anywhere a constant may be used in executable statements.

The format of a NWDSEN modifier is:

NWDSEN (symbolic-name)

---

symbolic-name Name of table or table item or string within the table.

RESTRICTIONS: The following restrictions apply to the NWDSEN modifier.

1. If a table item name or string name is given instead of a table name, the reference is to the table containing the table item or string.

2. NWDSEN cannot be modified during program execution; it represents a constant.

EXAMPLE: In the following example, the total number of words in a table is determined by multiplying the number of entries by the number of words in each entry.

```
ITEM TOTAL F $
TABLE TABL R 60 N $
BEGIN
ITEM ITEMA F $ ITEM ITEMB F $
END
TOTAL = NENT(TABL)*NWDSEN(TABL)$
```

3.2.6 LOC Modifier

The LOC (locate address) modifier refers to the location of a table, array, string, or item. LOC is particularly useful because table names must often be referenced with a displacement. The LOC modifier may be used anywhere a constant may be used.

The format of the LOC modifier is:

LOC (symbolic-name)

---

symbolic-name Name of table, array, string, or item.

RESTRICTIONS: The following restrictions apply to the use of the LOC modifier.

1. Only a table, array, string, or item name is acceptable in the “symbolic-name” field. Table names may not be subscripted; string names must be subscripted.

2. LOC cannot be modified during program execution; it represents a constant.

3. In the case of an item name, the value is the address of the word (or halfword, if M-packed) containing the item.

4. The use of LOC (string) is restricted to locating the first bead within an entry. Any valid entry subscript may be used. The bead subscript should be coded as zero. A non-zero bead subscript will be ignored, or, in the case of some dense-packed strings, will result in a diagnostic.

5. In order to process LOC correctly, the compiler requires an entry in the Library PDT for function ZVLOC.

EXAMPLE: The following example shows the MVI Library Routine used to move data into the 10th byte of Table DATA.

```
MVI (LOC(DATA) + 10, 4, 4H(ABCD))$
```

3.2.7 ADR Modifier

The ADR (address) modifier is identical to the LOC modifier except for items and strings with B or D packing. These return byte addresses. For D-packed multi-byte items, the address is of the byte containing the first bit of the item.

The format of the ADR modifier is:

ADR (symbolic-name)

symbolic-name as for LOC.

RESTRICTIONS: The following restrictions apply to the use of the ADR modifier:

1. Restrictions 1, 2, and 4 under LOC also apply to ADR.
2. In order to process ADR correctly, the compiler requires an entry in the Library PDT for function ZVADR.

4.0 ARITHMETIC EXPRESSIONS

4.1 ARITHMETIC OPERANDS AND OPERATORS

An arithmetic expression consists of a single constant or variable, or two or more constants and variables, joined by arithmetic operators. Arithmetic expressions can be used as subscripts or as arguments of BIT and BYTE modifiers. They can also be used in the Assignment, IF, FOR, and GOTO operative statements.

The kinds of terms that can be used in an arithmetic expression are:

1. Single item name.
2. Subscripted table item name.
3. Subscripted string name.
4. Subscripted array name.
5. BIT or BYTE modifiers.
6. NENT or NWDSEN modifiers.
7. Index name.
8. Constants.
9. LOC or ADR modifiers.
10. Function calls.

Integer, fixed-point, floating-point, EBCDIC, ASCII, and hexadecimal constants can be used in arithmetic expressions. Status value constants can be used only as single term expressions and then only in Assignment and if statements in conjunction with their associated status value variable.

Listed below are the JOVIAL arithmetic operators.

Operator	Use
+	Addition
–	Subtraction
*	Multiplication
/	Division
(*...*)	Exponentiation
ABS(...)	Absolute Value

RESTRICTIONS: The following restrictions apply to the use of arithmetic operators in arithmetic expressions.

1. Contiguous operators are not permitted. For example, + − 3 must be written as +(−3).
2. Implied multiplication is not permitted. For example, (XX) (YY) must be written as XX * YY or (XX) * (YY).
3. In exponentiation, if the base is negative, the exponent must be an integer. If the exponent has a fractional part (either fixed-point or floating-point), the base must be positive; the result is a floating-point number. EBCDIC and ASCII data are not permitted in an exponent.

EXAMPLES: The following examples illustrate the use of arithmetic operators.

1. Following is a list of legal arithmetic expressions in JOVIAL.

II+2	2 * KK − 6
XX − YY	(MA *NA) / 2
AVAL / 2.0	ABS (MA * NA) / 2
AB (* 2 *)	2.0 * (U1+V1)
BB * CC	ABS (MA)

2. Following is a list of illegal arithmetic expressions.

Illegal Expression	Reason
− + AB + BB	Contiguous operators. Rewrite as −(+AB)+BB.
XX* − YY	Contiguous operators. Rewrite as XX* (−YY) or as −YY * XX.
(AB + BC) (EF +GH)	Implied multiplication. Rewrite (AB + BC)*(EF + GH).

3. Following is a list of exponents written in JOVIAL notation.

Conventional Exponentiation	JOVIAL Exponentiation
AA^{3}	AA(*3*)
BB^{-3}	BA(*−3*)
CA^{CB*CC}	CA(*CB*CC*)
$DA^{DB^{DC}}$	DA(*DB(*DC*)*)
$EA^{EB^{EC+ED}} + 3.0^{EE}$	EA(*EB(*EC+ED*)*)+3.0(*EE*)
$(GA + GB)^{2}$	(GA+GB) (*2*)

4.2 RULES OF PRECEDENCE

The order in which arithmetic operations are to be performed can be indicated by parentheses. If parentheses are nested, the innermost parentheses are evaluated first.

The following examples show how parenthesized expressions are evaluated.

Example 1:

```
3*(2+5)*(8/2)
=3*7*4
=84
```

Example 2:

```
3+(4*(7+8))+2
=3+(4*15)+2
=3+60+2
=65
```

When the precedence is not indicated by parentheses, the order is as follows:

First exponentiation, then multiplication and division, and addition and subtraction last.

The following example shows the evaluation of an expression not in parentheses.

Example 3:

```
2+10/5+3(*2(*2*)*)*4−6
=2+10/5+3(*4*)*4−6
=2+10/5+81*4−6
=2+2+324−6
=322
```

The following two examples show how parentheses affect exponentiation. In example 4, parentheses are not used, so exponentiation is evaluated from right to left. In example 5, parentheses indicate that the exponentiation is to be performed from left to right.

Example 4:

```
2(*2(*3*)*)
=2(*8*)
=246
```

Example 5:

```
(2(*2*))(*3*)
=4(*3*)
=64
```

Items on the same level are normally evaluated from left to right. However, right to left evaluation can occur, especially if function calls are involved. The following example shows a case where an incorrect answer can occur because the right-hand function call is performed before the left-hand call, and the null arguments therefore do not contain the expected values.

Example 6:

```
If CLCI(ADDR,4,4H(ABCD))*CLCI(,,4H(EFGH))EQ 0 $
```

4.1.2 Mixing Types of Data

Although it is more efficient to use only one type of data in an expression, integer, fixed-point, and/or floating-point data can be mixed.

Figure 4−1 shows the types of data obtained when types are mixed in an expression. Since only one operation at a time is performed, only two possible types of data can be involved at the same time.

Data Types Involved	Operators	Type of Result
integer and integer	+−*	integer
integer and integer	/	fixed-point
integer and fixed-point	+−*/	fixed-point
integer and floating-point	+−*/	floating-point
fixed-point and fixed-point	+−*/	fixed-point
fixed-point and floating-point	+−*/	floating-point
floating-point and floating-point	+−*/	floating-point

FIGURE 4−1. RESULT OF MIXING DATA TYPES

RESTRICTIONS: The following restrictions apply to arithmetic expressions containing different types of data.

1. ASCII and EBCDIC data are not permitted in expressions containing floating-point data and are treated as unsigned integer data in expressions containing integer and fixed-point data.
2. When EBCDIC or ASCII data is used in addition or subtraction with unsigned data, logical arithmetic is performed.
3. When EBCDIC or ASCII data is used in addition or subtraction with signed data, the low-order four bytes are used with the high-order bit treated as the sign.
4. When EBCDIC or ASCII data is used in multiplication or division, the four low-order bytes are used with the high-order bit treated as the sign.

NOTE

When fixed point and integer data are mixed, grouping the A-type items with parentheses will preserve fractional accuracy.

EXAMPLES: In the following examples, showing the result obtained when data types are mixed in an expression, RED is an integer item, YELLOW is a fixed-point, and BLUE is a floating-point item.

Expression	Type
YELLOW * BLUE	floating-point
RED + 10	integer
BLUE/ 2 *RED	floating-point
RED/ 2 − RED	fixed-point

Expression	Type
YELLOW/ 10.5A1	fixed-point
(YELLOW + RED) /3.0	floating-point
RED(*2*) * 100	integer
RED – YELLOW	fixed-point
BLUE/YELLOW + RED	floating-point

The following are general statements about the type of result obtained from an arithmetic expression containing different types of data.

1. If the expression contains any floating-point data, the result will be floating-point.
2. If the expression contains only integer data, and no division is involved, the result will be integer.
3. If the expression contains only fixed-point data, fixed-point and integer data, or only integer data and division is involved, the result will be fixed-point.

4.3 ALIGNMENT

Alignment is automatically supplied when arithmetic operations are performed.

If a result is too large to fit into an integer field, it is truncated.

A floating-point number must not exceed the limits $5 * 10^{-79}$ and $7 * 10^{75}$. It is the programmer's responsibility to ensure that the limits are not exceeded.

If a result is too large to fit into a fixed-point field, it is truncated according to the following rules:

1. In addition or subtraction, the fractional bits are truncated before the operation is performed, in order to preserve integer bits. The effect of truncating fractional bits is to always bring the result closer to zero.
2. In multiplication, the number of bits allowed for the product is 31 bits plus a sign bit. The first thing done is to sum the integer bits for the multiplier with the integer bits for the multiplicand. If this sum is 31 or greater, all fraction bits are truncated. Truncation occurs on the high order bits if the sum is greater than 31. If the sum is less than 31, the number of bits remaining can be used for the fraction.
3. In fixed-point division, scaling depends on the scale and size of the dividend, divisor, and quotient:
   a. If the quotient destination is an explicitly defined non-floating point field, scaling will conform to the definition of the quotient.
   b. Otherwise (e.g., the quotient is an intermediate result), the integer portion of the quotient will be the number of integer bits in the dividend plus the number of fractional bits in the divisor. This assignment allows for the largest possible integer result. If this is greater than 31 magnitude bits, high-order bits are truncated and a warning diagnostic is issued. If this assignment is less than 31 magnitude bits, remaining will be used for fractions.

Examples:

```
ITEM AA     A    10   S     5   $
ITEM BB     A    15   S     5   $
ITEM CC     A    32   S    25   $
ITEM XX

AA = AA + BB/CC $
```

Here the quotient is an intermediate result; there are 9 integer bits in the dividend and 25 fractional bits in the divisor. 9 + 25 = 34 > 31. The warning is issued, and only 31 bits are retained.

```
CC = CC – AA/BB $
```

The quotient is again intermediate; there are four integer bits in the dividend and five fractional bits in the divisor. 4 + 5 = 9 < 31. The remaining 22 bits are fractional.

When the dividend is the product of fixed point multiplication, its size and scale are the sum of the sizes and scales of the multiplicand.

Example:

```
CC = (CC * AA)/BB + XX $
```

Here there are 6 + 4 = 10 integer bits in the dividend (25 + 5 = 30 fractional bits) and 5 fractional bits in the divisor. 10 + 5 = 15 < 31. The remaining 16 bits are fractional.

4. To preserve fractional bit accuracy in mixed fixed point and integer arithmetic, the fixed point constants or items should be grouped together by means of parentheses.

5.0 OPERATIVE STATEMENTS

Figure 5−1 summarizes the purpose of JOVIAL operative statements. The START control statement is included in the list since it is required in each JOVIAL program and, unlike other control statements, it is recognized only by the JOVIAL compiler. The statements are separated into categories according to their function: control information, data manipulation, logical operations, sequence control, defined procedures, and direct code. The GOTO and STOP statements appear in two categories (sequence control and defined procedures) because they have two functions.

Category	Statement	Purpose
Control Information (control statement)	START (control statement)	Indicates the type of program and identifies the compool.
	TERM	Indicates the end of the program.
Data Manipulation	Assignment	Sets a variable equal to the value of another variable, a constant, or an arithmetic expression.
	Exchange	Exchanges the values of two items or table entries.
	REMQUO	Performs integer division; produces quotient and a real remainder.
Logical Operations	IF	Evaluates a condition to determine whether it is true or false.
	IFEITH/ORIF	Evaluates a series of alternative conditions.
Sequence Control	FOR	Permits controlled repeated execution of an operative statement.
	TEST	Causes a transfer to the end of the statement under control of a FOR statement.
	GOTO	Twofold—(1) Causes a transfer within a program to a specified label. (2) Tests a switch.
	STOP	Interrupts program execution.
Defined Procedures	PROC	Identifies the beginning of a function or procedure.
	CLOSE	Identifies the beginning of a closed compound procedure.
	Call	Calls procedures.
	GOTO	Calls closed programs and closed compound procedures.
	RETURN	Causes a return to the calling program from within a function, procedure, or closed compound procedure.
	STOP	Twofold—(1) Causes a return to the calling program from a closed program. (2) Relinquishes control to the monitor in a main program.
Direct Code	ASSIGN	Permits a direct code reference to JOVIAL-defined data.

FIGURE 5–1. PURPOSE OF OPERATIVE STATEMENTS

5.1 STATEMENT LABELS

The statement label provides a means of branching to a given statement from some other point in the program. The statement label is coded according to the same rules as a data definition. The label must be immediately followed by a period (.), and must precede the statement to which it applies. Most operative statements can be labelled.

Any reference to a statement label may include the period. Where it is clear what is meant, as in a GOTO, the period may be omitted. If there is any ambiguity, it must be included.

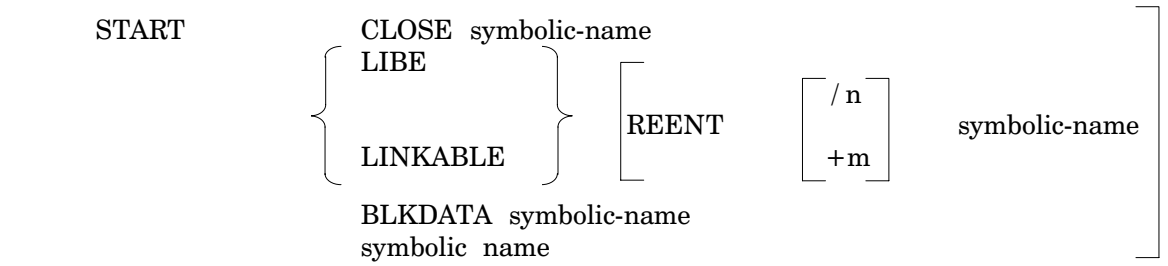
5.2 CONTROL INFORMATION

The START control statement must be the first statement in any JOVIAL program. It provides control information to the compiler. The TERM statement must be the last statement in any JOVIAL program.

5.2.1 START Control Statement

The START control statement provides information for controlling a JOVIAL program. It specifies the type of program, indicates which compool, if any, is to be used, specifies whether an object program is to be produced in spite of serious errors, and gives the load address of the program.

The format of the START control statement is:



CLOSE symbolic-name	Program is compiled as a closed program identified by a symbolic name.
LIBE symbolic-name	Program is compiled as a library routine identified by a symbolic name.
LINKABL symbolic-name	As "LIBE" but called by NAS SVC 104. (See NAS Monitor Handbook.)
REENT	If present, indicates that the program is to be compiled as a re-entrant routine.
/n	If present, with n = 1, 2, or 4, this indicates that the re-entrant storage requirement for the program will be "padded" by a factor of n/16 (rounded up to the next fullword boundary). n = 0 or ƀ has no effect, but may be coded here for compatibility purposes.
+m	If present, with m = 0 to 99, the reentrant storage requirement for the program is "padded" by 8m bytes. Use of +ƀ is permitted but has no effect.
BLKDATA	If present, indicates that the program is to be compiled as a Block-Data routine.
symbolic-name	Program is compiled as a main program identified by a symbolic name of 2–6 characters.

If none of these options is given, the program is compiled as a main program and has no identifying name.

POOL [‘compool-id] Indicates that the requested compool is to be used. A single quote-mark separates POOL from compool-id, and no blanks are permitted within the combined field. See restriction 6.

ASSEMBLE Indicates that an attempt is to be made to produce an object program in spite of errors of serious severity. This option is treated in more detail in Section 9 under “Compiler Diagnostics.”

WARNING

This option should be avoided unless absolutely necessary, as its use may cause the Compiler to execute a SYSDUMP.

load-address Indicates the starting location for loading the object program. It may be specified in either decimal or hexadecimal values. If decimal, format is an integer from 1 through 8 digits. If hexadecimal, format is X‘n’ where n represents 1 through 6 hexadecimal digits. If the load address is omitted, the loader determines the location of the object program.

remarks Descriptive information that has no effect on the compilation.

RESTRICTIONS: The following restrictions apply to the use of the START statement:

1. The fields in the START statement must appear in the order given and must be separated by at least one blank.
2. The fields must be contained within columns 1 through 66. There is no required column for the beginning of each field.
3. The START statement must be the first statement in any JOVIAL program, or a message indicating a serious error will be given.
4. No JOVIAL statement can appear on the START statement.
5. The difference between main programs, closed programs, Block-Data Programs, and library routines is explained in Section 6, “Defined Procedures.”
6. Under MVS, the ‘compool-id may be omitted and just POOL coded. However, if the ‘compool-id is included, it must match the low-order portion of the DSN on the CMPTAB DD statement. A mismatch will be considered a serious (or worse) error.
7. Instructions for creating and maintaining compools, explanations of control statement, and explanations of error messages are given in the IBM Data Processing System: Compool Edit User’s Manual (CMPEDT). MVS compools are covered in Section 10, JOVIAL Procedures.
8. The only fields on the START statement which are checked for validity are the symbolic-name field after the CLOSE, LIBE, or BLKDATA option, if one of these options is selected, and the Compool-ID field after POOL, if the option is selected. If any other field is encountered which cannot be classified, it is considered to be the beginning of the “remarks” field. No further checking on the START statement is done and no diagnostic is produced.

5.2.2 TERM Statement

The TERM statement identifies the end of a JOVIAL program and indicates where program execution is to begin. The format of a TERM statement is:

TERM[statement-label] $

statement-label — Label of the statement in the program with which execution is to begin. If this field is omitted, execution begins with the first operative statement.

RESTRICTIONS: The following restriction applies to the TERM statement.

A statement label in a TERM statement is permissible only in main programs.

5.3 DATA MANIPULATION

Data manipulation statements alter data or the arrangement of data. The three data manipulation statements are Assignment, Exchange, and REMQUO.

5.3.1 Assignment Statement

The Assignment statement is used to set a field to the value of an arithmetic expression.

The format of the Assignment statement is:

left-term = arithmetic-expression $

left-term — Any single variable of type 1 through 4 described in the section "Arithmetic Expressions," any such variable modified by BIT or BYTE, or a valid use of NENT or ENT modifiers.

arithmetic expression — An arithmetic expression.

RESTRICTIONS: The following restrictions apply to the use of the Assignment statement.

1. Data is stored in conformity with the description of the receiving field.
2. Decimal point alignment and truncation are performed, if necessary.
3. If EBCDIC or ASCII data is stored in a signed field and a 1-bit is moved into the sign position, subsequent references to the signed item will consider it to be a negative value.
4. When data is stored in an EBCDIC or ASCII doubleword field, the low-order 32 bits are put in the second word and the high-order bits are right-justified in the first word, preceded by leading zeros if necessary.
5. Caution should be exercised when the source and destination of an assignment statement are overlapping byte-aligned fields, with the destination to the right of the source. An MVC instruction may be generated which will "propagate" leading (non-overlapped) bytes of the source into the entire destination field. For example:

```
TABLE      TT        R   1     2                  $
  BEGIN
   ITEM    HH1       H   6     0    0   D         $
   ITEM    HH2       H   6     0   16   D         $
  END
H1 =  6H(ABCDEF) $
H2 =  HH1 $
```

will result in | A | B | A | B | A | B | A | B | , not in | A | B | A | B | C | D | E | F | . If this problem occurs, define an intermediate temporary item and code:

```
temp          =    source $
destination   =    temp $
```

EXAMPLES: The following examples show the uses of the Assignment statement.

Assume the following data declarations are given:

```
    ITEM COLOR $ V(RED) V(YELLOW) V(BLUE) $
    ITEM AAA I 32 U $
TABLE CONV R 30 N $

BEGIN
    ITEM FFF F $
    ITEM III I 32 U $
BEGIN 60 45 271 34 1278 ...END
END
```

Then the Assignment statement can be used for the following purposes.

1. To set an item to a constant value.

Statement	Comments
AAA = 29$	The value of Integer item AAA becomes 29.
FFF(AAA)=36.5 $	Item FFF in the last entry of Table CONV is set to 36.5.
COLOR=V(YELLOW)$	Item COLOR is set to 1. This is the only way a status value constant can appear in an Assignment statement.

2. To set an item to an arithmetic expression.

Statement	Comments
AAA = AAA – 1 $	The value of Item AAA is decremented from 29 to 28.
FFF(AAA) = III(AAA)*.60 $	Item FFF in the next to the last entry of Table CONV is set to .60 times the value of Items III in the same entry.

3. To perform data conversion.

Statement	Comments
FFF(0)= III(0) $	The integer 60 in Item III in the first entry of Table CONV is moved to Item FFF in the same entry and assumes the form of the floating-point number, 60.0.

5.3.2 Exchange Statement

The Exchange statement is used to exchange the values of two fields. It may also be used to exchange two table entries (for this second case, see Section 3, ENT modifier). The format of the Exchange statement is:

---

symbolic-name == symbolic-name $

---

symbolic-name — symbolic name of a single item, a table item, or an array element, subscripted if necessary.

RESTRICTIONS: The following restrictions apply to the use of the Exchange statement.

1. The data assumes the form of the receiving field.
2. Decimal point alignment and truncation are performed if necessary.
3. BIT and BYTE modifiers may not be used.
4. A blank must not be used between the two equal signs that form the exchange symbol.

EXAMPLE: In the following example of the use of the Exchange statement, the values of AAA and BBB are exchanged.

```
ITEM AAA F $

ITEM BBB I 32 S $

AAA == BBB $
```

The following list shows the effect of the Exchange statement on different possible values of AAA and BBB.

Before Exchange		After Exchange	
AAA	BBB	AAA	BBB
20.	30	30.	20
36.54	2781	2781.	37
.4	42	42.	0

5.3.3 REMQUO Statement

The REMQUO (remainder-quotient) statement is used to perform integer division and provide a quotient and a real remainder. The format of the REMQUO statement is:

REMQUO (dividend, divisor = quotient, remainder) $

---

dividend — An expression that is to be divided.

divisor — An expression that divides.

quotient — A variable that is the quotient of division.

remainder — A variable that is the remainder of the division.

RESTRICTIONS: The following restrictions apply to the REMQUO statement.

1. The quotient will have the appropriate algebraic sign. The remainder will have the sign of the dividend.
2. No parameters may be omitted.

EXAMPLE: The following example shows the use of the REMQUO statement.

```
ITEM DVDND I 32 S $
ITEM DVSR I 32 S $
ITEM QUOT I 32 S $
ITEM RMDR I 32 S $
REMQUO (DVDND, DVSR = QUOT, REMDR) $
```

The following list gives the quotient and remainder resulting from possible dividends and divisors.

DVND	DVSR	QUOT	RMDR
+38	−4	−9	+2
+7	+7	+1	+0
−16	+3	−5	−1

5.4 LOGICAL OPERATIONS

The IF statement is used to specify logical operations. It causes a condition to be evaluated to determine whether it is true or false. The condition is true or false depending upon the value of the expressions when the IF statement is executed.

5.4.1 IF Statement

The IF statement tests a simple or complex condition. When the condition is true, the first operative statement following the IF statement is executed. When the condition is false, the second operative statement after the IF statement is executed. The first statement following the IF statement may be simple or compound. A simple statement is a single operative statement. A compound statement is a series of operative statements enclosed in BEGIN−END brackets.

The format of an IF statement is:

```
IF simple-condition      $
   complex-condition
```

simple-condition — Two expressions joined by a relational operator (see “Conditions”, following).

complex-condition Two simple or complex conditions joined by a logical operator (see "Conditions").

RESTRICTIONS: The following restrictions apply to the use of the IF statement.

1. If a compound statement follows an IF statement, the last statement in the compound statement (the statement immediately preceding the END bracket) cannot be an IF statement.
2. Compound statements can be nested.
3. If a compound statement is labeled, the label can precede or follow the BEGIN bracket. Statements within compound statements can be labeled.
4. A FOR statement cannot be the first statement after the IF statement (the true exit) unless the FOR is in a compound statement.
5. A CLOSE or PROC statement cannot be the first or second statement after the IF statement (true or false exit).

5.4.1.1 Conditions. An IF statement can evaluate either simple or complex conditions.

1. A simple condition has the form:

---

expression relational-operator expression

---

expression Arithmetic expression.

relational-operator One of the following:

{ EQ } / { = }	equal to
NQ	not equal to
GR	greater than
GQ	greater than or equal to
LS	less than
LQ	less than or equal to

2. A complex condition has the form:

{ simple-condition / complex-condition } logical-operator { simple-condition / complex-condition }

---

simple-condition A simple condition.

complex-condition A complex condition.

logical-operator One of the following:

AND If both conditions are true, the condition is true; otherwise it is false.

OR If neither condition is true, the condition is false; otherwise it is true.

RESTRICTIONS: The following restrictions apply to conditions in an IF statement.

1. If a status value variable is compared to one of its status value constants, the collating sequence depends on the item statement defining the status variable, because status value constants are assigned the values 0, 1, 2,... in the order of their appearance in the item statement defining the status value variable.

2. If a status value variable is compared to another status value variable, it is the integer values that are compared.

EXAMPLE: The following examples show the use of the IF statement with simple and complex conditions.

1. IF statement with simple conditions.

Statement	Comments
IF FOOD EQ ENERGY $	The items FOOD and ENERGY are compared.
GOTO DINNER $	True exit. If FOOD = ENERGY, transfer to statement labeled DINNER.

Statement	Comments
POUNDS = FOOD $	False exit. If FOOD ≠ ENERGY, set the item POUNDS to the current values of FOOD.
GOTO DIET $	Transfer to statement labeled DIET.

2. IF statement with simple conditions.

Statement	Comments
IF AA EQ BB OR BB NQ CC $	AA is compared to BB and BB is compared to CC.
GOTO ACCEPT $	True exit. If either AA = BB or BB ≠ CC, a transfer is made to ACCEPT.
GOTO REJECT $	False exit. If both AA ≠ BB and BB = CC, a transfer is made to REJECT.

3. Typical simple conditions.

```
COLOR EQ V(RED)
AA GR BB
3* CCC NQ DD + EEE
AAA(*2*) – BBB(*2*) LS CCC(*2*)
```

4. Typical complex conditions

The condition AAA – BBB LQ CCC AND AAA + DDD GR EEE is true only if AAA – BBB is less than or equal to CCC and AAA + DDD is greater than EEE. It is false if either simple condition is false.

The condition AAA GR BBB OR CCC GQ DDD is true if either AAA is greater than BBB or CCC is greater than or equal to DDD. It is false only if both simple conditions are false.

5. IF statement followed by a compound statement.

```
IF AA EQ V(Z) $
BEGIN BB = 0 $
GOTO XX $
END
BB = 1 $
```

If AA has the status value Z, Item BB is set equal to 0 and a transfer is made to the statement labeled XX.

If AA does not have the status value Z, then BB is set equal to 1 and the next sequential instruction is executed as usual.

5.4.1.2 Abbreating Complex Conditions. Complex conditions may contain a certain amount of redundancy (immediate repetition of an expression, which consists of up to six terms).

AAA GR BBB AND BBB LS CCC + 2 AND CCC + 2 EQ DDD

If the identical expressions are separated only by the logical operator AND, the AND and the redundant expression can be omitted without changing the meaning of the condition.

For example, the preceding condition can be abbreviated as:

AAA GR BBB LS CCC + 2 EQ DDD

RESTRICTIONS: The following restrictions apply to abbreviating complex conditions:

1. The redundant expressions must be separated by the operator AND.
2. If one expression is abbreviated in a condition, all the permissible terms must be abbreviated.

5.4.1.3 The NOT Operator. The operator NOT is used to complement conditions, relational and logical operators and itself. It must not immediately precede a relational or logical operator; it can precede another NOT.

The following list shows the effect of NOT on relational and logical operators.

Complementation	Effect
NOT...EQ	NQ
NOT...NQ	EQ
NOT...GR	LQ
NOT...LQ	GR
NOT...GQ	LS
NOT...LS	GQ
NOT...AND	OR
NOT...OR	AND
NOT...NOT	Cancelled

The following examples show the effect of adding the operator NOT to a simple condition.

1. NOT AAA LS BBB is equivalent to AAA GQ BBB.

2. NOT NOT AAA EQ BBB is equivalent to AAA EQ BBB.

The operator NOT applies only to the condition following it unless several conditions and logical operators following NOT are enclosed in parentheses. If an expression in parentheses follows a NOT, the entire expression is complemented. The operators AND or OR are complemented only when parentheses enclose the conditions they connect.

The following examples show the effect of the operator NOT on complex conditions.

1. NOT AA EQ BB AND CC EQ DD is equivalent to AA NQ BB AND CC EQ DD.

2. NOT (AA EQ BB AND CC EQ DD) is equivalent to AA NQ BB OR CC NQ DD.

3. AA EQ BB AND NOT CC EQ DD is equivalent to AA EQ BB AND CC NQ DD.

4. NOT AA EQ BB OR CC EQ DD AND EE EQ FF is equivalent to AA NQ BB OR CC EQ DD AND EE EQ FF.

5. NOT (AA EQ BB OR CC EQ DD AND EE EQ FF) is equivalent to AA NQ BB AND (CC NQ DD OR EE NQ FF).

6. AA EQ BB OR NOT CC EQ DD AND EE EQ FF is equivalent to AA EQ BB OR CC NQ DD AND EE EQ FF.

When the operator NOT is used with an abbreviated complex condition, it must precede the entire condition. For example:

NOT(AA EQ BB NQ CC) is equivalent to AA NQ BB OR BB EQ CC.

5.4.1.4 Evaluation of Conditions. The following steps are taken to perform an evaluation of a complex condition.

1. Determine values of arithmetic expression.

2. Perform NOT complementation.

3. Evaluate simple conditions; true or false.

4. Evaluate logical operations. If parentheses are used, the evaluation is from the innermost parentheses outward. If both AND and OR are used, AND takes precedence.

5. JOVIAL will generate algebraic, as opposed to logical, comparisons if any arithmetic term is found in an IF statement.

Given the information that AA = AB = AC = 2, the evaluation of a condition is:

1. AA EQ 2 AND (AB NQ 2 AND NOT (AC+3 GR AB−5 AND NOT (AB EQ AC OR AC GQ 4))).

2. 2 EQ 2 AND (2 NQ 2 AND NOT (5 GR −3 AND NOT(2 EQ 2 OR 2 GQ 4))).

3. 2 EQ 2 AND (2 NQ 2 AND(5 LQ −3 OR (2 EQ 2 OR 2 GQ 4))).

4. TRUE AND (FALSE AND (FALSE OR (TRUE OR FALSE))).

5. TRUE AND (FALSE AND (FALSE OR TRUE)).

6. TRUE AND (FALSE AND TRUE).

7. TRUE AND FALSE.

8. FALSE.

Although the programmer performs these steps to ensure that his condition is logically constructed, the actual evaluation can be performed more quickly. For example, for the same statement, the compiler would generate code to perform the following operations.

1. If AA is not equal to 2, the condition is false; otherwise go to step 2.

2. If AB is equal to 2, the condition is false; otherwise go to step 3.

3. If AC + 3 is less than or equal to AB − 5, the condition is true; otherwise go to step 4.

4. If AB is equal to AC, the condition is true; otherwise go to step 5.

5. If AC is less than 4, the condition is false; otherwise the condition is true.

5.4.2 IFEITH/ORIF Statements

The IFEITH/ORIF statements are used together to specify a sequence of logical operations such that at most one of a series of alternative operations is performed. A series of conditions is tested for true or false values in sequence. When a false condition is encountered, the associated operation is skipped and the next condition is tested. The first time a true condition is found, the associated operation is performed, and then all following conditions and operations are skipped. It is possible that all conditions will be false and no operations will be performed. It is also possible to code a final condition which is trivially true, thus making the last operation an "ELSE" clause.

The format of the IFEITH/ORIF statement is:

```
IFEITH        condition        $
    simple-or-compound-statement
[ ORIF        condition        $ ]
[ simple-or-compound-statement ]
[ .                            ]
[ .                            ]
[ .                            ]
[ ORIF        condition        $ ]
simple-or-compound-statement
ORIF        { condition }      $
            {     1     }
simple-or-compound-statement
END
```

---

condition — Any simple or complex condition that may be code in an IF statement.

simple-or-compound-statement — Any simple statement, or group of statements enclosed in BEGIN−END brackets.

1 — Trivially true condition — must be coded exactly as shown.

RESTRICTIONS: The following restrictions apply to use of the IFEITH/ORIF statements:

1. At least one ORIF must follow an IFEITH.
2. Use of an ORIF anywhere else in a program is prohibited.
3. A final END must terminate the IFEITH/ORIF structure. There is no matching BEGIN for this END.

5.5 SEQUENCE CONTROL

Normally, operative statements are executed in the order in which they were written. The sequence control statements are used to change the normal order of statement execution. The sequence control statements are FOR, TEST, GOTO, and STOP.

5.5.1 FOR Statement

The FOR statement causes the statement that follows it (called the range of the FOR) to be repeatedly executed a specified number of times.

The range of the FOR can be a single statement or a compound statement. A compound statement is a sequence of statements enclosed in BEGIN and END brackets.

The number of times the range of the FOR is executed depends upon an index defined in the FOR statement. The index can be given a single value, indicating that the range is executed once; or it may be given an initial value and an increment, indicating that each time the range is executed, the index is incremented; or it may be given an initial value, an increment, and a maximum, indicating that each time the range is executed, the index is incremented until it passes the maximum and control passes beyond the range of the FOR.

The format of a FOR statement is:

```
FOR index = start [ ,step | ,step, max ] $
```

index — Single alphabetic letter. It represents a 24-bit unsigned integer value and is defined only in the FOR statement, never in an ITEM statement.

start — Initial value of index. Must be an arithmetic expression, representing a positive integer.

step — Value by which the index is incremented each time the range of the FOR has been executed. It must be an arithmetic expression, representing an integer.

max — Limit of the index. When the index exceeds the maximum (in the positive direction, if the increment is positive; in the negative direction, if the increment is negative), control passes beyond the range of the FOR. The maximum value must be an arithmetic expression representing a positive integer.

RESTRICTIONS: The following restrictions apply to the use of the FOR statement.

1. The initial value and maximum value of an index must be positive integers. The increment must be an integer, but it can be either positive or negative. If non-integers are used, they will be converted to integer.

2. The index is defined only within the range of the FOR.
3. The values of any variable in the expression defining the increment or the maximum can be changed in the range of the FOR.
4. A compound statement can be within another compound statement.
5. If the range of the FOR is a compound statement, and only an initial value was assigned to the index, the last statement in the range (the statement preceding the END bracket) must not be an IF statement.
6. If the range of the FOR is a compound statement, and an increment or an increment and a maximum were assigned to the index, and the last statement of the compound statement is an IF, a true condition causes incrementing and testing of the index and a false condition causes control to pass beyond the range of the FOR.
7. If a compound statement is labeled, the label may be either before or after the BEGIN bracket. Statements in the compound statement can be labeled.
8. A 'NENT modifier may be used when processing variable-length tables to ensure that all the entries in the table are processed. It is useful for processing fixed-length tables in case the number of entries in the table changes from compilation to compilation. Since NENT represents the number of entries, NENT minus 1 represents the last entry (entry counting begins with 0).

EXAMPLE: The following examples show the use of the FOR statement.

1. Index has only an initial value

Statement	Comments
FOR K = 1 $	Index K is assigned only one value, range of FOR executed only once.
TOP = K $	Item TOP is set equal to 1.
BUTT = K + BUTT $	Error. Outside range of FOR, K is not defined.

2. Index has initial value and increment.

Statement	Comments
FOR C = 1, 3 $	Index C is assigned initial value 1 and is incremented by 3 each time range is executed.
BEGIN	Indicated range is compound statement.

Statement	Comments
AREA(C) = 1/2 * (BASE(C)*ALT(C))$	Table Item AREA in entry C is set equal to half the product of table Items BASE and ALT in entry C.
IF C LS NENT (TABL) $	If C is less than the number of entries in TABL, C is incremented and the range of the FOR is executed again. If C is greater than or equal to the number of entries in TABL, control passes to the next sequential instruction.
END	

3. Index has initial value, increment, and maximum. Item IN is a table item in Table TABL.

Statement	Comments
FOR A = 0, 1, NENT (TABL) – 1 $	Index A has initial value 0, is incremented by 1 each time range is executed, and will not exceed the number of entries in TABL.
IN(A) = A + OUT $	Range of FOR is a single statement setting the values of IN.
AAA == BBB $	Exchange statement that will be executed only after Item IN in each entry of TABL has been processed.

4. Index has a negative increment.

Statement	Comments
FOR K=15, –3, 2 $	Index K has initial value 15, an increment of –3, and if index becomes less than 2, the range will not be executed.
BEGIN	Indicates compound statement as range.
HIGH(K) = CLOUD $	Assignment statement.
RAIN(K) = FAIR – SUN $	Assignment statement.
END	Indicates end of range.
FOR...	Statement outside range. Will be executed only after range executed for K = 15, 12, 9, 6, and 3.

5.5.1.1 Multiple FOR Statements. If several indexes are to be used in the range of a FOR statement, a list of consecutive FOR statements, each identifying one index, can be used. The range of all of these FOR statements is the single statement or compound statement following the last FOR in the list.

Initial values and increments can be given for each of the FOR statements in the list. Each time the range is executed, the indexes are incremented the specified amount. A maximum for an index is significant only for the first FOR statement in the list. When the maximum for this index is exceeded, control passes to the next operative statement beyond the range of the FOR statements. The maximum values for the other indexes are ignored. When incrementing the indexes in this common range, the last declared index is incremented first.

EXAMPLE: In the following example showing the use of multiple FOR statements, CLOCK and HAND are table items and HOUR is a single item.

Statement	Comments
FOR J=0, 1, 9 $	First FOR statement; gives maximum.
FOR K=9, –1 $	FOR statement with negative increments.

Statement	Comments
BEGIN	Indicates range is compound statement.
CLOCK(J)=HAND (K) $	Assignment statement.
HOUR = HOUR + CLOCK (J) $	Assignment statement.
END	End of range of FOR statements.
IF...	Outside range of FOR statements. Executed only after the range of the FOR statements is executed 10 times.

RESTRICTIONS: The following restriction applies to the use of multiple FOR statements. A statement label may appear only on the first FOR of the group. Use of a statement label on any other FOR in the group will be flagged as a serious error.

5.5.1.2 Nested FOR Statements. Nested FOR statements are used to form a loop within a loop. A FOR statement is nested if it is within the range of another FOR statement. Nested FOR statements are particularly useful for processing arrays as shown in the following example.

```
ARRAY CLASS 10 5 2 I 6 U $

FOR I = 0, 1, 1 $

BEGIN FOR J = 0, 1, 4 $

BEGIN FOR K = 0, 1, 9 $

BEGIN IF CLASS($K, J, I $) GQ 100 $

GOTO AA $ CLASS ($K, J, I $)= 100 $ TEST $

AA. CLASS($K, J, I $) = 0 $

END END END
```

1. Array CLASS has three dimensions whose sizes are 10, 5, and 2, respectively.
2. Initially all three indexes are set to 0. Each times K goes through a complete cycle (0 through 9), J is incremented by 1. Each time J goes through a complete cycle (0 through 4), I is incremented by 1. By the time the maximum for I is passed, the IF statement has been evaluated 100 times.

5.5.2 ALL Modifier

The ALL modifier is used only with FOR statements. With the use of ALL an entire table is processed, entry-by-entry, starting with entry 0. The ALL modifier replaces the operands in the FOR statement that specify the initial value, increment, and maximum value of the index.

For example, the statement

```
FOR A = ALL(TABL) $
```

is equivalent to the statement

```
FOR A = 0, 1, NENT(TABL) – 1 $
```

5.5.3 TEST Statement

When the range of a FOR statement is a compound statement, a TEST statement may be included to cause a transfer to the end of the range, where incrementing and testing of the index takes place. Often, a TEST statement is used as either the true or false exit of an IF statement. An example of other occurrences of the TEST statement is given in the discussion of the GOTO statement. See example 2, following.

An alternative use of the TEST statement is to exit from the range of the loop by transferring control to the statement following the END of the compound statement. This is done by using the EXIT option.

An index can be specified in the TEST statement. In the range of multiple FOR statements, the TEST statement with an index specified causes control to be transferred to the end of the common range; all indexes in the range, up to and including the specified index, will be incremented and tested. In the range of nested FOR statements, the TEST statement with an index specified causes control to be transferred to the end of the range controlled by the specified index. See example 3, following.

The format of the TEST statement is:

```
TEST [EXIT] [index] $
```

---

index	Name of an index defined in a FOR statement whose range includes the TEST statement.
EXIT	Specifies transfer out of loop.

RESTRICTIONS: The following restrictions apply to the TEST statement.

1. If a TEST statement appears in the range of nested FOR statements and no index is specified, a transfer is made to the end of the range controlled by the last index defined before the TEST statement appeared.

2. If a TEST statement appears in the range of multiple FOR statements and no index is specified, a transfer is made to the end of the common range and all indexes are incremented just as if the end of the range had been reached as usual.

EXAMPLES: The following examples show some uses of the TEST statement.

1. TEST statement with no index specified.

```
FOR I = 0, 1, 9 $

BEGIN IF ELBA($I$) EQ 0 $

TEST $

ELBA($I$) = ELBA($I$) (*–2*) $

I = I + I $

END
```

The TEST statement is used as the true exit of the IF statement. The two Assignment statements following the TEST statement are bypassed when the IF statement is true.

2. TEST statement in range of nested FOR statements.

```
FOR I = 0, 10, 99 $
BEGIN FOR J = I, 1, I + 9 $ BEGIN
IF AA($I$) EQ BB($I$) $
TEST $
CC($J$) = AA($I$)$
END
DD($I$) = AA($I$)$
END
```

Although no index is specified in the TEST statement, the statement is preceded by the FOR statement defining index J, so the effect is the same as though J appeared in the TEST statement. In order to transfer control to the end of the I-controlled range, the index I would have to be specified in the TEST statement.

3. TEST statement in range of multiple FOR statements.

```
FOR L = 0, 10, 49 $
FOR K = 2, 3 $
FOR M = ALPHA/2, −1, 1 $
BEGIN statement 1 $
statement 2 $
TEST K $
statement 3 $
END
```

When TEST K is encountered, a transfer is made to the end of the range, index K and index L are incremented, and index L is tested. Index M is left unchanged.

5.5.4 GOTO Statement

The GOTO statement is used to cause an unconditional transfer within a program, or to test an item or subscript switch, or to call an internal CLOSE or a separately compiled CLOSEd program.

The format of the GOTO statement is:

```
                  ⎧ close-name                          ⎫
GOTO              ⎪ close-program-name  [(=left-term)]  ⎪      $
                  ⎨                                     ⎬
                  ⎪ statement-label                     ⎪
                  ⎩ switch-name  [($subscript$)]        ⎭
```

close-name — Symbolic name on CLOSE statement.

closed-program-name — Symbolic name on START statement of program compiled as a CLOSEd program.

Statement-label	Symbolic name identifying the operative statement to which control is transferred.
switch-name	The name of the switch to be tested. The name must be subscripted if it is a subscript switch or if it is an item switch and the related item is defined in a table.
left-term	Item to be set with return code. This is valid only under MVS, and only on a call to a separately compiled CLOSEd program.

5.5.4.1 Unconditional Transfers. If the object of a GOTO statement is a statement label, control is transferred to the statement with the specified label.

A statement label is a symbolic-name composed of two six characters, the first of which must be alphabetic. When the label precedes a statement, it must be terminated with a period. When the label is used in a GOTO statement, the period is omitted.

A GOTO statement can be used to effect a transfer to any labeled operative statement in the same program region. Since a simple statement within a compound statement can be labeled, a transfer can be made into the middle of a FOR range or into the middle of a true or false exit of an IF statement. Transferring into the middle of a FOR range may leave the index undefined, however.

A GOTO may not be used to transfer control to another program region. Any procedure or function (PROC) is a region, and the portion of the program not in any PROC is a separate region. This means that it is not possible to use a GOTO to return from a PROC to a calling program statement label. A method for accomplishing this is given under Procedures in Section 6.

5.5.5 SWITCH Statement

A SWITCH statement is tested by a GOTO operative statement, and indicates a transfer to one of several locations depending upon the value of a specified item or subscript when the switch is tested. The SWITCH statement lists possible values of the item or subscript and corresponding labels to which transfer is made if the item or subscript has one of these values when the switch is tested.

5.5.5.1 Item Switches. Three statements are needed to define and test an item switch.

1. The item must be defined in a compiler-allocated ITEM statement, a programmer-allocated ITEM statement, or a single ITEM statement.

2. A SWITCH statement must be given to relate the item to a switch name and the values of the item to statement labels.

3. A GOTO statement with the switch name as its object must be given to test the switch. If the item has one of the values specified in the SWITCH statement, a transfer is made to the corresponding statement label. If the item does not have any of the specified values, control passes to the statement following the GOTO statement.

The format of an item SWITCH statement is:

```
SWITCH switch-name (item-name) = (value = statement-label
[, value = statement-label]...),
[, else-statement-label] $
```

switch-name	Programmer-assigned symbolic name used to identify the switch.
item-name	Name of a previously defined single item or table item.
value	A possible value of the item. Note that constants must be of the same type as the item and especially that status value constants are required for status items.

statement-label — Label of an operative statement or a closed-compound-procedure-name to which a transfer is to be made if the item has the corresponding value when the switch is tested. Using MVS JOVIAL only, the statement-label may also be a procedure with no arguments or a closed-program-name.

else-statement-label — As "statement-label", but specifying a transfer to be made when none of the conditions specified in the switch are met.

RESTRICTIONS: The following restrictions apply to item switches.

1. Because a SWITCH statement is declarative, it can appear anywhere in the program and not interrupt program flow. It need not be before the GOTO statement testing it, but it must be in the same region of the program. A region is a function, or a procedure, or a program excluding functions and procedures. Functions and procedures are explained in Section 3, "Defined Procedures."

2. The ITEM statement that defines the item whose values control the switch must precede the GOTO statement that tests the switch and the SWITCH statement.

3. All values in the statement must be unique.

4. A Procedure Dummy Statement Label (appearing as a formal input parameter in a PROC statement) may not be referenced by a SWITCH.

5. If a closed-compound-procedure-name is specified in a SWITCH, return from the closed-compound procedure will be to the statement following the GOTO that invoked the SWITCH.

6. Transfer to a closed-program-name will be allowed only if the name is declared by:

   CLOSE closed-program-name EXTRN $

EXAMPLE: The following program shows the statements required to define and test an item switch.

```
START
TABLE SET V40 N $
  BEGIN
    ITEM JUDGE I 5 U $
    ITEM VALUE S V(NONE) V(GOOD) V(BETTER) V(BEST) $
  END
FOR F = ALL (JUDGE) $
BEGIN
GOTO RATE ($F$) $
VALUE ($F$) = V(NONE) $ TEST $
AA. VALUE ($F$) = V(GOOD) $ TEST $
BB. VALUE ($F$) = V(BETTER) $ TEST $
CC. VALUE ($F$) = V(BEST) $
END
```

```
STOP $

SWITCH RATE (JUDGE) = (1 = AA, 2 = BB, 4 = CC, 3 = AA) $

TERM $
```

1. Table SET has two items in each entry. Item JUDGE is an integer field, and item VALUE is a status value field.

2. The first statement in the range of the FOR controlled by index F tests the item switch RATE. If Item JUDGE associated with switch RATE has a value of 1, 2, 3, or 4, a transfer is made to the corresponding statement label. For any other value of JUDGE, the statement immediately following the GOTO statement is executed.

3. Note that RATE must be subscripted because it is an item switch for Item JUDGE, which is in a table.

4. Note that TEST statements cause transfer of control to the end of the range when the status value has been set.

The following list shows possible values for JUDGE and the corresponding value for VALUE.

When JUDGE equal	Then VALUE equal
4	BEST
5	NONE
2	BETTER
1	GOOD
3	GOOD
0	NONE

5.5.5.2 Subscript Switches. A subscript switch is similar to an item switch except that a transfer is made to a specified statement depending upon the value of a subscript rather than the value of an item.

A statement label must be specified, either implicitly or explicitly, for each consecutive value of the subscript. Usually, the subscript is an index, but it may be any arithmetic expression.

The following statements must be given to define and test a subscript switch.

1. A SWITCH statement must be given to relate the switch name to the subscript and to specify statements to which transfer is to be made as the subscript assumes consecutive values. Either a statement label or a null field (indicated by two consecutive commas or by an opening parenthesis and a comma) must be given for each value of the subscript.

2. A GOTO statement with the switch name as the object must be given to test the switch. If the value of the subscript exceeds the number of fields in the SWITCH statement, the effect is as though the statement were filled out with null fields.

The format of the subscript SWITCH statement is:

```
SWITCH switch-name = (statement-label[,[statement-label]
```

]...)[,else-statement-label]$

switch-name	Programmer-assigned symbolic name used to identify the switch.
statement-label	Label of statement or a closed-compound-procedure name to which transfer is to be made for consecutive values of the subscript. First label for first value of subscript, second label for second value, etc. If a field is null, the transfer is to the else-statement-label or to the next sequential operative statement following the GOTO statement. Using MVS JOVIAL only, the statement-label may also be a procedure with no arguments or a closed-program-name.
else-statement-label	As "statement-label", but specifying a transfer to be made if the input value corresponds to a null field or is outside the range of explicitly coded fields.

RESTRICTIONS: The following restrictions apply to the use of subscript switches.

1. The SWITCH statement need not precede the GOTO statement that tests it, but it must be in the same region of the program.
2. A procedure Dummy Statement Label (appearing as a formal input parameter in a PROC statement) may not be referenced by a SWITCH statement.
3. If a closed-compound-procedure-name is specified in a switch, return from the closed-compound-procedure will be to the statement following the GOTO which invoked the SWITCH.
4. Transfer to a closed-program-name will be allowed only if the name is declared by:

   CLOSE closed-program-name EXTRN $

EXAMPLE: The following example gives a problem and a solution using subscript switches.

Given Table PLACE consisting of 40 entries with each entry containing one item:

1. Double the contents of the item in entries 1, 3, 7, 11, 13, 17, 21, 23, 27, 31, 33, and 37.
2. Square the contents of the item in entries 4, 6, 14, 16, 24, 26, 34, and 36.
3. Extract the cube root of the remaining values of PLACE.

Solution:

```
FOR K = 0, 10, 39, $
BEGIN FOR I = K, 1, K + 9 $
BEGIN GOTO CNTSW ($I−K$)$
PLACE ($I$) = PLACE($I$) (*1/3*) $ TEST $
SQ. PLACE($I$)=PLACE($I$) (*2*) $ TEST $
DOUBL. PLACE ($I$)=PLACE ($I$) *2$
END
END
SWITCH CNTSW=(,DOUBL,,DOUBL,SQ,,SQ,DOUBL,,,)$
```

Note that the last three commas before the closing parenthesis could be omitted, and the effect would be the same.

5.5.6 STOP Statement

The STOP statement is used to interrupt program execution. In a main program, if no statement label is given in the STOP statement, the compiler generates the symbolic instruction SVC SYSEOJ. When executed, this instruction causes a supervisor call interrupt which is interpreted by MVS as "end-of-job".

If a statement label is given, the compiler generates the symbolic instruction SVC SYSWAT. When executed, this instruction causes a supervisor call interrupt which is interpreted by MVS as a request to type "PROGRAM WAITING" on the operator's KVDT, wait for the operator to type "CONT", and then return to the program. Generated code following the SVC then transfers control to the statement label given in the STOP statement.

Within a closed program, a STOP statement returns control to the calling program as discussed in the section "Defined Procedures". STOP statement that are within closed program may not include statement labels.

The format of the STOP statement is

```
STOP [ (expression)     ]  $
     [ statement-label  ]
```

statement-label	The statement-label with which the program resumes if the operator restarts execution at the console typewriter.
expression	Return code to be passed to the caller. This is valid only under MVS, and only within a CLOSEd program, If any STOP statements in a CLOSEd program have return codes, all other STOP statements (without return codes) in the same program will have a return code of zero implied.

6.0 DEFINED PROCEDURES

A JOVIAL program can be segmented through the use of defined procedures. A defined procedure is a block of code that is located out of the line of flow of the program that executes it. It consists of operative statements and possibly data declarations. A call is issued to a defined procedure; the defined procedure is executed, and control returns to the point of departure from the calling program.

Closed compound procedures, functions, procedures, closed programs, and library routines are the five kinds of defined procedures. They differ in the method in which they are called and in the provision for data communication. A Block-Data program is not strictly a defined procedure, but is described in this section because it is a special type of compilation.

Closed programs and library routines can be compiled by themselves, but are executed only if called by another program. Closed-compound procedures, functions, and procedures must be compiled with the program that calls them.

A closed program is a complete JOVIAL program containing data declaration statements and operative statements. Data communication between a closed program and the calling program is through a compool.

A library routine is a function or a procedure stored on the library tape. Data communication is the same as for functions and procedures.

Functions and procedures contain data declarations and operative statements. Data communication is through parameters. Data declarations must be given for input and output parameters and for all data used in the function or procedure and not defined in the main program region or in compool.

Closed-compound procedures contain only operative statements. They can refer to any data defined for the region containing the closed-compound procedure.

6.1 CLOSED–COMPOUND PROCEDURES

A closed-compound procedure is a closed subroutine that is called from the region of the program in which is is defined. No parameters are passed to the closed-compound procedure.

A closed-compound procedure consists of a CLOSE statement, giving the name of the procedure, followed by operative statements. The call to a closed compound procedure is a GOTO statement with the closed-compound-procedure-name as its object.

6.1.1 Form of Closed-Compound Procedure

A closed-compound procedure consists of the following two parts:

1. A CLOSE statement giving the name of the closed-compound procedure.
2. A body consisting of operative statements enclosed in BEGIN and END brackets. During execution, when the END bracket is reached, control returns to the statement that followed the GOTO statement which called the closed-compound procedure. A RETURN statement can be used to cause a return before the END bracket is reached.

6.1.1.1 CLOSE Statement. The form of the CLOSE statement is:

```
CLOSE close-compound-procedure-name $
```

Closed-compound-procedure-name — Programmer-assigned symbolic name used to refer to the closed-compound procedure.

RESTRICTIONS: The following restrictions apply to closed-procedures.

1. A closed-compound procedure must not interrupt program flow. A GOTO, STOP, TEST, or RETURN statement should be precede a closed-compound procedure so that it is not executed unless it is called.
2. A closed-compound procedure must not contain functions or procedures.
3. A closed-compound procedure may be enclosed in BEGIN and END brackets. This means that it can be included in the range of a FOR statement.
4. A closed-compound procedure can refer to any data names or statement labels defined for the region in which it is defined.

6.1.2 Form of Call to Closed-Compound Procedure

A closed-compound procedure is called by a GOTO statement.

The form of the GOTO statement is:

GOTO closed-compound-procedure-name $

---

closed-compound-procedure-name — Symbolic name given in the CLOSE statement.

RESTRICTIONS: The following restrictions apply to calls made to closed-compound procedures.

1. A closed-compound procedure can be entered only if it is called by a GOTO statement.
2. Return from a closed-compound procedure is to the statement after the GOTO statement.
3. A call to a closed-compound procedure may not be made within the procedure itself.

6.1.3 Example of a Closed-Compound Procedure

The following example shows a main program that contains a closed-compound procedure.

Main Program	Comments
START MAIN	Beginning of main program named MAIN.
TABLE TRIP V 100 $	Table statement.
BEGIN	Beginning of definition of entry format.
ITEM TIME A 16 U 4 $	Item statement.
ITEM RATE I 8 U $	Item statement.
ITEM DIST A 24 U 4 $	Item statement.
END	End of definition of entry format.
FOR A = ALL (DIST) $	FOR statement.

Main Program	Comments
BEGIN	Beginning of compound FOR range.
IF RATE (A) GR 20 $	IF statement.
GOTO REKON $	True exit. Transfer to closed-compound procedure.
TIME (A) = TIME (A) * 60 $	Assignment statement.
IF TIME (A) GR 120 $	IF statement.
GOTO REKON $	True exit. Transfer to closed-compound procedure.
TEST $	False exit. Transfer to end of range of FOR.
CLOSE REKON $	Beginning of closed-compound procedure.
BEGIN	Begin body of closed-compound procedure.
DIST(A)=RATE(A) *TIME(A) $	Assignment statement.
END	End of body of closed-compound procedure.
END	End range of FOR.
STOP$	
TERM$	End program MAIN.

6.2 FUNCTIONS

Functions are defined procedures that produce one value each time they are called. The function call, which is the name of the function followed by input parameters enclosed in parentheses, is one operand of an arithmetic expression. In processing the expression, the function is executed when its value is required. (See “Rules of Precedence”.)

For example, assume SIN is a function that can receive one input parameter. Further assume that the statement IF SIN(A) EQ RAD is given. When the IF statement is encountered, the function SIN is called with A as the input parameter. The function is executed producing a single value to be compared to the Item RAD.

6.2.1 Form of Function

A function consists of three parts:

1. A PROC statement assigning a name to the function and naming the input parameters.
2. A heading that consists of data declarations. The function name, input parameters, and any data used in the function but not defined in the main program with which the function is compiled must be defined in the heading.
3. A body that consists of operative statements enclosed in BEGIN and END brackets. During execution, when the END bracket is reached, control returns to the statement containing the

function call. A RETURN statement can be used to cause a return to the calling program from within the body of the function rather than at its logical end. Return may not be via a GOTO statement specifying a label within the calling program.

6.2.1.1 PROC Statement. The form of a PROC statement to identify a function is:

PROC function-name ([input-parameter][,[input-parameter]]...)$

---

function-name	Programmer-assigned symbolic name used to identify the function.
input-parameter	Name of a single item, defined in the heading of the function, that will receive data from the calling program.

RESTRICTIONS: The following restrictions apply to functions.

1. The input parameters must be single items.
2. The names of the input parameters must be unique within the function but they can duplicate names defined in other regions of the program.
3. Functions must not contain other functions or procedures.
4. A function must not be enclosed in BEGIN and END brackets.
5. The output parameter (function-name) must be defined as a single item.
6. There must be at least one input parameter in a function.

6.2.2 Form of Function Call

A function call is not a complete operative statement. It can be part or all of an arithmetic expression. It consists of a function name followed by input parameters. After the function is executed, the output parameter contains the value determined by the operations in the function. The effect would be the same if instead of using a function, operative statements set an item to a value; and this item, instead of the function call, was used in an arithmetic expression.

The form of a function call is:

---

function-name (input-parameter[,[input-parameter]]...)

---

function-name	Function name given in the PROC statement.
input-parameter	Expression or table address that is to be used as an argument. The value of the expression specified as the first input parameter in the call is passed to the item specified as the first input parameter in the PROC statement, etc.

RESTRICTIONS: The following restrictions apply to function calls.

1. When a table name or unsubscripted array name is used as an input parameter, the address of the beginning of the table or array is transmitted to the corresponding input parameter given in the PROC statement, which must be an integer, EBCDIC, or ASCII item of 24 or more bits and a scale of zero.
2. The input parameters must correspond in number and order to the input parameters in the PROC statement.

3. A null field indicated by two consecutive commas (or by an opening parenthesis and a comma, or by a comma and a closing parenthesis, for the first and last parameter respectively) can be used to represent an omitted input parameter. If a null input parameter is used, the corresponding input parameter in the PROC statement remains set to the value it had the last time the function was called. If a function has only one input parameter, it may not be omitted in the call.
4. A function cannot call itself.
5. A function cannot call another defined procedure which, in turn, calls it on any level, i.e., the called defined procedure cannot call the function or call another defined procedure that calls the function, etc.
6. A function call can be used as an input parameter to another function or procedure.
7. The return to the calling program is to the operative statement that contains the function call.

6.2.3 Example of a Function

The following example shows a calling program and a function.

Program	Comments
START CALC	Identifies CALC as main program.
ITEM HRS F $	Item declaration.
ITEM SCALE F $	Item declaration.
ITEM RATE F $	Item declaration.
ITEM GROSS F $	Item declaration.
GROSS = AMT(HRS, SCALE) $	Assignment statement calling function AMT.
.	
.	
.	
.	
IF AMT(HRS, SCALE) GR 10000 $	IF statement calling function AMT.
GOTO LONG $	True exit.
GOTO SHORT $	False exit.
PROC AMT(TIME, SCALE) $	Identifies function AMT.
ITEM AMT F $	Describes function output parameter.
ITEM TIME F $	Describes input parameter.
ITEM SCALE F $	Describes input parameter.
ITEM PAY F $	Describes item used in function and not defined in region of program containing function call.

Program	Comments
BEGIN	Identifies beginning of body of function.
IF SCALE = 15 $	IF statement.
BEGIN AMT = 1000 $	Compoud statement used as true exit.
RETURN $ END	Sets function output parameter and returns to statement containing function call.
PAY = RATE * SCALE	False exit.
AMT = TIME * PAYS	Set function output parameter.
END	Return to calling statement.
LONG. ...	Statement label.
SHORT. ...	Statement label.
STOP $	
TERM $	End of program CALC.

6.3 PROCEDURES

A procedure is a type of defined procedure that can produce several values each time it is called. The procedure consists of a statement giving the procedure name and input and output parameters, followed by data declarations, followed by operative statements. The procedure call is a complete operative statement that calls the procedure and gives input and output parameters. After the procedure is executed, control returns to the statement after the procedure call.

6.3.1 Form of Procedure

A procedure consists of the following three parts:

1. A PROC statement that assigns a name to the procedure and names input parameters that will receive data from the calling program and output parameters that will return data to the calling program.
2. A heading that consists of data declarations. The input and output parameters must be defined in the heading. Any data used in the procedure that is not defined in the main program with which the procedure is compiled must also be defined in the heading.
3. A body that consists of one or more operative statements enclosed in BEGIN and END brackets. During execution, when the END bracket is encountered, control returns to the operative statement following the procedure call in the calling program. A RETURN statement can be used to return to the calling program from within the procedure rather than at the end. Return may not be via a GOTO statement specifying a statement label within the calling program.

6.3.1.1 PROC Statement. The form of the PROC statement used to identify procedure is:

```
PROC  procedure-name
[([input-parameter[,input-parameter...]]     ]
[=output-parameter[,output-parameter]...])]$
```

procedure-name	Programmer-assigned symbolic name used to identify the procedure.
input-parameter	Name of item defined in the procedure heading, which will receive data from the calling program, or a statement label (with period).
=	Separates input parameters from output parameters.
output-parameter	Name of item defined in the procedure heading, which will return data to the calling program.

RESTRICTIONS: The following restrictions apply to the procedures.

1. The output parameters must be single items. The input parameters must be single items or statement labels.
2. The names of the input and output parameters must be unique within the procedure, but they can duplicate the names of items in other regions of the program.
3. Procedures must not contain other procedures or functions.
4. A procedure must not be enclosed in BEGIN and END brackets.
5. A procedure with no parameters is written in the form: PROC procedure-name $.
6. A statement label appearing in the PROC statement must be unique within the procedure.

6.3.2 Form of Procedure Call

A procedure call is a complete operative statement. It consists of the name of the procedure, followed by input parameters that pass data to the procedure and output parameters that receive data from the procedure. After the procedure is executed, control returns to the statement after the procedure call. The items named as output parameters are set to the values determined in the procedure.

The form of a procedure call is:

```
procedure-name  [([input-parameter]
[,[input-parameter]]
..[=[output-parameter]]
[,[output-parameter]]  ...)] $
```

procedure-name	Procedure name given in the PROC statement.
input-parameter	Expression or table address to be used as argument. The value of the expression specified as the first input parameter in the call is passed to the item specified as the first input parameter in the proc statement, etc. A statement label (with period) defined in the calling region may also be used.
=	Separates input parameters from output parameters.
output-parameter	Name of variable previously defined in the calling program, which returns data from the procedure to the calling program. After the procedure is executed, the value of the variable specified as the first output parameter in the PROC statement is passed to the variable specified as the first output parameter in the call, etc.

RESTRICTIONS: The following restrictions apply to procedure calls.

1. When a table name, statement label, or unsubscripted array name is used as an input parameter, the address of the beginning of the table or array or statement address is transmitted to the

corresponding input parameter given in the PROC statement, which must be an integer, EBCDIC, or ASCII item of 24 or more magnitude bits and a scale of zero.

2. The input and output parameters must correspond in number and order to the input and output parameters in the PROC statement.

3. A null field, indicated by two consecutive commas, can be used to represent an omitted input or output parameter. If a null input parameter is used, the corresponding input parameter in the PROC statement remains set to the value it had the last time the procedure was called. If a null output parameter is used, any output value calculated by the procedure is ignored by the calling program.

4. A procedure cannot call itself.

5. A procedure cannot call any defined procedure that, in turn, calls it or calls another defined procedure that calls it, etc. That is, procedure calls cannot exhibit circularity.

6. A procedure call may not be used as an input parameter to another procedure.

7. The return to the calling program is to the operative statement following the procedure call.

8. A call to a procedure with no parameters is written in the form: procedure-name $.

9. If the procedure definition contains a statement label, and the procedure body a GOTO referencing that label, transfer will be to the address furnished by the calling program. Generally this should be a statement label in the calling region. Use of this option will, however, force base register initialization at the referenced label in the calling region. Use of a SWITCH reference such a label is not permitted.

6.3.3 Example of a Procedure

The following is an example of a procedure.

Procedure	Explanation
PROC CALC (RATE, TIME=DIST, ERROR) $	PROC statement identifying procedure CALC and parameters.
ITEM RATE F $	Data declarations for all input and output parameters.
ITEM TIME F $	
ITEM DIST F $	
ITEM ERRORS V(NO)	
V(YES) $	
BEGIN	Indicates beginning body of procedure.
IF RATE EQ 0 OR TIME EQ 0 $	Test for error condition.
BEGIN NUMERR = NUMERR + 1 $ ERROR = V(YES) $ RETURN $ END	True exit. Item NUMERR in the main program region is incremented by 1. ERROR is set to YES and return is to statement after the procedure call in the calling program.
ERROR = V(NO) $	False exit. No error condition exits.

Procedure	Explanation
DIST = RATE * TIME $	Set output parameter DIST.
END	Indicates end of procedure. Return is to statement after procedure call in the calling program.

6.4 CLOSED PROGRAMS

A closed program can be compiled by itself but is executed only if called from another JOVIAL program. The method of passing data between a closed program and the calling program is through a compool. The same compool must be specified on the START statements, of both programs.

6.4.1 Form of a Closed Program

A closed program is identified by the CLOSE option on the START statement. Following the word CLOSE is a programmer-assigned symbolic name that identifies the closed program so that it can be called by name.

Following the START statement in the closed program are data declaration and operative statements as for a main program. The only difference is in the compiler's interpretaton of STOP statements.

The form of the STOP statement is:

STOP[(expression)] $

The STOP statement, when encountered in a closed program, causes control to return to the statement following the GOTO statement that called the closed program.

RESTRICTIONS: The following restrictions apply to closed programs.

1. A closed program can contain defined procedures.
2. The method of data communication is through the compool.
3. Return from the closed program is to the statement after the call in the calling program. The method of return is through the STOP statement.
4. (expression) is a return code valid only under MVS. See the section on "Operative Statements" for further details.

6.4.2 Form of Call to Closed Program

A closed program is called by a GOTO statement in the calling program.

The form of the GOTO statement is:

GOTO closed-program-name[(= left-term)] $

---

closed-program-name	Symbolic name that appeared after the CLOSE option on the START statement at the beginning of the closed program.
left-term	Item in which a return code is to be stored.

RESTRICTIONS: The following restrictions apply to calls to closed programs.

1. The closed program is not executed unless it is called from another program at the execution time. Both of the programs must be loaded at the same time.

2. Data communication is through a compool.

3. Use of return code is valid only under MVS. See the section on "Operative Statements" for further details.

6.4.3 Declaration of a Closed Program

Declaration of a symbol as a closed-program-name in a calling program is always permitted and in some cases required. An otherwise undefined name used in a GOTO statement will be assumed to be a closed-program-name, but a warning will be issued. If an ordinary linkage is desired, either an explicit declaration or the above assumption is permissible. If linkage via NAS SVC 104 is required, an explicit declaration must be used.

The format of the declaration is:

CLOSE closed-program-name { EXTRN / LINKABL } $

---

closed-program-name	Symbolic name appearing on START statement of program being called.
EXTRN	Ordinary linkage to be used.
LINKABL	NAS SVC 104 linkage to be used.

RESTRICTIONS: The following restriction applies to closed-program declaration.

1. The declaration may appear at any point in the calling program where a CLOSE statement would be permitted.

6.5 LIBRARY ROUTINES

A library routine is a function or a procedure, separately compiled, that is stored in the JOVIAL library. It is extracted from the library at load time and loaded with the object program that calls it.

6.5.1 Form of Library Routine

A library routine begins with a START statement with the LIBE or LINKABL option specified on it and ends with a TERM statement. Between these statements is a procedure or a function; i.e., a PROC statement followed by the heading and body of a function or procedure.

RESTRICTIONS: The following restrictions apply to library routines.

1. A library routine is subject to the same restrictions as a function or a procedure except that there are no main program variables which could be referenced, and input or output parameters may not be status type items.

2. If a library routine contains a call to a closed program, the closed program must be in storage when the library routine is executed.

3. Reference can be made to compool data. Note that if the compool is changed, the library routine must be updated to reflect the change.

4. A library routine can be written in direct code, in which case a DIRECT bracket will follow the START statement. If the library routine is written in direct code, the programmer must supply the linkages to it (i.e., Prologue and Epilogue coding in the library routine). The programmer must also supply his own LIBEDT control statement.

5. The PROC statement immediately following the START statement must define a PROC whose name is identical to the program name on the START statement.

6. A library routine provides an exception to the rule that PROCs must not be declared between BEGIN and END brackets. PROCs may appear between the BEGIN and END brackets corresponding to the PROC statement immediately following the START statement.

6.5.2 Form of Call to Library Routines

If the library routine is a function, a function call is used; if it is a procedure, a procedure is used. The form of function and procedure calls is given under the headings "Functions" and "Procedures".

6.5.3 Example of a Library Routine

Below is a skeleton example of the statements needed in a library routine.

Statement	Explanation
START LIBE SALE	Identifies SALE as library routine.
PROC SALE(IN, OUT) $	PROC statement.
ITEM SALE F $	Begin heading of function and identifies SALE as a function.
ITEM IN F $	
ITEM OUT F $	
BEGIN	Begin body of function.
IF OUT GR ORDER $	
.	
.	
.	
END	End body of function.
TERM	End of library routine.

6.6 RETURN STATEMENT

The form of the RETURN statement is:

```
RETURN $
```

RESTRICTIONS: The following restrictions apply to RETURN statement.

The RETURN statement must appear between the BEGIN−END brackets of a procedure, function, or closed-compound procedure.

6.7 GENERAL COMMENT ABOUT DEFINED PROCEDURES

The following statements summarize the rules given for defined procedures and show the relationship between the different kinds of defined procedures.

1. A defined procedure can be entered only at the beginning.

2. After the defined procedure is executed, control returns to the point of departure from the calling program. Except for closed programs, the return occurs when an END bracket or a RETURN statement is encountered. In a closed program, the return occurs when a STOP statement is encountered. A GOTO statement can be used in a closed-compound procedure to transfer to any statement in the region in which it is defined.

3. Closed programs and library routines are compiled by themselves and must begin with a START statement and end with a TERM statement. They are executed only if called from another program. The rest of the defined procedures must be compiled with the program that calls them. They may be placed anywhere in the program, except that closed-compound procedures must not interrupt program flow.

4. Data communication between a calling program and a closed program is through a compool. Closed-compound procedures can refer to any data for the region in which they are defined. In the rest of the defined procedures, parameters in the calling statement and the PROC statement are used to pass data.

5. A defined procedure cannot call itself (i.e., recursive use is not permitted) but may call other defined procedures.

6. Calls to a defined procedure must not exhibit circularity. That is, for a sequence of defined procedures whose logic is such that each defined procedure calls a lower level defined procedure, the lower level defined procedure must not call higher level defined procedures.

7. Main programs, closed programs, and library routines can contain functions, procedures, and closed-compound procedures. Closed-compound procedures, functions, and procedures can contain closed-compound procedures.

8. A defined procedure can refer to a compool.

6.8 BLOCK−DATA PROGRAMS

A Block-Data Program is a special compilation whose purpose is to provide an object module containing preset data for various data declarations (usually compool defined or derived).

Data declarations are compiled normally, and the ZYDATA CSECT is placed at zero displacement relative to the start of the program. Operative statements are not required. They are permitted but are assembled in a DSECT and produce no object code. Use of DIRECT code to preset data is permitted and should be as described on page 7−4 with one exception: the last instruction in the sequence should be “ZYPROG DSECT”.

The use of the Block-Data Option is selected by specifying the keyword “BLKDATA” on the START statement.

6.9 SPECIAL RESTRICTIONS ON PROCEDURE/FUNCTION CALLS

Recent compiler changes have been made to replace certain Library Procedure/Function calls with in-line code. Where in-line code cannot be generated, the compiler will default to normal calls. However, since it cannot be easily determined which will result, there is no guarantee that a call with null arguments will actually be able to use values input to the last previous call. Some of these can be determined early enough to force all calls to actual Library linkage with resulting loss of object code efficiency. Others will result in diagnostics if null arguments are used.

Special restrictions on procedure or function calls are as follows:

a. MVC, MVI — null arguments should be avoided, but will be permitted at present.

b. CLC, CLCI, OC, OI, NC, NI, XC, XI, TR — null arguments are not permitted and will result in a diagnostic. OI, NI and XI may or may not be available depending on the Library tape used. They require Library PDT entries for procedures ZVOI, ZVNI and ZVXI in order to compile. Depending on whether in-line code or procedure calls are generated object modules for these may be needed in order to load and execute.

7.0 DIRECT CODE

Parts of JOVIAL programs can be written in Basic Assembler Language instead of the JOVIAL language. These parts are enclosed in DIRECT $ and JOVIAL brackets to identify them as direct code and to separate them from the rest of the program.

The information between DIRECT $ and JOVIAL is considered to be one JOVIAL statement. For example, such a statement may be the true branch of an IF statement or the range of a FOR. The DIRECT $ bracket may include an estimated count of the number of bytes of storage used in direct code.

The format is:

DIRECT[count] $

---

count An integer constant specifying the estimated number of bytes used in direct code.

RESTRICTIONS: The following restrictions apply to direct code:

1. The count should be a high estimate. Code generated from ASSIGN statements and literals should be included in the estimate.

2. If no count is specified, the compiler assumes the worst case, namely that the direct code is arbitrarily long. This will result in a LTORG that is often unnecessary. DIRECT with an excessively high estimated count is still better than no count at all.

Direct code may be entered only at the DIRECT $ bracket, either because it is the next sequential statement or because transfer to it was made by a GOTO statement. A statement label (providing a reference for transfers) can precede the bracket DIRECT $.

Control may not be transferred from a JOVIAL statement to the middle of a direct code. However, transfer of control within direct code is not restricted, even if the direct code is not within the same DIRECT $ and JOVIAL brackets.

All Basic Assembly Language machine instructions (including privileged instructions) and all assembly instructions except ICTL, SPEM and END can be used. The publication IBM Data Processing System: Basic Assembly Language User's Manual (BALASM) describes the Assembly Language.

The JOVIAL compiler generates code assuming the following types of program interrupts are masked out.

fixed-point overflow

decimal overflow

exponent underflow

significance

If these masks are changed in a section of direct code, it may be desirable to reset them before re-entering the JOVIAL-coded program. System masks are discussed in the IBM Data Processing System Principles of Operation.

The following sections explain the conventions used to refer to data in direct code.

7.1 ASSIGN STATEMENT — REFERENCE TO JOVIAL DATA BY NAME

The ASSIGN statement is a JOVIAL statement that may be used in direct code to refer by name to items defined in JOVIAL data declarations. There are two forms of the ASSIGN statement: the first sets a register to the current value of a JOVIAL item; the second sets a JOVIAL item to the current value of a register.

The forms of the ASSIGN statement are:

```
ASSIGN { R(scale)   = arithmetic-expression $ }
       { left-term  = R(scale)              $ }
```

R designates — Floating-point register 0 will be used if the left-term is a floating point item not modified by BIT or BYTE, or if the first term of an arithmetic-expression is a floating point item not modified by BIT or BYTE.

General register 1 will be used for EBCDIC or ASCII items of four or fewer characters, or for integer, fixed point, or status items.

General register pair 0–1 will be used for EBCDIC or ASCII items of more than four characters. BIT or BYTE modifiers will be treated as EBCDIC items.

scale — Integer specifying number of bits to the right of the binary point. Must be specified even if scale is zero.

left-term — As left-term for Assignment statement, except ENT is not permitted.

arithmetic-expression — As arithmetic-expression for Assignment statement, except ENT is not permitted.

RESTRICTIONS: The following restrictions apply to the ASSIGN statement.

1. An ASSIGN statement must appear on a card image by itself, except that it can be on the same card image as the DIRECT $ or JOVIAL bracket.
2. Data is not converted except for shifting to adjust the binary point for non-floating-point data.

EXAMPLE: The following example shows an ASSIGN statement used to refer to JOVIAL data by name.

```
ITEM ONE F $

DIRECT $

ASSIGN R (0) = ONE $

.

.

.

.

JOVIAL
```

7.2 REFERENCE TO ADDRESS OF JOVIAL DATA

Addresses of JOVIAL data defined in other parts of a JOVIAL program can be loaded into a general register using an instruction of the following format:

```
L    register,=A(xxname)
```

register	A general register. A blank must not follow the register specification.
=A(xxname)	An address constant literal that contains no embedded blanks. =A(...) is the assembly language method of indicating address constant literals. The expression enclosed in parentheses indicates the JOVIAL data whose address is to be loaded. The xx is a data prefix that indicates the program region and name is a JOVIAL-defined data name.

A data prefix of A1 indicates that the item was defined in the main program. Functions and procedures are assigned data prefixes in the order of their appearance in the program; A3, A5, A7, A9, AB, AD,..., AX, AZ, B1, B3,..., YZ. All compool data is assigned to the prefix ZX.

To reference data which may be in the same region, the main program, or the compool, use the prefix _1. In resolving this reference, the compiler will first assign the prefix for the current region; if not found, a prefix of A1 will be assigned; if not found then, a prefix of ZX will be assigned. If the required name is not found with either of the three prefixes, the data name is unresolved, and an assembly error will occur.

EXAMPLE: In the following example, showing direct code references to the addresses of JOVIAL data, assume that the main program contains a function and then a procedure. In the main program region, Item ONE is defined; in the function, Item TWO is defined; in the procedure, Item THREE is defined. Then the following statements can be given to load the addresses of Items ONE, TWO, and THREE into general registers 2, 3, and 4:

```
L  2,=A(A1ONE)

L  3,=A(A3TWO)

L  4,=A(A5THREE)
```

7.3 REFERENCE TO JOVIAL STATEMENT LABELS

The address of JOVIAL statement labels defined in other parts of a JOVIAL program can be loaded into a general register, using an instruction of the following format.

```
L    register,=A(yylabel)
```

register	A general register. A blank must not follow the register.
=A(yylabel)	An address constant literal that contains no embedded blanks. The letters yy represent a statement prefix that indicates the program region and label is a JOVIAL label.

A data prefix of A0 indicates that the label is in the main program. Functions and procedures are assigned statement prefixes in the order of their appearance in the program; A2, A4, A6, A8, AA, AC,..., AW, AY, B0, B2,..., YY. Library program names are assigned the prefix ZW or ZV.

To reference a label which is in the same region, you may prefix the label with _0. The compiler will assign the prefix of current region.

7.4 ADDITIONAL DIRECT CODE LIMITATIONS

1. The compiler generates BAL symbols as described above to avoid duplication of names in various program regions and assigns these symbols to a "$" qualification field. This qualification carries

into direct code sequences unless the programmer uses his own QUAL statement. Care must be taken to ensure that conflicts between direct code symbols and JOVIAL-generated symbols are avoided. The programmer is permitted to reference and transfer freely between regions, although the results may be catastrophic.

2. While it is beyond the scope of this manual to explore he possibilities in detail, much more extensive referencing of JOVIAL defined data is possible than is suggested above. The key to this extension is that the programmer is allowed the complete BAL assembly language including all features found in code generated by the compiler. In taking advantage of this facility it is necessary to remember that the compiler has established conventions, and direct code sequences must be compatible when related to JOVIAL-generated code.

   The best way of learning the required conventions is to examine some JOVIAL-generated code that performs a function similar to what is desired and transcribe it as a direct code sequence, with appropriate changes.

3. If a DIRECT code sequence is within a FOR loop, or if the DIRECT code contains ASSIGN statements, there is a possibility of conflict between compiler-assigned registers and user-assigned DIRECT code registers. To avoid these conflicts, the programmer should use the RESERVE pseudo-op to guarantee that the compiler will not assign specified registers. See Section 8, Control Pseudo-Operations, for the use of RESERVE and RELEASE pseudo-ops. An alternative method that may be used within FOR loops is to save all registers at the beginning of the direct code sequence and to restore them at the end. This is not effective for DIRECT code containing an ASSIGN statement.

4. Usually, register 12 will be provided as a base register if the DIRECT statement immediately follows an executable JOVIAL statement. Under that condition, a DIRECT statement omitting the byte count sets the base to an address value just prior to the programmer's BAL code; but a DIRECT statement including the count is only assured of a base adequate to cover the number of bytes specified by the count. DIRECT statements not following executable JOVIAL statements cannot be assured a base register at all.

5. At the end of the direct code sequence, be sure to include a "DROP" for every "USING" within this sequence of direct code (except register 12, which does not require this). There is no need to load or restore any register at this time, except as noted in paragraph 3 above.

6. Items coded for "M" packing are aligned only to halfword boundaries. This means that they must be referenced with halfword instructions such as LH. All other generated data names are aligned to full-word boundaries, permitting instructions such as L, A, M, but often requiring extracting and shifting after loading into a register.

7. JOVIAL provisions for presetting data are at times cumbersome, especially for character data. Such presetting can be done within a direct code sequence in the main program under the following restrictions.

   a. Precede the presetting statements with the statements:

```
ZYDATA      CSECT
label       EQU       *
```

   where "label" represents any unique label. This preserves the value found in the ZYDATA location counter.

   b. Origin to each area with a statement of the form "ORG xxname" where xx represents the appropriate data prefix and name is the name of a JOVIAL table or array.

   c. After each ORG card image use DC & DS statements to describe the desired preset constants.

d. Follow the presetting statements with the statements:

```
        ORG     label
ZYPROG  CSECT
```

where "label" indicates the same unique label used in paragraph a. This restores the ZYDATA location counter to its previous value.

WARNING

Adherence to this technique is required to ensure that JOVIAL-generated data is properly aligned.

8. In addressing arrays, it is necessary to know the location of the various elements in storage. Each element is stored right-justified in a byte, halfword, single word, or two consecutive words, depending on field format.

byte	1–8 bits, unsigned
halfword	9–15 bits, unsigned, or 2–16 bits, signed
word	16–32 bits, unsigned, or 17–32 bits, signed
two words	33–64 bits, unsigned

The order of elements in storage is best shown by the following example of a main program array.

```
ARRAY BETA 2 2 2 I 32 S $
```

JOVIAL Subscript	BAL Address
BETA ($0, 0, 0$)	A1BETA
BETA ($1, 0, 0$)	A1BETA+4
BETA ($0, 1, 0$)	A1BETA+8
BETA ($1, 1, 0$)	A1BETA+12
BETA ($0, 0, 1$)	A1BETA+16
BETA ($1, 0, 1$)	A1BETA+20
BETA ($0, 1, 1$)	A1BETA+24
BETA ($1, 1, 1$)	A1BETA+28

9. Direct code appearing in the body of a PROC (including library routines) must not change the value of register 13. This is especially important in REENT programs.

7.5 EXAMPLE OF DIRECT CODE

The following example shows direct code used as the body of a procedure.

Statements	Comments
ITEM AA F$	Item in main prog.
PROC BA$	Identifies procedure BA.

Statements	Comments
ITEM BBB F$	Item in procedure.
BEGIN	Begins body of procedure.
DIRECT 30 $	Indicates direct code.
L 1,=A(A1AAA)	Loads address of AAA into register 1.
L 2, =A(A3BBB)	Loads address of BBB into register 2.
L 3,=A(CCC)	Loads address of CCC into register 3.
B DDD	Branch to end of direct code.
CCC DS F	Defines CCC as fullword item.
DDD EQU *	Sets up DDD as false.
JOVIAL	Ends direct code.
END	Ends body of procedure.

7.6 BAL DEBUG STATEMENTS USING DIRECT CODE

BAL Debug statements [see IBM Data Processing System: Debugging System User's Manual (DEBUGG) for format] can be inserted into a JOVIAL source program by using direct code (see Figure 7−1).

```
     Name        Operations          Operand
1         8   10        14  16      20      25      30      35      40      45      50
START TST     DBG           ST      ZZ      TAKE HEX DUMP OF DATA
ITEM DD       A 32 $          4 $
ITEM CC       I 32 S          $
DIRECT $        DUMP          HEX, ID AI DD $ ACC $
AO ZZ.$
JOVIAL
XX.DD=          5.5A4         $
YY.CC=          64 $
ZZ.STOP         $
TERM XX       $
```

FIGURE 7−1. EXAMPLE OF DEBUG STATEMENTS USING DIRECT CODE

7.6.1 Using Debug Statements

The following advantages can be gained by the use of BAL Debug statements:

1. They provide a convenient way for a programmer to print the contents of a tape, as it looks at the end-of-job. The JOVIAL library routines do not include a tape dump program.

2. It is not necessary to program tests for conditions on which to dump. The conditional dump statement will do this.

3. The programmer can request a trace of a section or sections of his program.

4. The programmer can define the area he wants printed in an emergency.

The following rules apply to the use of BAL Debug statements:

1. The Register-Storage Conditional Dump statement should be used with extreme caution when referring to JOVIAL source statements. The programmer must be familiar with the JOVIAL-generated code to be able to predict which quantity will occupy a given register at a given point in the program.

2. The rules for reference to JOVIAL statement labels and data names apply.

The programmer can also request tape prints or define emergency dumps with load-time debug statement (preceded by a $OBJ card image).

7.7 DIRECT CODE COMPOOL REFERENCE

Normally no special precautions need be taken referencing compool from Direct Code. The compiler can detect these references and will generate any needed PSEG card images. The exception to this is a reference to the original (ZX-prefixed) name of a Dynamically Equated Table of an item therein. Unless the segment is forced in with a ".␣PSEG segnam" or reference to another, non-Dynamically-Equated, Table in the segment, the symbols will be treated as undefined.

While use of ".␣PSEG segnam" is more efficient and is recommended, it is also possible to use BAL PSEG card in the Direct Code to define such symbols.

8.0 CONTROL 'PSEUDO–OPERATIONS'

Certain pseudo-operations are provided to allow programmer control over the format and content of the listing and to specify which general registers are to be reserved from JOVIAL use in direct code.

Control pseudo-ops can be used any place in the JOVIAL program except within direct code. The RESERVE pseudo-op has the additional restriction that it cannot be used in a FOR loop. Except for the TITLE statement, the TABLE statement, and the NLIST statement, they can not precede the START statement.

All control pseudo-ops must have the character sequence period-blank in line columns 1 and 2. Only one control pseudo-op and no other JOVIAL statements may appear on a line.

8.1 EJECT PSEUDO–OP

The EJECT pseudo-op is used to eject to a new page in the JOVIAL source language listing and/or in the generated assembly language listing.

The format of the EJECT pseudo-op is:

```
.b EJECT  [ { JOV  }  ]
          [ { BAL  }  ]
          [ { BOTH }  ]
```

JOV	Eject a page in the JOVIAL source listing only.
BAL	Eject a page in the generated BAL listing only.
BOTH	Eject a page in the JOVIAL source listing and in the generated BAL listing. If the field is omitted, BOTH is assumed.

8.2 SPACE PSEUDO–OP

The SPACE pseudo-op is used to space one or more lines in the JOVIAL source language listing and/or in the generated assembly language listing.

The format of the SPACE pseudo-op is:

```
.b SPACE  [number]  [ { JOV  }  ]
                    [ { BAL  }  ]
                    [ { BOTH }  ]
```

number	One or two decimal digits specifying the number of spaces desired in the listing. If omitted, one space is assumed.
JOV	Space lines in the JOVIAL source listing only.
BAL	Space lines in the generated BAL listing only.
BOTH	Space lines in the JOVIAL source listing and in the generated BAL listing. If the field is omitted, BOTH is assumed.

8.3 NLIST PSEUDO–OP

The NLIST pseudo-op is used to suspend the listing of the JOVIAL source program and/or of the generated BAL program. This statement may precede the START statement in the MVS version of JOVIAL.

The format of the NLIST pseudo-op is:

JOV	Suppress the JOVIAL source listing only.
BAL	Suppress the generated BAL listing only.
BOTH	Suppress the JOVIAL source list and the generated BAL listing. If the field is omitted, BOTH is assumed.
JCOM	Suppress the listing of JOVIAL source as comments in the BAL listing.

8.4 LIST PSEUDO–OP

The LIST pseudo-op is used to resume listing, which was suspended because of a previous NLIST pseudo-op. Listing can be resumed in the JOVIAL source listing and/or in the generated BAL listing.

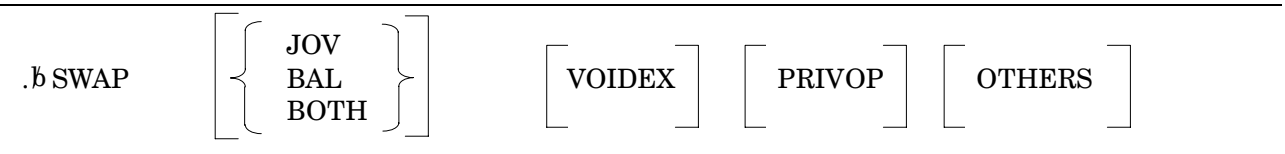
The format of the LIST pseudo-op is:

JOV	Resume listing the JOVIAL source program only.
BAL	Resume listing the generated BAL program only.
BOTH	Resume listing the JOVIAL source program and the generated BAL program. If the field is omitted, BOTH is assumed.
JCOM	Resume listing the JOVIAL source as comments in the BAL listing.

8.5 SWAP PSEUDO–OP

The SWAP pseudo-op is used to suppress printing of JOVIAL warning diagnostic messages and/or BAL possible error messages. Some selectivity is provided for suppression of BAL error messages. JOVIAL warning diagnostics are suppressed only in the assembly language listing. If a JOVIAL diagnostic listing is generated because of a serious error, warning messages will be printed regardless of the SWAP options specified.

The format of the SWAP pseudo-op is:



JOV	Suppress only JOVIAL warning diagnostics in the generated BAL listing.
BAL	Suppress only BAL possible error diagnostics in the generated BAL listing. The selectivity options VOIDEX, PRIVOP, and OTHERS are applicable.
BOTH	Suppress JOVIAL warnings and BAL possible errors in the generated BAL listing. The selectivity options VOIDEX, PRIVOP, and OTHERS are applicable. If this field is omitted, BOTH is assumed.
VOIDEX	Suppress printing of the possible error message: FIELD n HAS A VOID EXPRESSION. This option applies only if BAL or BOTH was specified or assumed.
PRIVOP	Suppress printing of the possible error message: USE OF A PRIVILEGED OPERATION. This option applies only if BAL or BOTH was specified or assumed.
OTHERS	Suppress printing of all other BAL possible error messages. This option applies only if BAL or BOTH was specified or assumed. Any combination of VOIDEX, PRIVOP, and OTHERS may be specified in any order. If all three are omitted, VOIDEX and PRIVOP are assumed.

8.6 RESERVE PSEUDO–OP

The RESERVE pseudo-op guarantees that the programmer may use registers in DIRECT code without having conflicts with JOVIAL-assigned registers. The RESERVE pseudo-op should be used when ASSIGN statements appear in a DIRECT sequence, and when a section of DIRECT code appears within a FOR loop.

The format of the RESERVE pseudo-op is:

.b RESERVE register-number [register-number] ...

---

register-number	A 1-digit decimal number specifying a register to be reserved from JOVIAL use in direct code.

RESTRICTIONS: The following restrictions apply to the RESERVE pseudo-op:

1. Only registers 2 through 9 may be RESERVED.
2. The RESERVE pseudo-op cannot be used within a FOR loop or within direct code.
3. Though it is legal to RESERVE as many as eight registers, it is advisable to RESERVE only the minimum number of required registers. The more registers that are RESERVED, the better the chance that inefficient code will be generated by JOVIAL.
4. The RESERVE pseudo-op has no effect on JOVIAL's assignment of registers as data bases because these are dropped before entering a DIRECT sequence.

8.7 RELEASE PSEUDO–OP

The RELEASE pseudo-op is used to free a register that was previously RESERVED from JOVIAL use.

The format of the RELEASE pseudo-op is:

.b RELEASE register-number [register-number] ...

---

register-number — A 1-digit decimal number specifying a register to be reserved for JOVIAL use. All RESERVED registers need not be RELEASED at once.

8.8 PSEG PSEUDO–OP

The PSEG pseudo-op provides a means of forcing the compiler to recognize an otherwise unreferenced compool segment. Under MVS, all data-names referenced on a PSEG will appear in the INDEX, and XREF output if present.

The format of the PSEG pseudo-op is:

.ƀ PSEG data-name [data-name] ...

---

data-name — The name of a TABLE, ITEM, STRING, ARRAY, or compool segment.

RESTRICTIONS: The following restrictions apply to the PSEG pseudo-op.

1. If a name cannot be found in the compool, a warning is issued and the compiler proceeds to the next name (if present).
2. If there is no compool, or an undecipherable field is found, a warning is issued and the rest of the card image is ignored.
3. If the name of a dynamically-equated table or one of its items appears, it will be treated as undefined. See the section on the EQUATE statement for more detail.
4. More than one reference to a single segment is permitted, so this may be used to force names into XREF output under MVS.

8.9 DUMP PSEUDO–OP

The DUMP pseudo-op allows the user to include requests for debugging services without resorting to DIRECT code.

The format of the DUMP pseudo-op is:

.ƀ DUMP { DUMP / DUMPC / DUMPR / DUMPE } Operand

Operand — A debug pseudo-op operand acceptable to the BAL Assembler.

RESTRICTIONS: The following restrictions apply to the DUMP pseudo-op:

1. The first field (“DUMP”) must terminate in or before column 8.
2. The second field (“DUMP”, “DUMPC”, “DUMPR”, or “DUMPE”) must not start before column 10.
3. The third field (operand) must be separated from the second field by at least one blank, and must terminate in or before column 66.
4. The compiler provides a label for the debug request. This label is in the operative code section of the program, and appears after the last preceding operative statement, and at or before the next succeeding operative statement.

5. Use of any BAL op-codes other than those listed will lead to either JOVIAL or BAL serious errors.

8.10 RELOAD PSEUDO−OP

The RELOAD pseudo-op allows the user to force reloading of any registers currently containing base values for Dynamic Equates. It is intended for use where the program has modified base-item values out-of-line where the compiler will not detect it and provide automatic reloading.

The format of the RELOAD pseudo-op is:

.␣RELOAD

8.11 TITLE PSEUDO−OP

The TITLE pseudo-op allows the user to specify up to 48 characters of title information to appear in his JOVIAL and BAL listing header lines. Unlike the other pseudo-ops, the TITLE pseudo-op may precede the START statement. For this reason, it must be entered in fixed columns.

The format of the TITLE pseudo-op is:

```
                    1                   6
Col       1         6                   3
          .␣TITLE  variable text for header .....
```

8.12 HOOK PSEUDO−OP

The HOOK pseudo-op allows the user to create an entry point to the program.

The format of the HOOK pseudo-op is:

.␣HOOK name

---

name — A 1−6 character alphanumeric symbol, the first character of which is a letter.

The code generated by the HOOK pseudo-op is as follows:

```
ZH name   DS      OH
          ENTRY   ZH name
```

8.13 INCLUDE PSEUDO−OP

The INCLUDE pseudo-op allows the user to include source statements from a source library into the program.

The format of the INCLUDE pseudo-op is:

.␣INCLUDE member-name

---

member-name — The name of the member of the source library (defined on the //SYSLIB DD statement) to be included.

RESTRICTIONS: The following restrictions apply to the INCLUDE pseudo-op.

1. It is available on MVS JOVIAL only.
2. Included members may contain INCLUDE statements, but the nesting level may not exceed 15.
3. An included member cannot include itself, however remotely.
4. Included DIRECT code must be terminated by a "JOVIAL" bracket in the included member.

NOTE

See Section 10 for examples and explanation of JCL requirements when using the INCLUDE pseudo-op.

8.14 TABLE PSEUDO–OP

The TABLE pseudo-op is used to reallocate memory work space in Phase 1 of the JOVIAL compiler. If used it should allow the user to reduce the MVS region size needed for a given program.

The TABLE pseudo-op must precede the start statement.

The table names and values to be entered, for a given program, will be listed by JOVIAL after each successful completion.

The format of the TABLE pseudo-op is:

. ƀTABLE name value [name value] ...

---

name — Name of a JOVIAL work table to be changed.

value — A two or three digit number.

9.0 HELPFUL HINTS FOR JOVIAL USERS

The purpose of this section is to aid the programmer in avoiding potential debugging problems and to write more efficient JOVIAL code. Readers may submit contributions to this section for publication in future editions of the JOVIAL User's Manual.

Following is a list of suggestions for programming in the JOVIAL Language.

1. Do multiply by powers of 2 where appropriate, including negative powers of 2 (exp *.25A2 will give better code than exp/4, but do not try exp *.25 as this is floating point and not interpreted as a power of 2).

   Do be careful when you wish fractions truncated.

   ```
   ITEM II I 32 S $
   ITEM AA A 32 S 2 $
        II = 7 $
        AA = II/4 $
   ```

   "Since the dividend was a full word integer the quotient has no fractional bits and AA equals 1.0."

   ```
        AA = II *.25A2 $
   ```

   "Here the generated code is L 1, A1II ST 1, A1AA which is economical and gives the more precise answer of AA equals 1.75."

2. Do use MVI procedure calls and explicitly specify every argument in every call. Expect nearly perfect in line code when length is constant, a reasonably efficient subroutine call when it is not.

3. Recognize that, almost universally, you will be penalized code-wise if you make byte packing less than 8 bits, medium packing other than I 16 S or A 16 S, or normal packing other than 32 or 64 bits. Single-bit dense-packed status items are very efficient, especially if not subscripted; otherwise avoid dense-packing.

   Dense-packed items which will be used in arithmetic computations should be right-justified in the word whenever possible. This is not necessary where the primary use is a compare against a constant; i.e., for use in "IF STAT EQ V(MAYBE) $" it makes no difference whether we define STAT as:

   ```
   ITEM STAT S 2 V(YES) V(NO) V(MAYBE) n 11 D $
   ```

   or

   ```
   ITEM STAT S 2 V(YES) V(NO V(MAYBE) n 30 D $
   ```

4. Note that JOVIAL will generate an LPR instruction only in response to an absolute expression and will truncate leading magnitude bits only in response to violation of paragraph 3.

5. However, remember that small fields are occasionally appropriate since, as a general rule, if less integer bits are declared in source fields, results will be permitted more fractional bits without

truncation. Also, when comparing items in different field sizes, CLC can be used if, for example, one field is H3, dense packed, on byte boundaries, and the other is I 20 U, unpacked.

6. Note that "ADR(name)+constant" or "LOC(name)+constant" is processed as a single term when the rules of precedence permit and is therefore preferred whenever usable.

7. Note that at any time small positive constants are preferred over small negative constants.

   Bad example:

```
TMP = TMP+12 $
     LA        1,12
     A         1,A1TMP
     ST        1,A1TMP
MVI(TMP-4,8,LONGH) $
     L         1,A1TMP
     SH        1,=X'0004'
     MVC       0(8,1),A1LONGH
```

   Good example:

```
MVI(TMP+8,8,LONGH) $
     L         1,A1TMP
     MVC       8(8,1),A1LONGH
TMP = TMP+12 $
     LA        1,12
     A         1,A1TMP
     ST        1,A1TMP
MVI(LOC(TABL)+17,8,LONGH) $
     MVC       A1TABL+17(8),A1LONGH
```

8. Review the following special procedure summary:

   MVC — always moves 1–256 bytes, loss of coding efficiency if any arguments are omitted in any call.

   MVI — always moves 1–8 bytes, loss of coding efficiency if any arguments are omitted in any call.

   CLC — always compares 1–256 bytes, serious diagnostic if arguments are omitted.

   CLCI — always compares 1–8 bytes, serious diagnostic if arguments are omitted with constant length field, warning diagnostic and possible unexpected results if arguments are omitted with other than constant length field.

   NC/OC/XC/TR — always execute the appropriate 370 instruction with length of 1–256 bytes, serious diagnostic if arguments are omitted.

   NI/OI/XI — execute NC/NI/OC/OI/XC/XI instructions with length of 1–8 bytes, serious diagnostic if arguments are omitted with constant length field. These calls should be used with variable length field only if procedure is declared in program or on library and missing arguments cause diagnostic and invalid results.

9. Note that for all these calls, if an address expression includes a small (1–999) positive constant, it is almost always cheapest to write the constant as the last term of the argument expression in the procedure call.

10. Note also, for variable length MVC/CLC/NC/OC/XC/TR, if a small positive constant is involved it should be the last term of the length expression. This is especially advantageous if the constant is one.

11. Review all references to Table ME in your programs. Previously, it was good coding practice to use MEMST or MESTH rather than MVI, MVT, or CLCI. Now, the best coding practice is to use MVI, and CLCI, while MVT still does not provide good code.

    Note most references to MESTR should be changed. If MVI and CLCI are inappropriate MESTH will almost accomplish the desired result with much less code than MESTR.

12. Avoid statement labels and GOTOs. Use of IF and IFEITH/ORIF with BEGIN−END brackets is almost always more efficient.

13. Avoid excessive use of CLOSEs and PROCs, especially those called only once. In-line code can save considerable overhead.

14. Use extreme caution when combining CLOSEs and FOR-loops. The following example will compile correctly but will cause terrible program bugs when executed.

```
FOR A =...$
    BEGIN "A"
    .
    .
    .
    GOTO KLOZ $
    .
    .
    .
    END "A"
    .
    .
    .
CLOSE KLOZ $
    BEGIN
    .
    .
    .
    FOR A =...$
    .
    .
    .
    END
```

    After GOTO KLOZ $, the value of A has been destroyed. A PROC should be used instead.

15. When a data address is contained in an item, it may be more economical to declare a dummy table and use the item as a Dynamic Equate base than to use the item explicitly.

    EXAMPLE:

```
MVC (ITUM + 3,...) $
MVC (ITUM + 10,...) $
 .
 .
 .
MVC (ITUM + 127,...) $
```

    could be recoded as:

```
TABLE DUM R 1 1 $ BEGIN END
```

```
EQUATE ITUM/DUM $
MVC (LOC(DUM) + 3,...) $
MVC (LOC(DUM) + 10,...) $
 .
 .
 .
MVC (LOC(DUM) + 127,...) $
```

with a saving of a Load and an Add on each MVC in return for a single load at the beginning.

16. Do not intermix CLOSE and PROC declarations, as the literal pool with be dumped extra times.

Bad:	Main Program	Good:	Main Program
	PROC		CLOSE
	CLOSE		CLOSE
	PROC		PROC
	CLOSE		PROC
	PROC		PROC
	6 literal pools		only 4 literal pools

17. When extensive subscripting is needed, define a FOR−Loop and subscript with the FOR−Index, instead of using an item subscript. Often even more savings can be achieved using a Dynamic Equate set to the desired entry, and eliminating the subscript entirely.

18. Do not mask data being accessed when same or more restrictive masking will be applied in store operation.

    BIT ($X, 3$)(YY)=BIT ($29, 3$)(ZZ) $ is less efficient than:
    BIT ($X, 3$)(YY)=ZZ$

19. Make good use of DIRECT code for presetting tables (in non-reentrant programs) instead of using operative statements.

20. Watch out for PROCs which may reset a Dynamic Equate base item. Unless the item name appears as an output parameter of the PROC, follow the call with a ".RELOAD" to insure base register integrity.

21. When a FOR-loop index is used only for loop control and NOT as 'an index, coding:

    FOR A = 1, 1, ITUM $

    is more efficient than coding:
    FOR A = 0, 1, ITUM−1 $

    where ITUM is a simple item or an unsubscripted N-packed table item.

10.0 JOVIAL PROCEDURES

10.1 JOVIAL PROCEDURES

JOVIAL programs which are intended to execute under MVS should be compiled a closed programs. JOVIAL accesses a disk-resident compool and library. The library PDT dataset is required for all JOVIAL compilations. The compool table dataset is required only if the program being compiled requires a compool.

10.1.1 JOVIAL Compilation

An ISPF panel is provided to execute JOVIAL compilations under MVS. The panel is accessed from Option F.B from the primary options menu. The panel appears as follows:

```
                                JOVIAL COMPILE

COMMAND ===>

ENTER JOVIAL PARAMETERS    ===>   INDEX,STRUC, XREF
ENTER BAL PARAMETERS       ===>   LIST,ANALYZ,LIBGO,LOAD

SPECIFY SOURCE LIBRARY:
  PROJECT    ===>   HE0110
  GROUP      ===>   RNASOP ===>       ===>        ===>
  TYPE       ===>   SOURCE           (LIB OR SOURCE)
  MEMBER     ===>   DTOA
READ PASSWORD ===>                   (ONLY IF REQUIRED)

LIST FILEID       ===>                        (LIST OF MEMBERS TO COMPILE)

ASSEMBLER TYPE ===>   HOSTBAL     (9020BAL OR HOSTBAL-CONTAINS 370 INSTR)
SAVE OBJECT    ===>   YES         (YES OR NO-USE WITH CARE)
INCLUDE LIBRARIES REQUIRED?    ===>   YES      (YES OR NO)
PRINT OUTPUT   ===>   YES         YES OR NO)

COMPOOL REQUIRED? ===>                LIB ROUTINES REQUIRED?
SPECIFY COMPOOL (OPTIONAL):           ===>   YES      ( YES OR NO )
  PROJECT    ===>   HE0110
  GROUP      ===>   RNASOP
```

The following JOVIAL parameters can be specified:

- INDEX — will cause a cross reference listing to be provided.
- NOINDEX — will suppress the cross reference listing. If neither INDEX nor NOINDEX is coded, INDEX is assumed.
- SYNCK — will cause Phase 1 diagnostics to be listed in line with the JOVIAL listing. WARNING: If this option is used, JOVIAL quits after Phase 1.
- STRUC — will cause the JOVIAL listing to be formatted according to the rules of structured programming.
- XREF — will cause JOVIAL XREF card images to be produced.

Default JOVIAL parameters are INDEX, STRUC, and XREF.

The following BAL parameters can be specified.

- LIST — causes a program listing to be produced.
- PUNCH — causes an object module to be produced.
- ANALYZ — causes a cross-reference listing to be produced.
- XREF — causes XRF data to be produced.
- PUNCHC — causes referenced compool segments to be assembled, one assembly per segment.
- LISTD — causes the compool DSECTs to be listed.
- LISTP — causes compool segments assemblies to be listed.
- PUNCHS — this option is ignored.
- INDEX — this option is ignored.
- LOAD — causes an object module to be produced.
- LIBGO — causes LIB card images to be included in the object module.

Default BAL parameters are LIST, ANALYZ, LIBGO, and LOAD.

The source library specifies the JOVIAL source. The library type must be either LIB or SOURCE. Additional libraries can be specified in GRP2, GRP3, or GRP4. These libraries must have the same project and type as specified for the first library. Multiple compiles may be performed by leaving the member name blank and supplying a CMS file identifier in the form FILENAME, FILETYPE, FILEMODE which contains a list of members to be compiled. These members should exist in the library specified as the source library. If the ISPF library containing the JOVIAL source is read protected, the read password for the CMS minidisk containing the library must be entered.

The assembler type must be specified as 9020BAL or HOSTBAL (HOSTBAL includes 370 instructions). Specify YES to object from the assembly. Specify YES to have the output printed.

If a compool is required for the JOVIAL compilation, enter YES, otherwise specify NO. If the compool needed for compilation is other than the default (XXXXXXXX.YYYYYYYY.CTAB where XXXXXXXX is the source project and YYYYYYYY is the source group), specify the appropriate project and group.

If include libraries or library routines are required, enter YES in the appropriate fields, otherwise enter NO. If YES is specified for either, the following panel will be displayed:

JOVIAL INCLUDE LIBRARY

COMMAND ===>

ADDITIONAL INCLUDE LIBRARIES AND PDT DATA SETS

SPECIFY INCLUDE LIBRARIES (FULLY QUALIFIED OS DATA SET NAME) :

DSNAME1 ===> INCLIB1

DSNAME2 ===> INCLIB2

DSNAME3 ===>

SPECIFY ADDITIONAL PDT DATA SETS (FULLY QUALIFIED OS DATA SET NAME) :

DSNAME1 ===> ASUP.PDT.LIB001

DSNAME2 ===>

You may specify up to three MVS cataloged datasets for the include libraries which will be concatenated in the SYSLIB DD statement. You may also specify two MVS cataloged datasets for library routines which will be concatenated in the SYSLIBPDT DD statement. The JOVINC and LPDT datasets are already included and do not need to be entered here.

10.1.2 Deleted

10.2 DELETED

10.2.1 Deleted

10.2.2 MVS Region Size

The region size needed for JOVIAL, under MVS, may now be reduced by the user with information supplied by the compiler. At the end of each successful completion, JOVIAL will tell the user if the requested region was too small, reasonable, or too large. The compiler will also list the change in region size possible for this program with this compool.

JOVIAL will also produce a listing of new pseudo operation statements that should be entered before the START statement the next time this program is run. The .TABLE statements are free format but must contain a period in column one. The statement should be entered as listed. They are used to reassign internal work space used by the compiler and should allow a further reduction in the required region size.

10.2.3 Compool Data Sets

The compool provided for use under MVS consists of three disk-resident data sets. The TAB data set includes the compool directory and the data declaration tables. The reserve (RSV) data set contains the compool reserves of the form —

```
                EXTRN   ZXNAME
ZXNAME          DSECT....
```

The object data set contains an object module for each compool segment. It is used by the linkage editor to resolve external references to the compool segments. Table 10−1 describes the compool data sets.

TABLE 10−1. COMPOOL DATA SETS

Data Set Name	Data Set Description
XXXXXXXX.YYYYYYYY.CTAB	Compool Tables Data Set. Required by the JOVIAL compiler.
XXXXXXXX.YYYYYYYY.CRSV	Compool Reserves Data Set. Required by the BAL assembler.
XXXXXXXX.YYYYYYYY.COBJ	Data Set of Compool Segment Object Decks Required by the MVS linkage editor.
YYYYYYYY = Group name	
XXXXXXXX = Project name	

10.2.3.1 TAB Data Set

a. Name — XXXXXXXX.YYYYYYYY.CTAB

b. Organization — partitioned

c. Record Format — undefined

d. Blocksize — 3456 bytes

10.2.3.2 RSV Data Set

a. Name — XXXXXXXX.YYYYYYYY.CRSV

b. Organization — partitioned

c. Record Format — undefined (may be treated as if it were FB, LRECL = 80).

d. Blocksize — 3520

10.2.3.3 Object Data Set

a. Name — XXXXXXXX.YYYYYYYY.COBJ

b. Organization — partitioned

c. Record Format — fixed blocked

d. Blocksize — 2240 bytes

e. Logical Record — 80 bytes

10.3 DELETED

10.3.1 Deleted

10.3.2 Generating a Compool

An ISPF panel is provided which executes JOVIAL in the compool-generation mode. The panel is accessed from Option G.D. from the primary options panel.

```
                                COMPOOL EDIT

COMMAND    ===>

ENTER THE LEVEL AT WHICH THE COMPOOL EDIT IS TAKING PLACE;
LEVEL      ===>  R                        D   DEVELOPMENT
                                          R   RELEASE

ENTER THE COMPOOL ID ===> COMP0110

SPECIFY COMPOOL SOURCE LIBRARY:

   PROJECT      ===>  HE0110
   GROUP        ===>  NASOP       ( NAME ONLY — DO NOT PREFIX WITH LEVEL )

ENTER INFORMATION FOR THE COMPOOL EDIT DATASET:
   UNIT         ===>  3380        ( 3380 OR SYSDA )
   VOL SER      ===>  MD0001
```

This panel invokes the JOVIAL compiler to compile the compool segments. A new compool id will be created from the compool id specified on the panel. Specify the library level (development or release) at which the compool edit is taking place. The compool id should be a valid name (up to eight characters

in length) which can be used to reference the compool which is created. Specify the unit and volume serial of the disk which will contain the compool dataset. The compool project name is the project name of the compool items. The compool group name is the group name of the compool items.

10.3.3 Deleted

10.4 DELETED

10.4.1 Deleted

10.4.2 Deleted

10.4.3 Deleted

11.0 JOVIAL COMPILER

11.1 COMPILER INPUT

Figure 11−1 shows the flow of input of the compiler and the types of output produced by the compiler.

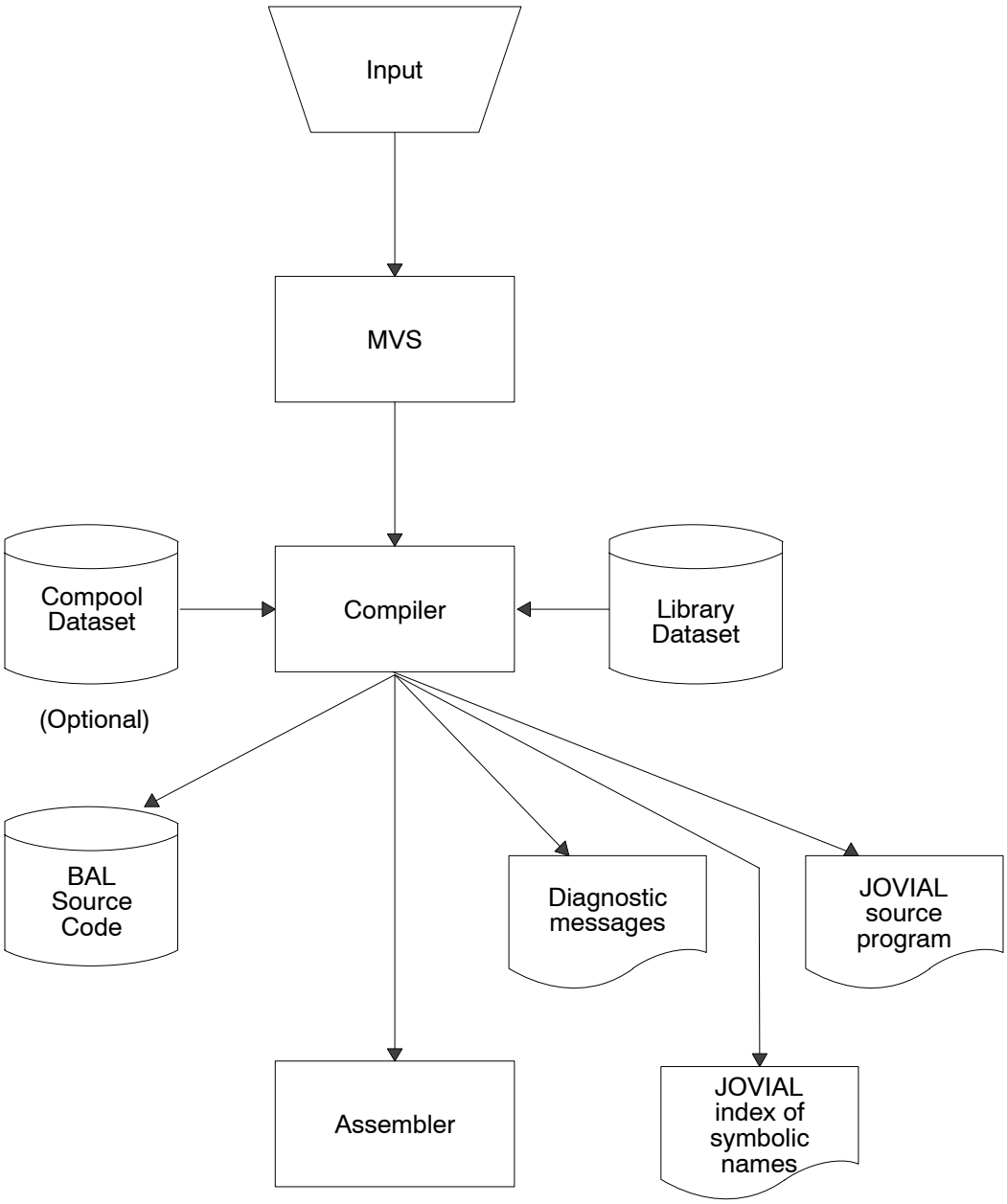


FIGURE 11–1. COMPILER INPUT/OUTPUT FLOW

Programs written in the JOVIAL language are the primary input to the compiler. If a compool is requested, the compiler reads in the compool from the compool dataset. In addition, the compiler reads in tabular information from the library dataset, whether or not a library routine is requested by the source program.

The library is described in detail in the IBM Processing System: Library User's Manual (LIBRARY). Compool and the compool edit program are described in the IBM Data Processing System: Compool Edit User's Manual (CMPEDT).

11.2 FUNCTION OF THE COMPILER

The compiler translates JOVIAL source programs into BAL for subsequent processing by the assembler and, when necessary, issues diagnostic messages. The compiler also produces a listing of the JOVIAL source programs.

The translation is performed in four phases (I, IIA, IID, and III), under supervision of the compiler coordinator. In Phase I, the JOVIAL source program statements are validated, listed on the system output unit, and processed into a canonical form (an irreducible, ordered representation), which is placed on a work file as input to Phase IIA. Further, the BAL coding required for storage-allocation requirements of the source program is placed on the primary work file. In Phase IIA, BAL coding is generated for expressions, linkage, and logic from Phase I, is placed on a second work file as input to Phase IID. The necessary BAL code for storage allocation of temporary results is added to the primary work file. In Phase IID, BAL code is generated from loop control and subscripting. The files for this phase are processed ina fashion similiar to that of Phase IIA. In Phase III, the BAL coding generated in the preceding phases is altered (as needed) to obtain optimization, registers are assigned, diagnostic messages are produced, and the coding is edited to produce a complete BAL source program which is added to the primary work file for submission to the assembler.

11.2.1 Compiler Coordinator

The major function of the coordinator is to tie together the compiler phases and to direct compilation. The coordinator communicates directly with MVS and calls in the phases, as needed from disk. It also controls all I/O operations (including reads, writes, and repositions) that the phases request.

At the start of compilation (start of INDEX under MVS), after allocating storage for all relevant compiler tables, the coordinator reads the procedure descriptor table from the library dataset into the library portion of the procedure table. To the compiler, this table acts as a directory of the routines available on the library dataset and provides parameter requirements for each routine.

If errors are detected during this initialization, e.g., too little storage, an error message is issued and the compilation is terminated. These messages, which are printed on SYSOUT/SYSPRINT are self-explanatory, and are therefore not listed in detail in this manual.

If a fatal or major error (one that halts compilation) is found during compilation, control is passed to the diagnostic processor, which places all source statments containing errors (and their corresponding diagnostic messages) on the system output unit. The diagnostic processor then returns control to the monitor, which goes on to the next job. If compilation is completed successfully, or if the programmer requests an assembly regardless of serious errors, the coordinator passes control to the monitor, which then gives control to the assembler.

11.2.2 Phase I

Phase I performs four major functions. It scans the JOVIAL source program, identifies the syntactic elements of each statement, and generats diagnostic codes for any syntactically incorrect statements; it builds the Process File, which contains input for each of the succeeding phases; it constructs tables of information that will be needed in subsequent phases; and it allocates storage for all program items, tables, arrays, and indexes.

11.2.2.1 Scan. Phase I scans the source program on three levels. It scans each statement character-by-character to validate the characters and to identify its elements. Next, it makes an element-by-element

syntactic scan of the entire statement. It also scans the entire source program to identify bracketing, nesting, transfer points, etc.

11.2.2.2 Tables. Phase I constructs seven basic tables: data, dimension, label, procedure, switch, block, and base.

The data table contains the names and descriptions of all items, strings, tables, arrays, and parameter items defined in the source program. The table also includes compool data names and descriptions when a compool is requested. The data table is built and used in Phase I.

The dimension table contains the products of the dimensions of any arrays defined in the data table. It also holds information concerning strings (i.e., skip factor, number of beads per word, and number of bytes per entry). It remains in storage through Phase IID.

The label table includes all statement names referred to or defined in the source program. This table remains in storage for Phase IIA, which uses it to process GOTO canonical forms.

The procedure table is built from the scanning of procedure/function headings and contains all information needed to generate linkage coding. All functions and procedures found in the source program, as well as the procedure descriptor table obtained from the library PDT, are recorded here.

The switch table includes the name and description of the item to be tested by each item switch. This table is used when item switch calls are processed in Phase IIA.

The block and base tables are built for Phase III. The block table contains all data names and their relative locations in storage. Each name is assigned a number that indicates the block within which the name is located. (A block is a collection of contiguous data covered by one regiser.) The base table holds the address constant that is to be loaded into a register when referencing data in each block.

11.2.2.3 Process File. This file contains the principal information processed by the compiler, and is passed from phase to phase using the work files. The file consists of seven types of process items:

1. The statement process item contains an original source statement and a compiler-generated statement number.

2. The canonical form process item presents, in coded form, information given in a statement process item.

3. The diagnostic process item may furnish one or more diagnostic codes that relate to a statement process item.

4. The BAL processes item holds a line of precursor code or a line of actual BAL code.

5. The blocking process item indicates the start of a block of coding. This is the point in the executable program to which transfer of control might be made or at which consecutive instruction sequence resumes after an interrruption. It is necessary to reload certain registers at this point. Such points include referenced statement labels, the beginning of closed subroutines, after DIRECT code, etc.

6. The table information entry indicates the type of each variable and its frequency of use. This is used to facilitate the allocations of base registers.

7. The Data Definition item contains information on data and procedure declarations.

All process items start with a standard field of four bytes. The first byte identifies the type of process item. The second contains a process code that directs the production of the final output. The third and fourth bytes are a binary count of the number of bytes in the process item.

11.2.2.4 Storage Assignment. The final processing by Phase I is to produce the information needed by the assembler for storage assignment and to place this information on the primary work file. The primary

work file contains the output of the compiler (BAL source program) and is used as input to the assembler. Here, Phase I performs the following functions:

1. Issues the START statement for the assembler.
2. Issues all CSECT pseudo-operations, labeled: ZYPROG (to define main-line coding), ZYPROC (to define procedure and function coding), and ZYDATA (to define data areas).
3. Copies, from the compool dataset, the BAL coding needed to reserve the common storage area.
4. Issues BAL pseudo-operations to allocate storage for all declared tables, arrays, items (except for dummy parameters), and FOR indexes.
5. Issues BAL code for initial value constants.

11.2.3 Phase IIA

Phase IIA generates code for expressions, calls, and logic statements, transferring to the appropriate subprocessor for actual code generation. Linkage coding is inserted between input expression evaluation and output parameter manipulations. For logic, conditional tests are inserted between the expression evaluations in the conditional statement.

The expression processor produces the coding required to evaluate expressions in any JOVIAL statement. This is accomplished in three scans. The first scan eliminates redundancy in common subexpressions and determines the order in which the expressions should be evaluated; the second finds when conversion and data unpacking is needed; and the third produces the BAL code.

The linkage processor, which is part of the first scan of the expression processor, produces calling sequences in BAL code for procedure, function, closed program, and closed-compound procedure calls. It also provides linkage for switch calls and simple branching coding.

The logic processor, also part of the first scan of the expression processor, generates coding for switch declarations and analyzes and generates coding for conditional statements.

11.2.4 Phase IID

The basic function of Phase IID is to produce code to control FOR loops in the object program. Phase IID also generates code to address subscripted data names. In each case, this phase attempts to produce optimum coding. Phase IID processes the input file segment by segment, rather than statement by statement (as in Phase IIA). A segment is as much of the Process File as the compiler can handle for analysis.

11.2.5 Phase III

This phase edits the Process File to produce BAL output that can be processed by the assembler. The editing includes the algorithmic assignment of certain registers, the altering of object code when required to optimize register assignments, and the issuing of diagnostic messages.

NOTE

General registers 0 and 1 are used as accumulators; 2 through 10 are used for three functions-data addresses, indexing, and temporary storage of partial results; 11 is used as a spill register (when all other registers are busy); 12 is used as a base register for the block being coded; 13, 14, 15 are used for linkage (14 is also used as an accumulator). Floating-point registers 0 and 6 are used as accumulators; 2 and 4 are used for temporary storage of partial results.

Finalizing the BAL source code in each block of the coding may cause a LTORG to be issued at the end of the block. The LTORG establishes the starting point for all literals defined within the block. This also permits literals to be addressed by the register covering the block's execution.

The editing portion of Phase III transforms process item formats into BAL card images, removing any embedded blanks within the operand field of the instruction, suppressing high-order zeros on addends, etc.

11.3 SUCCESSFUL COMPILATION OUTPUT

The compiler compiles the JOVIAL source program and produces diagnostic messages reflecting conditions occurring during compilation. Compiler output, including diagnostic messages, is placed on the primary work file when compilation is completed. Further, the JOVIAL source program is listed on the system output unit. The programmer may request several optional forms of compiler and assembler output by specifying the desired options.

As an aid to the programmer, the original JOVIAL statements appear as comments in the BAL listing prepared by the assembler, when requested, with each statement followed by its BAL coding unless the .␣NLIST JCOM is used.

If the .␣NLIST JCOM pseudo-op is used, JOVIAL statements following the .␣NLIST JCOM will not appear as comments in the BAL listing. However, the statement number of the JOVIAL statement which would precede the BAL code is placed in the sequence number field of the BAL code.

11.3.1 XREF Card Image Output

When the XREF option is selected, the compiler produces card images for input to XREF and other similar programs. These card images all start with .XRFn (n a numeric character) in columns 1–5.

The formats of the .XRF3, .XRF4, and .XRF7 card images, output under MVS, are described in the XREF User's Manual. The MVS .XRF3 card image has, additionally, the compool name starting in column 24 and the library name starting in column 36.

Two additional card images, XRF8 and XRF9, will appear in MVS output.

The XRF8 card image will indicate the space available in the calling program for arguments, register save, etc. for each called library routine. The format of this card image is:

Start Column	Number of Columns	Contents
1	5	.XRF8
6	6	Calling prog name (as other XRF card images)
13, 25, 37, 49, 61	8	Called libe rtn name (left just., trailing blanks)
21, 33, 45, 57, 69	4	Size needed (EBCDIC decimal bytes, rt just. with leading zeros)

The last card image may have 1–5 fields.

The XRF9 card image will contain references to included source members. The format of this card is:

Start Column	Number of Columns	Contents
1	5	.XRF8
6	6	Calling program name
14, 23, 32, 41, 50, 59	8	Name of included source member.

The last card image may have 1–6 fields.

11.4 COMPILER SYSTEM REQUIREMENTS

The JOVIAL compiler requires a certain amount of peripheral I/O equipment in order to function.

The I/O requirements to run under MVS are as follows:

DDNAME	R/O*	Type	Notes
SYSPRINT	R	PRINTER	SYSOUT
SYSIN	R	DASD	JOVIAL source code
SYSPUNCH	R	DASD	XREF output
SYSUT1	R	DASD	Work space: output from compiler, input to BAL
SYSUT2	R	DASD	Work space
SYSUT3	R	DASD	Work space
SYSUT4	R	DASD	Work space
SYSUT5	R	DASD	Work space
SYSUT6	R	DASD	Work space
CMPTAB	O	DASD	Required when COMPOOL is used

*R = Required, O = Optional

11.5 COMPILER DIAGNOSTICS

There are four levels of diagnostic severity: fatal, major, serious, and warning. Only the first two will halt compilation.

A fatal error occurs only when a compiler table overflows. When the phase in which the fatal error occurred is completed, the statement that caused the fatal error and the corresponding diagnostic message, as well as any other diagnostics found by the compiler, are placed on the system output unit by the diagnostic processor. Compilation is then abandoned and control is returned to the monitor.

A major error is an error which makes any further processing meaningless (for example, a nested procedure declaration or an invalid table declaration). When this error is encountered, an internal switch is set, but compilation continues until the phase is ended. At that time, control is transferred to the diagnostic processor as if a fatal error has been found. Compilation stops, and control is returned to the monitor.

Serious and warning errors do not stop compilation. Normally, however, serious errors will inhibit passing of compiler output to the assembler. Often, however, serious errors in Phase I will produce output unacceptable to later phases, causing abrupt termination of the job, and usually a SYSDUMP. For this reason the ASSEMBLE option should be avoided, especially during initial attempts to compile a program. The ASSEMBLE option is aborted by any major or fatal errors. Under MVS, the COND field should be used.

When a serious error occurs, no BAL coding will be produced for the erroneous statement. Instead, the original JOVIAL statement and a diagnostic message will be listed.

If any errors occur, those statements causing errors and the corresponding diagnostic messages are placed on the system output unit. Warning errors do not inhibit assembly, but may cause unexpected results.

The diagnostic messages issued are listed in Table 11−1 alphabetically, indicating their severity: fatal, major, serious, or warning. Words in a message that may vary from situation to situation (e.g., the name of a table or an item) are denoted by "******". If the variable word is the first word of the message, the message is listed alphabetically by the second word. The 2-character alphanumeric diagnostic code is given in parentheses after each message. This code is primarily for internal compiler use.

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES

A CLOSE ENDED WITH THE EXIT OF AN IF PENDING (33)

Serious.

A CLOSE MAY NOT BE THE EXIT OF AN IF (24)

Serious

A PRESET CONSTANT MUST BE PRECEDED BY A TABLE ITEM DECLARATION (AA)

Serious. Initial values can be assigned only to table items.

A PROC ENDED WITH THE EXIT OF AN IF PENDING (2E)

Serious.

A STATUS CONSTANT MAY NOT BE DEFINED BY A PARAMETER ITEM (AE)

Serious.

A SYMBOL HAS MORE THAN SIX OR LESS THAN TWO CHARACTERS (18)

Serious.

ALL STATUS VALUE CONSTANTS IN THIS STATUS ITEM ARE NOT UNIQUE (B7)

Warning. Status value constants within an ITEM statement must be unique although there may be duplicates among different items. All references to this constant will yield the first value.

ARRAY ITEM IS USED WITH INCORRECT NUMBER OF SUBSCRIPTS (65)

Serious. The number of subscripts used to refer to an item in an array must equal the number of dimensions in the array.

***** ARRAY OR STRING MUST BE SUBSCRIPTED (F9)

Serious.

ARRAY MAY NOT CONTAIN STATUS ITEMS (C2)

Major.

ASSIGN STATEMENT NOT WITHIN DIRECT/JOVIAL BRACKETS (A5)

Serious. An ASSIGN statement has meaning only in direct code.

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

****** BAD OP CODE. POSSIBLE MACHINE OR COMPILER ERROR (F4)

Serious.

BIT/BYTE MAY NOT BE USED IN AN EXCHANGE STATEMENT (86)

Serious.

BOUNDARY RESTRICTIONS VIOLATED IN THIS ITEM DECLARATION (CA)

Major. In programmer-allocated ITEM statements, the programmer must ensure that: (1) integer are fixed point items and EBCDIC or ASCII items of four or fewer characters do not cross word boundaries (2) EBCDIC and ASCII items of more than four characters do not cross two word boundaries and (3) floating point items are stored in fullwords.

BRANCH ADDRESS IN A SWITCH DECLARATION IS NOT A STATEMENT LABEL (7F)

Serious.

CALLED PROCEDURE/FUNCTION IS NOT REENTRANT (88)

Serious.

CLOSED STATEMENT NOT FOLLOWED BY A BEGIN STATEMENT (9C)

Warning. A closed compound procedure has no heading. Body must be enclosed in BEGIN−END brackets. BEGIN assumed.

CLOSE STATEMENT NOT PRECEDED BY A TRANSFER OR CLOSE (15)

Warning. Closed compound procedures must be called, not executed in line.

COMPARISON REQUIRES MORE SIGNIFICANCE THAN ONE MACHINE WORD (78)

Warning.

COMPILER RECORD EXCEEDED. STATEMENT IS TOO LONG (90)

Serious.

COMPILER TABLE EXCEEDED. STATEMENT IS TOO LONG OR COMPLEX (75)

Serious. Suggest segmenting statement.

****** COMPILER TABLE OVERFLOWED. COMPILATION ABANDONED (FF)

Fatal. Suggest segmenting program, or running with more storage (see Appendix G)

COMPOOL REQUESTED BUT NO TAPE ATTACHED (23)

Serious. The compool tape must be on an attached input/output unit if a compool is requested.

COMPOOL VARIABLE APPEARED BEFORE ITS DYNAMIC EQUATE STATEMENT (38)

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

Serious.

CONSTANT SET INTO AN ENTRY MUST BE ZERO (83)

Serious.

****** DATA NAME IS NOT UNIQUE (F7)

Major. May also appear as serious diagnostic.

DIVISION WILL PRODUCE A QUOTIENT WITH MORE THAN 31 INTEGER BITS (77)

Warning. Quotient will be rounded.

DOUBLE WORD CONSTANT TO CONSTANT COMPARE IS INVALID (82)

Serious.

DOUBLE WORD EBCDIC OR ASCII ITEM COMPARED WITH SIGNED ITEM (68)

Serious.

DUPLICATE TABLE DECLARED WITHOUT ORIGINAL TABLE DECLARATION (94)

Major.

DYNAMIC EQUATE BASE ITEM INVALID OR IN INVALID TABLE (DD)

Serious.

EBCDIC OR ASCII ITEM APPEARS IN A FLOATING POINT EXPRESSION (66)

Serious.

EBCDIC OR ASCII ITEM APPEARS IN AN EXPONENTIATION EXPRESSION (67)

Serious.

****** ELEMENT INVALIDLY PLACED IN EQUATE STATEMENT (F2)

Serious.

END OF INPUT FOUND BEFORE THE END OF A TABLE DECLARATION (C8)

Major.

EQUATE STATEMENT CONTAINS AN INVALID ELEMENT (2C)

Serious.

EQUATE STATEMENT MAY NOT BE IN A PROGRAMMER−ALLOCATED TABLE (D6)

Serious. Table items that share storage in a programmer-allocated table are simply described by the programmer.

****** EQUATED ITEM NOT IN THIS TABLE (F5)

Serious. Table items can be equated only to other table items in the same compiler-allocated table.

****** EXIT ALWAYS TAKEN BY A COMPARE IN THIS IF STATEMENT (F3)

Warning. Variable word will be "TRUE" or "FALSE".

FIELD FOLLOWING DIRECT ON THIS CARD WAS IGNORED (CD)

Warning. Only an ASSIGN statement can be on the same card image with a DIRECT bracket. Any other statement is ignored.

FIRST AND THIRD PARAMETERS OF A FOR MAY NOT BE NEGATIVE (35)

Serious.

FOR INDEX IS NOT UNIQUE IN THIS FOR RANGE (CC)

Serious.

FOR MUST BE FOLLOWED BY INDEX AND EQUAL SIGN (5A)

Serious.

FOR STATEMENT HAS INVALID PARAMETER(S) (73)

Serious.

IFEITH DECLARED WITHOUT ANY ORIF'S FOLLOWING (3A)

Serious.

ILLEGAL PLACEMENT OF IF STATEMENT (1D)

Serious.

ILLEGAL REGISTER IN RESERVE OR RELEASE — REST OF CARD IGNORED (E5)

Warning.

ILLEGAL TO LABEL THIS STATEMENT (19)

Serious.

IMPLICITLY CALLED LIBRARY ROUTINE NOT ON LIBRARY TAPE (81)

Serious.

IMPROPER TERMINATION OF A DATA DECLARATION (B4)

Serious. ITEM statement must terminate with a $. A $ is assumed.

INCONSISTENT MODE OR SIGN OF A PRESET CONSTANT (AC)

Serious. Initial values assigned to table items must conform to the description of the item.

INDEX IN THIS STATEMENT WAS NOT PREVIOUSLY DEFINED (98)

Serious. An index must be defined in a FOR statement and remains defined only in the range of the FOR.

INDEX IN THIS TEST STATEMENT WAS NOT PREVIOUSLY DEFINED (93)

Serious. An index must be defined in a FOR statement and remains defined only in the range of the FOR.

INVALID CHARACTER IN THIS STATEMENT (01)

Serious.

INVALID ELEMENT AT THE START OF AN EXPRESSION (40)

Serious.

INVALID ELEMENT AT THE START OF THE LEFT TERM (49)

Serious.

INVALID ELEMENT FOLLOWS THE WORD ASSIGN (5E)

Serious.

INVALID ELEMENT IN A FUNCTION CALL (58)

Serious.

****** INVALID ELEMENT IN AN EXPRESSION (FD)

Serious.

INVALID ELEMENT IN AN IF STATEMENT (64)

Serious.

INVALID EXPRESSION IN AN IF STATEMENT (62)

Serious.

INVALID FIELD SPECIFIED IN A MODIFIED ITEM (7E)

Serious.

INVALID FORM OF A BIT/BYTE MODIFIER ARGUMENT (48)

Serious.

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

INVALID FORM OF A CLOSE STATEMENT (0B)

Serious.

INVALID FORM OF A DUMMY ITEM DECLARATION (AF)

Serious. A data declaration of a parameter in the heading of the function or procedure is incorrect.

INVALID FORM OF A DUMMY PARAMETER (63)

Serious. A parameter given in the PROC statement is incorrect.

INVALID FORM OF A FIXED POINT CONSTANT (05)

Serious.

INVALID FORM OF A FLOATING POINT CONSTANT (04)

Serious.

INVALID FORM OF A FOR STATEMENT (5B)

Serious.

INVALID FORM OF A FUNCTION CALL (59)

Serious.

INVALID FORM OF A GOTO STATEMENT (5D)

Serious.

INVALID FORM OF A HEXADECIMAL CONSTANT (02)

Serious.

INVALID FORM OF A PRESET CONSTANT LIST (AB)

Serious. Initial values can be assigned to table items only. They are enclosed in BEGIN−END brackets and follow the ITEM statement.

INVALID FORM OF A PROC (61)

Serious.

INVALID FORM OF A PROCEDURE CALL (55)

Serious.

INVALID FORM OF A REMQUO STATEMENT (57)

Serious.

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

INVALID FORM OF A RETURN STATEMENT (0E)

Warning.

INVALID FORM OF A RIGHT TERM (53)

Serious.

INVALID FORM OF A SINGLE ITEM OR PARAMETER ITEM DECLARATION (B0)

Serious.

INVALID FORM OF A STATUS CONSTANT (09)

Serious.

INVALID FORM OF A STOP STATEMENT (21)

Warning.

INVALID FORM OF A STRING DECLARATION (BF)

Major.

INVALID FORM OF A SUBSCRIPT SWITCH DECLARATION (2B)

Serious.

INVALID FORM OF A TABLE DECLARATION (31)

Major.

INVALID FORM OF A TABLE ITEM DECLARATION (BC)

Serious.

INVALID FORM OF A TERM STATEMENT (AD)

Warning.

INVALID FORM OF A TEST STATEMENT (92)

Serious.

INVALID FORM OF AN ALL MODIFIER ARGUMENT (5C)

Serious.

INVALID FORM OF AN ARRAY DECLARATION (99)

Major.

INVALID FORM OF AN ASCII CONSTANT (08)

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

Serious.

INVALID FORM OF AN ASSIGN STATEMENT (5F)

Serious.

INVALID FORM OF AN ASSIGNMENT OR EXCHANGE STATEMENT (4F)

Serious.

INVALID FORM OF AN EBCDIC CONSTANT (07)

Serious.

INVALID FORM OF AN ENT ARGUMENT (46)

Serious.

INVALID FORM OF AN EQUATE STATEMENT (D7)

Serious.

INVALID FORM OF AN IMBEDDED COMMENT (0A)

Serious.

INVALID FORM OF AN INTEGER CONSTANT (06)

Serious.

INVALID FORM OF AN ITEM SWITCH DECLARATION (27)

Serious.

INVALID FORM OF PROGRAM NAME (1B)

Serious. A program name must be a valid symbolic name.

INVALID FORM OF STATEMENT LABEL ON DIRECT OPERATOR (CF)

Serious.

INVALID LABEL IN A TERM STATEMENT (A1)

Warning.

INVALID MEMBER NAME (8C)

Warning. The INCLUDE pseudo-op is ignored.

INVALID NUMBER OF ARGUMENTS IN THE CALL (70)

Serious. The number of parameters in a call to a function, procedure, or library defined procedure must equal the number of arguments in the PROC statement. Also used when null arguments appear in a call to a routine which may have code generated in line. In this case it may be serious or warning.

INVALID NUMBER OF INPUT ARGUMENTS IN A FUNCTION CALL (74)

Serious. The number of parameters in a function call must equal the number of input parameters in the PROC statement identifying the function. Also used when null arguments appear in a call to a routine which may have code generated in line.

INVALID OUTPUT ARGUMENT IN A PROCEDURE CALL (56)

Serious.

****** INVALID PLACEMENT OF A STATEMENT LABEL (FC)

Serious. Statement labels must precede the statement being labeled. The label of a compound statement may precede or follow the BEGIN bracket.

INVALID PLACEMENT OF RELATIONAL OR LOGICAL OPERATOR (3E)

Serious.

INVALID REGISTER SCALE FACTOR IN ASSIGN STATEMENT (85)

Serious.

INVALID SCALING FACTOR IN THIS FIXED POINT ITEM DECLARATION (B5)

Warning. Scaling factor must not exceed number of bits in items and absolute maximum is 31.

INVALID SCALING FACTOR OR LENGTH IN A FIXED POINT CONSTANT (B9)

Warning. Maximum for scaling factor and length is 31. Scaling factor must not exceed length.

INVALID STARTING BIT NUMBER IN THIS STRING DECLARATION (D3)

Major.

INVALID TERM IN AN ASSIGN STATEMENT (60)

Serious.

INVALID TERM MODIFIED BY NENT OR NWDSEN (52)

Serious.

INVALID TO EQUATE A TABLE WITH PRESET CONSTANTS (DE)

Serious. If initial values are assigned to a table, the table cannot share storage with any other table.

INVALID TO LABEL A NON−OPERATIVE STATEMENT (1E)

Warning. Label ignored.

INVALID USE OF A DECIMAL POINT (03)

Serious. Decimal points can be specified only for fixed-point, floating-point, EBCDIC, or ASCII fields.

INVALID USE OF A STATUS CONSTANT (45)

Serious. Status value constants must not be used in parameter ITEM statements of ARRAY statements.

INVALID USE OF ENT (51)

Serious.

INVALID USE OF EQUAL SIGNS (3D)

Serious.

INVALID USE OF EXPONENTIATION (43)

Serious. JOVIAL form of exponent is (*...*).

INVALID USE OF 'LOC' FUNCTION (87)

Serious.

INVALID USE OF NENT OF FIXED LENGTH TABLE (7D)

Serious. The value of NENT is assigned by the compiler for fixed-length tables and cannot be changed during program execution.

INVALID USE OF NOT (3F)

Serious.

INVALID USE OF NWDSEN (4C)

Serious.

****** INVALID USE OF PARAMETER ITEM (F8)

Serious. The value of parameter items cannot be changed during program execution. Parameter items must not be used instead of compiler-allocated or programmer-allocated items to describe entries in tables. Only integer parameter items can be used to provide constant information in data declaration (e.g., start-word, start-bit).

INVALID USE OF PARAMETERS (41)

Serious. Parentheses must be paired. They are used in arithmetic expressions and IF statements to indicate precedence. They surround subscripts, objects or modifiers, object of ABS, and exponents.

****** INVALID USE OF RESERVED JOVIAL WORD (FB)

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

Serious. Word ****** is a reserved JOVIAL word and can be used only for its specified purpose. May also appear as a warning diagnostic.

INVALID USE OF SUBSCRIPTING (42)

Serious.

****** INVALID USE OF TABLE NAME (FA)

Serious.

ITEM NAME IN DUPLICATE TABLE IS NOT UNIQUE WITH SUFFIX APPENDED (96)

Serious.

ITEM NAME IN DUPLICATE TABLE IS TOO LONG WITH APPENDED SUFFIX (95)

Serious. Symbolic names must not exceed six characters. Suffix letter on duplicate table name is used to distinguish items in duplicate table.

JOVIAL BRACKET MISSING (CE)

Major.

JOVIAL BRACKET NOT PRECEDED BY DIRECT BRACKET (A6)

Serious. Brackets must be paired.

JOVIAL PSEUDO−OP IS INVALID OR CONTAINS ILLEGAL FIELD (D0)

Warning. Applies to a “period-blank” pseudo-op.

LEFT PARENTHESIS MUST FOLLOW ABS (50)

Serious.

LESS THAN TWO NAMES IN THIS EQUATE STATEMENT (D9)

Serious.

LESS THAN 2 STATES IN THIS STATUS TYPE STRING DECLARATION (D5)

Major.

LIBRARY PROGRAM NOT INTRODUCED BY PROC OR DIRECT (9B)

Major.

MAXIMUM NESTING LEVEL EXCEEDED (8B)

Warning. The INCLUDE pseudo-op is ignored.

MAXIMUM NUMBER OF PRESET CONSTANTS EXCEEDED−EXCESS IGNORED (BE)

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

Warning.

MEMBER NOT FOUND (89)

Warning. The INCLUDE pseudo-op is ignored.

MISSING ARITHMETIC OPERATOR (47)

Serious.

MISSING LEFT TERM IN THIS STATEMENT (4A)

Serious.

MISSING OPERAND IN THIS STATEMENT (4D)

Serious.

MISSING PARAMETER IN A DUPLICATE TABLE DECLARATION (8D)

Serious.

MOD IS NOT A VALID SYMBOLIC NAME (25)

Serious. The name of the compool given on the START statement must be a valid symbolic name.

MOD ON START CARD NOT FOUND ON COMPOOL (97)

Serious. No compool was found that matched the symbolic name on the START card image.

MODES OF ITEM AND CONSTANT DO NOT AGREE (79)

Warning. The type (integer, fixed-point, etc.) of a constant must agree with the type of the field in which it is to be stored. Field type dominates.

MORE PROCEDURE DECLARATIONS IN ONE PROGRAM THAN ALLOWED (14)

Fatal.

N, M, B, OR D OMITTED IN THIS ITEM DECLARATION (C4)

Warning. Item storage was not designated in TABLE statement. D is assumed.

NAME INVALID ON TERM STATEMENT OF A LIBRARY PROCEDURE (20)

Warning.

NESTED PROCEDURE DECLARATIONS ARE ILLEGAL (11)

Major.

NO. OF PRESET CONSTANTS EXCEEDS NO. OF ENTRIES — EXCESS IGNORED (3C)

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

Warning.

NON−INTEGER PARAMETER ITEM INVALIDLY USED IN A DATA DECLARATION (30)

Major. Only integer parameter items can be used instead of integers to describe data (e.g., number of bits per entry, size of table).

NON−OPERATIVE STATEMENT MAY NOT BE IN THE BODY OF PROC OR CLOSE (DF)

Serious. All data declarations in functions and procedures must be in heading. Closed compound procedures must not contain data declarations.

NULL ARGUMENTS SHOULD NOT BE USED IN CALL TO THIS PROCEDURE (37)

Warning or Serious.

NUMBER BEADS/WORD IN STRING DECLARATION IS ZERO OR MORE THAN 32 (32)

Major.

NUMBER OF BEADS PER WORD IS TOO GREAT IN THIS STRING DECLARATION (D2)

Serious.

NUMBER OF BITS IN THIS DATA DECLARATION IS OMITTED OR INVALID (B2)

Warning. Size in bits of integer and fixed-point fields must be specified. Range of size is from 1 to 32 bits. Maximum size is assumed.

NUMBER OF BITS SPECIFIED IN THIS DECLARATION IS INCORRECT (C5)

Warning. Size in bits of integer and fixed-point fields must be given. Range is from 1 to 32 bits. Maximum size is assumed.

NUMBER OF BITS SPECIFIED IN THIS STRING DECLARATION IS INCORRECT (D4)

Major.

NUMBER OF STATUS VALUES IN THIS ITEM DECLARATION IS INCORRECT (B1)

Serious. Number of status values given does not agree with number specified in the programmer allocated ITEM statement.

NUMBER OF WORDS PER TABLE ENTRY ARE UNEQUAL (84)

Serious.

OPERATIVE STATEMENT INVALID IN THE HEADING OF A PROC (E3)

Serious. A function or procedure heading consists of data declaration only.

ORIF/END MUST BE SEQUENCED AS FALSE EXIT OF IFEITH/ORIF (39)

Major. Either ORIF was found where not expected, or some other statement was found where compiler expected ORIF or END.

PARAMETER ITEMS MAY NOT BE DECLARED IN A TABLE (C3)

Serious.

POOL SPECIFIED WITHOUT TAPE ID AND/OR MOD FOLLOWING (0F)

Serious. If a compool is requested on the START statement, it must be identified by the compool tape-ID and specified compool-ID.

PREMATURE EOF. POSSIBLE MACHINE OR COMPILER ERROR (EF)

Fatal.

PRESET CONSTANT INTRODUCED BY BEGIN, BUT NO CONSTANT FOLLOWED (9F)

Warning.

PRESET CONSTANT TOO LONG FOR ITEM (34)

Warning.

PRESET CONSTANTS MUST BE TERMINATED BY AN END (E2)

Warning. Initial values, assigned to table items, must be enclosed in BEGIN−END brackets and follow the ITEM statement. END bracket is assumed.

PROC DUMMY ITEM INCOMPATIBLE WITH USE OF LABEL OR TABLE NAME (6F)

Serious.

PROC NAME DOES NOT MATCH START CARD NAME IN LIBE COMPILATION (9A)

Major.

PROCEDURE CALL IS WITHIN RANGE OF ITS PROCEDURE DECLARATION (26)

Serious. A procedure cannot call itself.

PROCEDURE DECLARATION — DUMMY PARAMETERS NOT ALL DEFINED (10)

Major. All input and output parameters must be declared in the procedure heading.

PROCEDURE DECLARATION HAS A DUPLICATE DUMMY PARAMETER (13)

Major. Input and output parameters must be unique within the procedure although they may duplicate names in other regions of the program.

PROCEDURE DECLARATION WITHOUT A BODY BEGIN (D8)

Major. The body of a procedure (operative statements) must be enclosed in BEGIN−END brackets.

TABLE 11−1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

PROCEDURE DECLARATION WITHOUT A BODY END (DA)

Major. The body of a procedure (operative statements) must be enclosed in BEGIN−END brackets.

PROGRAM DOES NOT CONTAIN A STOP STATEMENT (E1)

Warning.

PROGRAM ENDED WITH THE TRUE OR FALSE EXIT OF AN IF (D1)

Serious.

RECURSIVE INCLUDED MEMBERS (8A)

Warning. The INCLUDE pseudo-op is ignored.

RESERVE STATEMENT WITHIN DIRECT CODE OR RANGE OF A FOR LOOP (E4)

Warning.

RETURN STATEMENT IS NOT IN THE RANGE OF A PROC OR CLOSE (0D)

Serious. A RETURN is meaningful only within a function, procedure, closed compound procedure, or library defined procedure.

RIGHT AND LEFT TERM IDENTITY. NO CODE PRODUCED (80)

Warning. In Assignment or Exchange statement, right term and left term are the same. Statement was ignored.

RIGHT TERM MISSING IN THIS STATEMENT (4E)

Serious.

SCALING FACTORS OF ITEM AND CONSTANT DO NOT AGREE (7A)

Warning. Constant and item description do not agree. Item description dominates.

SIGN DESIGNATION IN THIS ITEM DECLARATION IS NOT S OR U (B3)

Warning. Only S or U can be used; S is assumed.

SIGNS OF ITEM AND CONSTANT DO NOT AGREE (7B)

Warning. Constant sign does not agree with item field.

SIZE OF PROGRAMMER ALLOCATED TABLE ENTRY IS INVALID (8F)

Major.

SIZE OF THIS ARRAY EXCEEDS 16,777,216 BYTES (C7)

Major.

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

SIZES OF ITEM AND CONSTANT DO NOT AGREE (7C)

Warning. Constant does not fit in item field. Constant is truncated.

START CARD SPECIFIES BOTH CLOSE AND LIBE OPTIONS (1A)

Serious.

START CARD SPECIFIES CLOSE OPTION BUT DOES NOT SPECIFY NAME (1C)

Serious. A closed program is executed only if called from another program. It may be called by name.

START STATEMENT ENCOUNTERED AFTER FIRST STATEMENT PROCESSED (A4)

Serious. START must be the first statement in any JOVIAL program. It must not appear elsewhere in the program.

STATEMENT HAS A TERM MIXING OR IMPROPERLY USED (4B)

Serious.

STATEMENT HAS MORE THAN ONE LABEL (22)

Warning. Second label ignored.

STATEMENT HAS MORE THAN 2000 CHARACTERS (16)

Serious.

STATEMENT HAS MORE THAN 256 ELEMENTS (17)

Serious.

STATEMENT INVALID IN THE BODY OF A TABLE DECLARATION (A0)

Serious. No operative statements can appear among data declaration statements describing a table entry.

STATEMENT LABEL IS NOT UNIQUE (1F)

Serious.

STATUS CONSTANT NOT VALID FOR THIS STATUS ITEM (2F)

Serious.

STRING DECLARATION INVALID IN A COMPILER ALLOCATED TABLE (9E)

Serious.

STRING DECLARATION MAY NOT APPEAR OUTSIDE OF A TABLE DECLARATION (A3)

Serious.

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

****** SUBROUTINE NAME IS NOT UNIQUE (F6)

Serious. Defined procedure name ****** is not unique.

SUBSCRIPT OMITTED ON SUBSCRIPT SWITCH CALL (6B)

Serious.

SWITCH DECLARATION CONTAINS NO VALID TESTS OR BRANCHES (0C)

Serious.

SYNTAX LIST OVERFLOW — STATEMENT HAS MORE THAN 20 LEVELS (B8)

Serious.

TABLE DECLARATION NOT FOLLOWED BY A BEGIN STATEMENT (9D)

Major. ITEM and STRING statements describing entry format must be enclosed in BEGIN–END brackets.

TABLE ITEM DECLARATION INCONSISTENT WITH N, M, OR B (A8)

Serious. The table item description conflicts with the packing factor given in the table description.

TABLE ITEM LIST INVALID OR HAS MORE THAN 4096 CHARACTERS (C0)

Major.

TERM STATEMENT MUST FOLLOW THE PROC END IN A LIBE PROGRAM (36)

Serious.

TEST STATEMENT IS NOT IN THE RANGE OF A FOR STATEMENT (91)

Serious. TEST is meaningful only in range of a FOR.

THE CALLED PROCEDURE WAS NOT DECLARED (6E)

Serious. Check the spelling of the procedure name. Procedure and calling program must be compiled together or must be in the library.

THE CALLED SWITCH WAS NOT DECLARED (69)

Serious. Check the spelling of the switch name. SWITCH statement must be in the same region as the switch call.

THE FUNCTION CALLED WAS DECLARED AS A PROCEDURE (71)

Serious. Ensure that no equal sign appears among parameters in the PROC statement.

THE FUNCTION CALLED WAS NOT DECLARED (72)

Serious. Check the spelling of the function name in the PROC statement. Function must be compiled with the calling program or must be in the library.

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

THE ITEM IN THIS SWITCH CALL MAY NOT BE SUBSCRIPTED (6A)

Serious. Subscript specified for single item.

THE LAST OR ONLY STATEMENT IN THE RANGE OF A FOR MAY NOT BE AN IF (C1)

Serious.

THE NUMBER OF CHARACTERS IN THIS ITEM DECLARATION IS INVALID (B6)

Warning. EBCDIC and ASCII items may only be 1 to 8 characters long.

THE PROCEDURE CALLED HAS ARGUMENTS OR IS A FUNCTION (3B)

Serious. The procedure called from a SWITCH has input or output arguments or is defined as a function.

THE PROCEDURE CALLED WAS DECLARED AS A FUNCTION (6D)

Serious. In procedures, an equal sign must separate input and output parameters in the PROC statement.

THE TRUE EXIT OF AN IF STATEMENT MAY NOT BE A FOR STATEMENT (E0)

Serious.

THERE IS NO TERM STATEMENT IN THIS PROGRAM (8E)

Warning. All JOVIAL programs must end with a TERM statement. Program execution will begin with the first operative statement.

THIS EQUATE STATEMENT HAS TWO OR MORE COMPOOL NAMES (DB)

Serious.

THIS IS AN END WITHOUT A MATCHING BEGIN (A9)

Major.

THIS LITERAL IS TOO LARGE TO REPRESENT IN ONE MACHINE WORD (BA)

Warning. The constant is larger than the maximum permitted.

THIS PROGRAM HAS NO OPERATIVE STATEMENT (C6)

Major.

THIS STATEMENT CANNOT BE CLASSIFIED (A7)

Serious. Check statement format and spelling.

TOO MANY LOGICAL OPERATORS IN THIS STATEMENT (2D)

Serious.

TABLE 11–1. JOVIAL DIAGNOSTIC MESSAGES (Continued)

TRANSFER POINT IS NOT A STATEMENT LABEL IN THE SAME REGION (6C)

Serious. May also appear as a warning diagnostic.

TWO ARITHMETIC OPERATORS IN A ROW (44)

Serious.

****** UNDEFINED DATA NAME (FE)

Serious. Name ****** was not declared. May also appear as a warning diagnostic.

UNEQUAL NUMBER OF BEGINS AND ENDS IN THIS PROGRAM (BD)

Major. Brackets must be paired.

UNSIGNED TERM OF MORE THAN 31 BITS APPEARS IN AN EXPRESSION (76)

Warning.

USE OF EXPONENTIATION IS AMBIGUOUS (BB)

Warning.

VALUE IN AN ITEM SWITCH DECLARATION IS NOT UNIQUE (2A)

Serious. Values must not be repeated in the same item SWITCH statement.

VALUE IN AN ITEM SWITCH DECLARATION IS NOT VALID OR IS OMITTED (29)

Serious. Values must agree with the item description.

VARIABLE IN AN ITEM SWITCH DECLARATION IS NOT A VALID NAME (28)

Serious. The item name in the SWITCH statement is not valid. Check spelling.

WORD NUMBER IN THIS DECLARATION INCONSISTENT WITH ENTRY LENGTH (C9)

Warning. In a programmer-allocated TABLE statement, the specified number of words per entry conflicts with the word number indicated by ITEM statement.

WORD OR BIT NUMBER OMITTED IN THIS ITEM DECLARATION (CB)

Major.

12.0 JOVIAL STRUCTURED LISTING

In order to make maintenance of structured JOVIAL programs easier, the compiler will, as an option, produce a structured listing of the JOVIAL source. This relieves the programmer of the responsibility of assuring that the JOVIAL source is in the correct columns.

To request a structured listing, code the 'STRUC' option on the ISPF panel.

Table 12−1 details the use of the print positions on the print lines.

TABLE 12−1. PRINT LINE GENERAL FORMAT

Print Position	Format
1−4	Card Image Number
5	Blank
6−12	Statement Label
13	Blank
14−79	Statement
80−118	Comments
119−132	Column 67−80 of Source Card Image

Exceptions:

1. The following are listed without formatting starting in print position 6.
   a. Comments starting in Column 1 or continuations of comments starting in Column 1.
   b. Direct Code.
   c. The START statement (except that unnecessary blanks are eliminated).
2. Statements preceding the START statement and following the TERM statement are listed without formatting starting in print position 1.

The following is a list of the rules that the compiler will follow in producing the structured listing.

Rules for Statement Formatting:

1. Unnecessary blanks are eliminated.
2. One statement per line.
3. A statement may be spread over several lines, but input from different statements will never appear on the same line. Because a statement may be spread over several lines, a 'logical' page

may require more than one physical page. To prevent this it is recommended that code which would be put on a separate line by JOVFORM should be put on a separate statement by the JOVIAL user (i.e., STATUS values). IF statements should be coded:

```
IF relational expression 1 OR
   relational expression 2 AND
   relational expression 3 $).
```

4. Nothing will be indented more than 40 print positions from print position 6.

5. IF, IFEITH, ORIF, FOR, PROC, CLOSE (CLOSE COMPOUND Procedure declaration), BEGIN (except those preceding tabular item definitions or presets) cause the following statement to be indented two more print positions.

   a. BEGIN, for table definitions and item presets, is idented 1 more than the previous statement and causes the next statement to be idented 1 more than the begin.

6. END (except an IFEITH series END or an END bracketing tabular items or presets) will be intended 2 less than the previous statement and the next statement will be indented 2 less than the END.

   a. An IFEITH series END and the following statement will be indented as far as the IFEITH which started the series.

   b. END, bracketing tabular items or presets, will be idented one less than the previous statement and causes the next statement to be indented one less than the END.

7. Continuations are indented six print positions more than the first line of the statement.

8. If a continuation is forced, the statement is broken off at / + − =,) (*.

9. In logical expressions, AND and OR are placed so as to be the end of a line.

10. Data definition rules. Table 12−2 details the print position assignments for fields on data definition statements.

    a. A field is left-justified in its starting position.

    b. If a field is longer than expected, the following fields are spaced one space between fields until there is enough room to put the field in its proper place.

12.1 COMMENTS

If the comment starts in column 1, the comment and any of its continuations are listed without formatting, starting in print position 6.

All other comments are placed in print positions 80−118. The double quotes (") always are placed in print positions 80, 81 and 117, 118. If a comment will not fit, it is continued on the next line, broken at a blank if possible.

12.2 ERROR MESSAGES

The following error messages may be produced:

END WITHOUT BEGIN

BEGIN WITHOUT END

ORIF WITHOUT IFEITH

'$' IN A COMMENT

TABLE 12–2. DATA DEFINITION FORMAT

Field	Print Position***
TABLE*	14
ITEM (non-tabular)	14
ARRAY*	14
ITEM (tabular)	16
STRING	16
Name	23
Item type	29
Parameter item value	30
No. of Bits or Bytes	31
sign	34
status value**	34
scale	37
start word	39
Start bit	42
Packing factor	45
Skip	47
Bead limit	50
dollar sign ($)	53

*All fields past the name field on TABLE and ARRAY declarations are placed one space between fields

**Each status value is placed on a separate line.

***Add 2 to each print position for data declarations inside procedures.

Appendix A

JOVIAL OPERATORS AND RESERVED WORDS

JOVIAL OPERATORS

A JOVIAL operator is a word or a symbol that instructs the computer to perform some operation. The following list gives the categories of JOVIAL operators.

Declaration	Contextual Modifiers	Relational	Arithmetic	Sequential	Bracket
ITEM	EXIT	EQ	+	GOTO	START−TERM
TABLE	EXTRN	NQ	−	STOP	BEGIN−END
STRING	LINKABLE	LS	*	RETURN	DIRECT−JOVIAL
ARRAY	LIBE	LQ	(*...*)	IF	(...)
SWITCH	REENT	GR	/	FOR	"..."
PROC	POOL	GQ	ABS(...)	TEST	($...$)
CLOSE			REMQUO	IFEITH	
				ORIF	

Assignment	Logical	Modifiers	Separation	Allocation
=	AND	ALL	$	EQUATE
==	OR	ENT	.	
ASSIGN	NOT	NENT	,	
		NWDSEN	blank	
		BIT		
		BYTE		

RESERVED WORDS

Reserved words have special meanings in JOVIAL programs and may be used only for their assigned purpose. They may not be used as symbolic names. The following list gives JOVIAL reserved words.

ABS JOVIAL
ALL

AND	LQ
ARRAY	LS
ASSIGN	
	NENT
BEGIN	NOT
BIT	NQ
BYTE	NWDSEN
CLOSE	OR
	ORIF
DIRECT	PROC
ENT	RETURN
EQ	
EQUATE	START
	STOP
FOR	STRING
	SWITCH
GOTO	
GQ	TABLE
GR	TERM
IF	TEST
IFEITH	
ITEM	

CONTEXTUAL MODIFIERS

Contextual Modifiers are words which have special meaning to the compiler, but which are not strictly reserved. The meaning of these modifiers is determined from the context in which they are found; thus they may be used as symbolic names as long as it is clear that they will not be used ambiguously by the compiler. The following list gives JOVIAL Contextual Modifiers and areas where they are used.

EXIT — TEST statement

EXTRN — CLOSE declaration

LINKABL — CLOSE declaration, START Statement

LIBE — START statement

REENT — START statement

POOL — START statement

TERMS IN A JOVIAL STATEMENT

The following list gives JOVIAL terms. Blanks must not be embedded in terms.

labels

constants* (e.g. V(RED), 3C(A B), and X(4A)).

symbolic names

reserved words

*EBCDIC and ASCII constants can contain embedded blanks because the blank is an acceptable character.

+

−

*

/

=

==

(preceding arithmetic operands and function parameters

) following arithmetic operands and function parameters

($ subscript opener

$) subscript terminator

(* exponentiation opener

*) exponentiation terminator

, separator

$

Although a blank may always be used to separate terms, one is not mandatory except in the following cases:

1. A blank must follow a fixed-point, integer, or floating-point constant unless the first character of the next term is a special character other than a period.

2. If the last character of a term and the first character of the following term are alphabetic, a blank must appear between the terms. For example, a blank must separate a reserved word from a symbolic name.

Appendix B

STATEMENT FORMATS

FORMATS OF DATA DECLARATION STATEMENTS

The following alphabetical list gives the name, purpose, and format of each data declaration statement.

ARRAY

Used to describe in array of one or more dimensions.

ARRAY array-name d_1 d_2 ... d_n field-format $

Compiler-Allocated ITEM

Describes single items and items combined to form compiler-allocated tables.

ITEM item-name field-format $

Compiler-Allocated TABLE

Describes table in which compiler allocates storage.

TABLE table-name { V / R } entry-limit [{ N / M / D / B }] $

Duplicating TABLE

Used to define a new table by repeating the description of a previously defined table.

TABLE modified-table-name [{ V / R } entry-limit L $]

EQUATE

Permits two more single items, or table items, or arrays to share the same storage area.

EQUATE symbolic-name = symbolic-name

[=symbolic-name] ... $

Item SWITCH

Gives values of an item and corresponding statement labels to which transfer is to be made if the item has one of the specified values when the switch is tested.

SWITCH switch-name (item-name) =
(value = statement-label [,value = statement-label]...)
[,else-statement-label] $

Parameter ITEM

Assigns symbolic names to constant values.

ITEM item-name constant-value $

Programmer-Allocated ITEM

Describes items that make up entries in programmer-allocated tables.

ITEM item-name field-format start-word

Start-bit { N / M / D / B } $

Programmer-Allocated TABLE

Describes tables for which the programmer determines the allocation of storage.

TABLE table-name { V / R } entry-limit word-limit $

STRING

Repeats specified variable field formats in a programmer-allocated table entry.

STRING string-name field-format start-word

Start-bit { N / M / D / B } skip bead-limit $

Subscript SWITCH

Specifies statement labels to which transfer is to be made as a specified subscript assumes consecutive values.

```
SWITCH switch-name = (statement label[,[statement label]]...)
[,else-statement-label] $
```

FORMATS OF OPERATIVE STATEMENTS

The following alphabetical list gives the name, purpose, and format of each operative statement.

ASSIGN

Used in direct code to refer JOVIAL-defined data.

```
ASSIGN  R(scale)  =  arithmetic-expression $
        left-term = R (scale)               $
```

Assignment

Sets an item to a specified value.

```
left-term = arithmetic expression $
```

CLOSE

Identifies a closed-compound procedure.

```
CLOSE closed-compound-procedure-name $
```

Exchange

Switches the values of two items or two table entries.

```
item-name == item-name $
ENT (table name) == ENT(table-name)      $
```

FOR

Indicates that the next statement is to be executed a specified number of times under the control of an index.

```
FOR index = start [ ,step
                  ,step, max ] $
```

Function Calls

Part of an operative statement used to call a function

```
function-name  [(input-parameter)
               [,[input-parameter]]...)]  $
```

GOTO

Causes an unconditional transfer, tests a switch, or calls a closed program or closed-compound procedure.

```
GOTO { statement-label
     { switch-name [($subscript$)]
     { closed-program-name [(=left-term)]
     { closed-compound-procedure-name }  $
```

IF

Tests an expression for a true or false condition.

```
IF { simple-condition  }  $
   { complex-condition }
```

IFEITH/ORIF

Tests a series of alternative conditions for true or false.

```
IFEITH          condition  $
simple-or-compound-statement
[ORIF           condition  $
  simple'-or'-compound-statement
  .
  .
  .
  .
  .
  .]
ORIF  { condition } $
      { 1       }
  simple-or-compound-statement
END
```

PROC

Identifies functions and procedures.

Option 1.

```
PROC function-name (input-parameter [, input-parameter]...) $
```

Option 2.

```
PROC procedure-name [([input-parameter]
[,[input-parameter]]..[=[output-parameter]] [,[output-parameter]]...)]      $
```

Procedure Call

Calls procedures.

```
procedure-name [([input-parameter]
[,[input-parameter]]..[=[output-parameter]] [,[output-parameter]]...)]      $
```

REMQUO

Performs integer division and provides a quotient and remainder.

```
REMQUO (dividend,divisor=quotient, remainder) $
```

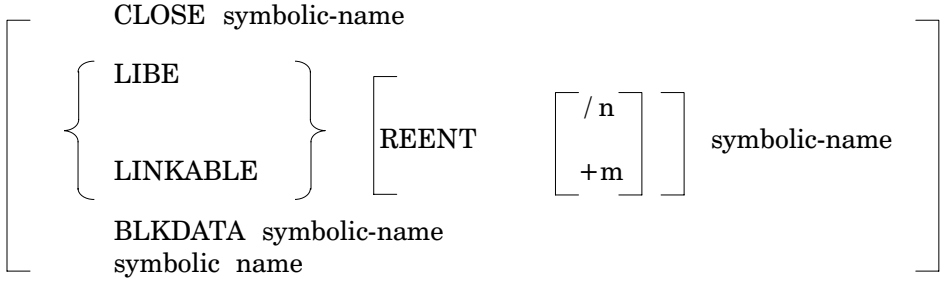
RETURN

Causes return from function, procedure, closed-compound procedure, and library routines before the END bracket of the body is reached.

```
RETURN $
```

START

Control statement that specifies type of program, whether or not compool is to be used, whether serious errors suppress loading, and where the program is to be loaded.




```
[ POOL [ 'compool-id ]] [ASSEMBLE ]
[ load-address ] [ remarks ]
```

STOP

Causes interruption in program execution, and causes return from a closed program to the calling program.

(return-code)

STOP $

statement-label

TERM

Indicates end of a JOVIAL program and the statement at which execution begins.

TERM [statement-label] $

TEST

Causes transfer from within to the end of a FOR range.

TEST [EXIT] [index] $

Appendix C

OPERATIVE STATEMENT SEQUENCING

The following list summarizes the sequence in which operative statements are executed.

Statement	Sequencing
Assignment	Next operative statement.
Exchange	Next operative statement.
FOR (index) $	Next operative statement.
FOR (index, increment) $	Next operative statement (range) repeated until there is a transfer outside the range, then statement transferred to.
FOR (index, increment, max) $	Next operative statement (range) until maximum of index is passed, then statement following the range.
Function Call	First operative statement of the function. After the function is executed, return is to the statement containing the function call.
GOTO statement-label $	Statement whose label is given.
GOTO closed-compound-procedure-name $	First operative statement of closed-compound procedure specified. After the closed-compound procedure is executed, return is to the next operative statement following the call.
GOTO closed-program-name $	First operative statement of closed program specified. After the closed program is executed, return is to the next operative statement following the call.
GOTO switch-name $	Switch statement specified. If equality is found, or a default is specified, control is transferred to the statement whose label is specified. If equality is not found, return is to the next operative statement following the call.
IF condition $	If condition is true, next operative statement. If condition is false, second operative statement.
IFEITH condition $	If condition is true, next operative statement. If condition is false, first ORIF statement.

Statement	Sequencing
ORIF condition $	If condition is true, next operative statement. If condition is false, next ORIF (if present) or first operative statement following end of IFEITH/ORIF group.
Procedure call	First operative statement of procedure. After the procedure is executed, return is to the statement immediately following the call.
REMQUO	Next operative statement.
RETURN $	In functions, operative statement containing the call. In procedures and closed-compound procedures, operative statement immediately following the call.
STOP $	In closed program, statement following the call in the calling program. Terminates program unless it is a closed program.
STOP statement-label $	Statement with specified label when programming is restarted, at the console.
TEST $	End of range of FOR containing TEST.
TEST index $	End of range of FOR controlled by specified index.
TEST EXIT $	Statement following end of range of FOR containing TEST EXIT.
TEST EXIT index $	Statement following end of range of FOR controlled by specified index.

Appendix D

HEXADECIMAL–DECIMAL CONVERSION TABLE

The tables in this appendix provide for direct conversion of decimal and hexadecimal numbers in the following ranges:

Hexadecimal	Decimal
000 to FFF	0000 to 4095

For numbers outside the range of these tables, add the following values to figures in the tables:

Hexadecimal	Decimal
1000	4096
2000	8091
3000	12288
4000	16384
5000	20480
6000	24576
7000	28672
8000	32768
9000	36864
A000	40960
B000	45056
C000	49152
D000	53248
E000	57344
F000	61440

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583

TABLE D–1. HEXADECIMAL–DECIMAL CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Appendix E

DELETED

Appendix F

LISTING OF JOVIAL SOURCE PROGRAM

```
Line No.                                                    Statement

0001            START MEAN                                  "MAIN PROGRAM MEAN"
0002      TABLE SEARCH R 10 1$                              "TABLE SEARCH HAS TWO ITEMS"
0003       BEGIN
0004        ITEM ARG I 16 U 0 0 D $
0005                BEGIN
0006                 2 0 10 9 80 60 90 44 55 20             "INITIAL VALUES"
0007                END
0008        ITEM FOUND I 16 U 0 16 D$
0009       END
0010      TABLE COUNT R 100 1 $                             "TABLE COUNT HAS TWO ITEMS"
0011       BEGIN
0012        ITEM ADD I 16 U 0 16 D$
0013                BEGIN 1 END
0014        ITEM SQR I 16 U 0 0 D$
0015       END
0016      ITEM TOT I 32 S$                                  "SINGLE ITEMS"
0017      ITEM DEV I 32 S$
0018      ITEM AA I 32 S $
0019      ITEM BB I 32 U$
0020      ITEM CC I 32 U$
0021      ITEM DD I 32 U$
0022      A01.      TOT=0$                                  "FIRST OPERATIVE STATEMENT"
0023                DEV=0$
0024                FOR A=1,1,NENT(COUNT)-1$                "FOR STATEMENT"
0025                  ADD($A$)=ADD($A-1$) +1$
0026                FOR A=ALL(COUNT)$                       "EQUIVALENT TO CARD 0024"
0027                 TOT=ADD($A$)+TOT$
0028                TOT=TOT/NENT(COUNT)$
0029                FOR A=ALL(COUNT)$
0030           BEGIN                                        "BEGIN RANGE OF FOR"
0031                 AA=TOT-ADD($A$)$
0032                 AA=AA(*2*)$
0033                 SQR($A$)=AA$
0034                 DEV=DEV+AA$
0035       END                                              "END RANGE OF FOR"
0036       FOR A=0,1,NENT(SEARCH)-1$
0037       BEGIN                                            "BEGIN FOR RANGE FOR A"
0038                IF ARG($A$) LS ADD($0$) OR ARG($A$)     "COMPLEX IF"
0039                 GR ADD($NENT(COUNT)-1$)$
0040       BEGIN                                            "BEGIN TRUE EXIT"
0041                 FOUND($A$)=0$
0042                 TEST A$                                "TO FOR END FOR A"
0043       END
```

```
Line No.                                                    Statement

0044                AA=NENT(COUNT)-1$
0045                DD=0$
0046                CC=AA/2$
0047                BB=CC$
0048                FOR B = 0,1,6$                          "NESTED FOR"
0049          BEGIN
0050                IF ARG($A$) EQ ADD($BB$)$
0051          BEGIN                                         "BEGIN TRUE EXIT"
0052                FOUND($A$)=ADD($BB$)$
0053                TEST A$
0054          END                                           "END TRUE EXIT"
0055                IF ARG($A$) GR ADD($BB$)$
0056          BEGIN                                         "BEGIN TRUE EXIT"
0057                CC=BB$
0058          A02.  BB=(AA-CC)/2+CC$
0059                DD=CC$
0060                TEST B$                                 "TO FOR END FOR B"
0061          END                                           "END TRUE EXIT"
0062                AA=BB$
0063                CC=DD$
0064                TOTO A02$
0065          END @@B@@                                     "END RANGE CONTROLLED BY B"
0066          END @@A@@                                     "END RANGE CONTROLLED BY A"
0067                 FOR A=0,1,NENT(SEARCH)-1$
0068                 BEGIN
0069                    IF ARG($A$) NQ FOUND ($A$)$
0070                       GOTO ANO$
0071                 END
0072          IF TOT NQ 50 $
0073                 BEGIN
0074          ANO.      CORE(SEARCH,10,5H(ZILCH),1)$   "LIBRARY ROUTINE"
0075                    GOTO OUT$
0076                 END
0077          CORE(SEARCH,10,6H(A  OKAY),1)$            "LIBRARY ROUTINE"
```

```
Line No.                                    Statement

0078        OUT.AA=DD$
0079                    STOP$
0080                    TERM$

                    A OKAY

            00C3E8              00020002
            00C3EC              00000000
            00C3F0              000A000A
            00C3F4              00090009
            00C3F8              00500050
            00C3FC              003C003C
            00C400              005A005A
            00C404              002C002C
            00C408              00370037
            00C40C              00140014

END OF EXECUTION

END OF JOB

ELAPSED TIME 00/00/00
```

Appendix G

JOVIAL COMPILER LIMITS

JOVIAL COMPILER LIMITS

TABLE G−1. JOVIAL COMPILER LIMITS

Variable	Limit
Maximum number of simple EQUATE statements allowed on one table	140
Maximum number of simple equates for tables, arrays, and single items	200
Maximum number of status constants (including those from compool)	6000
Maximum number of parameter items (including compool)	1600
Maximum number of unique FOR indexes	250
Maximum number of statement and switch labels	6800
Maximum number of item switches	210
Maximum number of data names and unique indexes (including compool)	17200
Maximum number of strings (including compool) or equivalent arrays*	1400

*Number given is based on strings, which occupy 4 words/STRING, Equivalent in arrays can be computed using (D+1) words/ARRAY, where D=number of dimensions

TABLE G−2. JOVIAL COMPILER LIMITS (STORAGE−INDEPENDENT)

Variables	Limit
Maximum number of levels in 1 statement	20
Maximum number of preset constants	2,101

TABLE G−2. JOVIAL COMPILER LIMITS (STORAGE−INDEPENDENT) (CONTINUED)

Variables	Limit
Maximum number of parenthesized expressions in an IF statement	38
Maximum number of bytes allowed in an array (product of all the dimensions)	16,777,216

JOVIAL Compiler Procedure Prefix Limits

There are two factors limiting the number of procedure or function declarations allowed in a compilation. One is the storage available at compilation time; the other is the availability of prefixes to be at compilation time; the other is the availability of prefixes to be assigned. The limit is based on the number of routines on the Library (whether referenced or not) plus the number of internal PROC statements. The actual limit is always the lesser of the prefix limit (formulas G1 and G2) and the storage limit (Table G−3). Table G−3 gives estimates based on average number of arguments, as well as actual bytes of storage available. Each procedure (internal or library) requires 16 bytes plus 19 bytes per argument, while each function requires 35 bytes plus 19 bytes per explicit argument (the 35 bytes include the implicit output argument for the function name). Studies have shown an average of 3.1 arguments per routine on a library tape.

$$2P + L \leq 648 \qquad \text{(G1)}$$

and

$$P \geq 107 \qquad \text{(G2)}$$

where:

P is number of PROC statements declared in program, and L is number of routines on the Library dataset (whether referenced or not).

TABLE G−3. JOVIAL COMPILER PROCEDURE/FUNCTION LIMITS

	Limit
Available Storage (bytes)	49800
Maximum number of procedure or function declarations (inc. Library) assuming 3 arguments per procedure	694
Maximum assuming 6 arguments per procedure	383

MVS COMPILER LIMITS

Under MVS the compiler requires a minimum region of 250K in order to operate. This assumes a small to moderate sized program and a small compool (or no compool). With a large compool (NAS compool with 6200 data names) a region of at least 350K is required; programs with much internal data and many segments referenced will need more.

Compool generation (CMPGEN option) will run in the minimum region for small compools. A 6200-data-name NAS compool will require 450K for generation.