ND-00199C

# CMS-2Y PROGRAMMING

## A SELF-INSTRUCTIONAL MANUAL




FLEET COMBAT DIRECTION SYSTEMS SUPPORT ACTIVITY

SAN DIEGO, CALIFORNIA

31 JULY 1981

Table of Contents

Table of Contents

Table of Contents

Table of Contents

# Preface

## Purpose:

The purpose of this manual is to teach the CMS-2Y high-level language to entry level programmers.

## Scope:

The text will cover the language using simple, straightforward explanations, examples and problems so that the student will progress through actual work with each facet of language development. The seldom used and the extremely complex programming areas will be covered in considerably less detail in the Appendices, or will be referenced to other documents. Fundamentals of data processing are presumed to be a part of the student's knowledge base.

It is expected that the entry level programmers who will be assigned to the study of the CMS-2Y language through the use of this manual, will have had (as a minimum) one course in a high-level programming language, such as FORTRAN, PL-1, PASCAL or BASIC. Actual programming experience in any one of the above-mentioned languages is not an absolute prerequisite, but is certainly considered to be advantageous.

## History:

The CMS-2Y software system was developed by the Navy as a computer programming language for use in designing large-scale tactical data processing systems and other real-time computer systems. These systems make stringent time and space demands that require a flexible language with more programmer control of machine features than is provided in most high-level languages.

The original Navy programming language CS-1 (Compiling System-1 developed in the 1950s) was oriented to the CP-642/USQ-20 family of second-generation computers. With the advent of third-generation machines and the forseeable development of still more advanced systems, a more powerful and flexible language was needed.

The first version of CMS-2 (now designated CMS-2Q) was used to write programs for the CP-642 and the AN/UYK-7 computers. This version (which accepts some CS-1 statements) served as a transition from CS-1 to full CMS-2 language programs.

CMS-2M is a version of the language that includes some features oriented specifically to mini-computers, such as the AN/UYK-20 and the AN/AYK-14.

CMS-2Y is the version of CMS-2 which is implemented in its full scope to generate code for the AN/UYK-7 computer. This is the version that is covered in this manual.

How to Use This Manual:

This self-instructional manual has been developed to teach the beginner the fundamentals of programming in the CMS-2Y language.

The format used in a self-instructional manual is quite different from that of a reference manual for experienced programmers. This manual is designed for use as a teaching instrument only.

There is no time limit or speed limit imposed on the student. He or she will progress at his or her own rate of speed. If an area is not completely clear the first time through, it should be reviewed until there are no further questions.

There are large numbers of problems and exercises scattered throughout the manual. In every case, the correct answer is given on the back of the problem page. The student is to work each problem on scratch paper. The answer is then checked against the correct answer. If the answer given by the student is incorrect, the previous pages should be reviewed. There is nothing to keep the student from looking at the correct answer before attempting to work the problem except the realization that he will not learn to program if he does this.

For best results, it is suggested that this manual be studied for not more than two hours at a time and not more than two such (two-hour) sessions a day. There is a great deal of material to be assimilated. Attempting to push through too fast will reduce retention of material covered.

It must also be understood that completion of this manual will not qualify the student as an expert programmer. It will teach him the fundamentals of programming using the CMS-2Y language. He will have acquired the basic tools of programming, but only practical experience as a working programmer can develop the knowledge and skill required to be considered an expert.

Acknowledgements:

This manual was prepared for FCDSSA by the CMS-2 staff of Computer Sciences Corporation. The preparation was led by Dr. James A. Saxon and Rachel S. Treat.

# Chapter 1

## Introductory Features

## A. LANGUAGE DESCRIPTION

CMS-2Y is a high-level programming language with modern state-of-the-art techniques including features such as block structures and structured programming capabilities. It was designed to satisfy Navy requirements for real-time, command and control and tactical data systems applications.

The language is oriented to the solution of problems using English words and algebraic phrases to accomplish its purpose. The arrangement of data and the instructions are handled in a systematic set of English-like and arithmetic statements.

The basic objectives of the language are to provide:

(1) a relatively machine-independent programming language,

(2) programs that are easily read and understood,

(3) program segments that may be independently compiled.

## B. LANGUAGE STRUCTURE

A CMS-2Y program is composed of an orderly set of statements made up of various words, letters, numbers and symbols. These are used to describe the operations to be performed and to define the data to be used during the execution of the program.

A CMS-2Y program, developed to solve a particular problem, is composed of two basic parts:

1. Declarative Statements - These statements provide direction to the compiler and define data to be manipulated. Data declarations are grouped into units called data blocks. With the exception of some highly specialized input/output statements, data declarations result in machine code that is not executable by the computer.

2. Dynamic Statements - These statements manipulate data and describe logic flow. They are grouped into sub-programs called procedures and functions. Dynamic statements cause the generation of machine instructions that are executed by the computer.

The basic CMS-2Y program consists of sub-program statements and data description statements (data blocks). At this point, we are primarily concerned with terminology. All of the above terms will become clear as the student progresses through the manual.

In summary, statements are:

1. Declarative - in data blocks that specify compiler control and describe data areas.

2. Dynamic - in procedures and functions that manipulate data and describe logic. Dynamic statements cause the generation of object code.

---

WORK AREA

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. What are the two basic types of program statements?

2. What are the two general uses of declarative statements?

3. In general, what does a procedure accomplish?

4. Which type of statement is not executed by the computer?

5. Which type of statement is executed by the computer?

CORRECT ANSWERS

1. declarative and dynamic
2. to define data and direct the compiler
3. manipulates data and describes logic
4. declarative statements
5. dynamic statements

---

## C. CHARACTER SET

The CMS-2Y character set consists of letters, digits, the space or blank, delimiters, the terminator symbol and special characters.

1. Letters - the 26 letters of the alphabet A through Z
2. Digits - the 10 Arabic numerals 0 through 9
3. Space or Blank - the space or blank is normally used as a separator
4. Delimiters - these are special symbols having defined meanings for the compiler. Delimiters and spaces are used, as appropriate, to specify and separate parts of a CMS-2Y statement.

Delimiters are:

- `+` addition operator and unary plus (positive sign)
- `-` subtraction operator and unary minus (negative sign)
- `*` multiplication operator
- `/` division operator
- `**` exponentiation operator
- `.` decimal point, octal point (often called radix point) and label delimiter
- `(` beginning enclosure for expressions, lists and other units
- `)` ending enclosure for the above
- `,` (comma) separator for items in a list
- `'` (apostrophe) single apostrophe (use to be discussed later)
- `''` two consecutive apostrophes delimit programmer notes

5. Terminator - $ The dollar sign indicates the end of a CMS-2Y statement.
6. Special Characters - A special character is any character other than those mentioned above that can be input or output through the I/O devices of a particular installation.

## D. IDENTIFIERS

Identifiers are arbitrary names used to identify data units or to label parts of a CMS-2Y program. Names are composed of alphanumeric characters (letters or numbers). The first character of a name must be a letter. Names may be no longer than eight characters. Delimiters are not allowed within identifiers.

EXAMPLES

| Examples of valid names: | | Examples of invalid names: |
|---|---|---|
| A | F4973AC2 | 6DPOY (numeric in first position) |
| ALPHA | CONTR | ELEMENT243 (longer than 8 characters) |
| C46B | ASSIGNED | PAY-REC (delimiters not allowed) |

Any sequence of letters and numbers may be used as long as the basic rules are obeyed, but it is advantageous to devise and use names that will also be meaningful to the programmer and to other people.

There are a number of special words (called Reserved Words) which may not be used as programmer-devised names. These special words activate appropriate compiler operations. A list of Reserved Words may be found in Appendix E of this manual.

---

WORK AREA

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. What separates parts of CMS-2Y statements?
2. What is the meaning of the following delimiters?
   a. *  b. /  c. **
3. What is used at the end of every CMS-2Y statement?
4. What are the four basic rules for programmer-devised names?
5. Special words that are restricted from being used by the programmer as names are called _____.

CORRECT ANSWERS

1. delimiters (arithmetic operators, special symbols) and spaces
2. a. multiplication operator
   b. division operator
   c. exponentiation operator
3. dollar sign $
4. names are composed of letters and numbers-first character must be a letter-maximum length is 8 characters-delimiters not allowed
5. reserved words

---

## E. STATEMENT FORMAT

The compiler receives CMS-2Y programs in a punched card format therefore 80 columns are available for each line of input. A single card image becomes one program line. Columns 1-10 are not used by the compiler and may be used by the programmer for any purpose (i.e. identification, line sequencing, etc.).

The compiler looks at columns 11-80 of the lines as a continuous stream of characters in free format. Column 80 of one line is followed immediately by column 11 of the next line. Parts of a statement are separated by spaces or delimiters and a statement is always ended with a dollar sign ($).

EXAMPLES

1. Columns 1 - 10 / Columns 11 - 80

```
VRBL (DIF1,DIF2) A 16 S 3 $
```

Note that the various parts of this statement are separated by spaces and delimiters (comma and parentheses).

2. All of the following are correct:

```
ALPHA + BETA
ALPHA+ BETA
ALPHA +BETA
ALPHA+BETA
```

Note that a delimiter alone may be used as a separator or any number of spaces may be used between parts of a statement.

In subsequent examples in this manual, columns 1-10 are not shown and all statement lines are assumed to be in columns 11-80.

Remember that:
(1) a single program statement may take up more than one line,
(2) more than one program statement may be written on a single line, and
(3) spaces may be used liberally between parts of the statement for the sake of clarity and readability.

## F. LABELS

A label is a programmer-assigned statement identifier that is placed at the beginning of a dynamic statement. Labels obey all the rules for identifiers and in addition must be immediately followed by a period (.). A label is required only if the statement must be referenced from another location within the sub-program.

EXAMPLE

The following example shows format only and is not intended to be logically connected.

```
        SET LBASE TO NXTIN $
CMS2LBL.  SET INXT TO SAVIN $
    IF PID(INX+2,CLASS) EQ 43
          THEN GOTO CMS2LBL $
```

The identifier CMS2LBL is a label for the statement SET INXT TO SAVIN $.

Note that the statement beginning with IF, although extending over two lines, is only one CMS-2Y statement. This ability to arrange statements in structured, easily read fashion is a benefit of CMS-2Y's free-form format.

---

WORK AREA

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. a. How many columns may be used in writing one line of a CMS-2Y program?

   b. Which columns are not used by the compiler?

2. In the example given above, what is the label?

3. What is the length of a program line?

4. What is the length of a program statement?

CORRECT ANSWERS

1. a. total of 80 columns, but only 70 are usable for the program
   b. 1 through 10
2. CMS2LBL. A label is always ended with a period.
3. as long or as short as the programmer wishes to make it within 70 characters
4. as short as a part of one line or as long as several lines

---

## G. NUMBERS

The CMS-2Y language will accept three different forms of numeric notation: decimal, octal and exponent.

### 1. Decimal:

Decimal notation is as follows: digits 0-9, decimal point and minus (-) sign for negative numbers. Positive numbers do not require a sign, but the + may be used if desired.

EXAMPLES

48 -3 22.04 -14.6 12674358 +14.6

Note that commas are never used to separate segments of large numbers.

### 2. Octal:

Decimal notation is assumed unless octal is specified. If octal is desired, the letter O is used immediately preceeding the parentheses which must enclose the number. As in decimal notation, negative values are indicated by the minus (-) sign which must be placed in front of the O.

EXAMPLES

O(177) -O(177) O(2.76) -O(.276)

### 3. Decimal and Octal Exponent:

If desired, numbers may be expressed in exponent notation. The symbol E (exponent) is used at the end of a number, followed by the exponent value which may be either positive (unsigned) or negative.

EXAMPLES

| Decimal Notation | Exponent Notation |
|---|---|
| 21 x 10**2 | 21E2 |
| 12.3 x 10**-2 | 12.3E-2 |
| 9 x 10**-5 | 9E-5 |

It must be remembered that all numbers within the parentheses are octal numbers in octal notation.

| Octal Notation | Exponent Notation |
|---|---|
| O(1.77 x 10**2) | O(1.77E2) |
| O(10**2)=8**2 | |
| -O(2.46 x 10 **-11) | -O(2.46E-11) |
| O(10**-11)=8**-9 | |

---

WORK AREA

1. Write the following numbers in CMS-2Y acceptable notation.

   a. octal 246
   b. decimal 246
   c. negative 246 decimal
   d. octal 246 times 8 to the second power
   e. decimal 246 times 10 to the fourth power
   f. decimal 24.6 times 10 to the minus 2 power

2. What is the meaning of the following notation?

   a. 2.7E-4
   b. 27E5
   c. O(27E3)
   d. O(2.7E-2)

CORRECT ANSWERS

1. a. O(246)
   b. 246
   c. -246
   d. O(246E2)
   e. 246E4
   f. 24.6E-2

2. a. decimal 2.7 times 10 to the minus fourth power
   b. decimal 27 times 10 to the fifth power
   c. octal 27 times 8 to the third power
   d. octal 2.7 times 8 to the minus second power

---

## H. COMMENTS AND NOTES

Navy Standards require the use of comments and notes in computer programs. Additionally, the use of comments and notes tend to remind the programmer of various key steps in the program. These are particularly helpful if responsibility for a program moves from one person to another.

Lengthy documentation may be placed in a program by using a separate comment statement. Short comments may be inserted within other declarative or dynamic statements by the use of notes.

A comment statement is indicated by the reserved word COMMENT. This is followed by a string of characters (not including $) that gives the documentation information. It is ended with $.

Notes consist of two consecutive apostrophes (''), a string of characters other than $ and two more consecutive apostrophes. The single character representing double quotes (") is not a note delimiter. Notes may be used anywhere that spaces are permitted.

EXAMPLES

```
1.      COMMENT    EDLOPRNT WILL TRANSFER THE DATA IN THE  PRINT
                   LINE TO THE LOADER $
        SET VBINX  TO LOADINX ''SAVE CONTROL WORD INDEX'' $
```

The first example (starting with COMMENT and finishing with $) is the comment statement. The note is contained in double apostrophes in the second example.

```
2.          COMMENT    THIS  PROCEDURE REFORMATS THE DATA IN BUFFER1
                       FOR OUTPUT ON THE  ON-LINE  PRINTER,  STORING
                       THE NEW FORMAT IN BUFFER2 $
```

This example of a COMMENT statement shows a common use - the descriptive documentation of software segments.

```
3.          SET ABFLG TO 0 ''INDICATES BAD DATA ENCOUNTERED'' $
            GOTO NXTSTEP ''SEE WHAT CAN BE SALVAGED'' $
```

The notes accompanying the above statements illustrate documentation of program logic. This can be a reminder for the original programmer and a life-saver for his or her successor.

---

WORK AREA

1. What is used to indicate that a comment is to follow?

2. What is used to enclose a note in CMS-2Y?

3. Why is it a good policy to use comments and notes liberally throughout a program?

CORRECT ANSWERS

1. the reserved word COMMENT

2. double apostrophes ('') on either side of the note

3. as a future reminder of important parts of the program

---

## Chapter 1 Summary

This chapter introduces the basic structure of CMS-2Y, including both the executable and the non-executable statements. It describes the character set used in the language, the programmer developed names (called identifiers), the format for writing program statements, rules for labels, methods used to express numbers both in octal and in decimal notation and the rules for comments and notes.

All of these items are basic to the writing of CMS-2Y programs and most items will be used again and again throughout the manual. The rules contained in the chapter will become an integral part of the programmer's knowledge and will, before long, be used almost automatically by the programmer.

## Chapter 1 Quiz

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. Which type of statement generates non-executable machine code?

2. Which type of statement generates executable machine code?

3. How many spaces should be placed between parts of a statement?

4. Which of the following are incorrect names and what is wrong with each?
   a. DELTA1
   b. 1ALPHA
   c. BETA459A3
   d. GAMMA253

5. In the following example:
   a. how many statements are there?
   b. name the labels.

EXAMPLE

```
        IF FLG1 THEN GOTO LBLJ $
        SET IND1 TO IND3*5 $
LBLK.   SET IND2 TO IND1+15 $
LBLJ.   IF IND2 LT 30 THEN SET FLG2 TO 1 $
        IF FLG3 EQ 0 THEN GOTO LBLK $
        SET FLG1 TO 0 $
```

6. Write the following in CMS-2Y acceptable notation.
   a. octal 1.77 x 8**-3
   b. octal 17.7
   c. octal 177 x 8**2
   d. decimal 1,420.16
   e. decimal 14 x 10**-3
   f. decimal 1.4 x 10**5

7. What is wrong with the following?
   a. O(18E2)
   b. 177
   c. O(17)E2
   d. 17.7E+2

8. a. What is used to delimit notes?
   b. What reserved word is always used with a comment?

## CORRECT ANSWERS

1. declarative statements

2. dynamic statements

3. at least one space, but may be as many as the programmer desires, except that spaces are not always necessary when delimiters are used (e.g. X+Y, ALPHA-BETA)

4. b. first position must be a letter
   c. too long - total of 8 positions are allowed

5. a. 6-each statement is ended with a dollar sign $
   b. LBLK and LBLJ

Note that the programmer devised names are easy to pick out in this example. In the first line - FLG1 and LBLJ, in the second line - IND1 and IND3 (the *5 simply means 'multiply by 5' the amount found in location called IND3).

6. a. O(1.77E-3) The O must always precede the number and the entire number must be enclosed in parentheses.
   b. O(17.7)
   c. O(177E2)
   d. 1420.16 comma not allowed
   e. 14E-3
   f. 1.4E5

7. a. the number 8 is illegal in octal notation
   b. nothing wrong
   c. the entire expression must be enclosed in parentheses (e.g. O(17E2)
   d. nothing wrong, + not required, but may be used

8. a. two consecutive apostrophes - do not use double quote symbol

   b. COMMENT

# Chapter 2

# NUMERIC VARIABLES AND ASSIGNMENTS

## Numeric Variables and Assignments

A CMS-2Y variable is a single data unit used in a program and referenced by its programmer-assigned name. Variables may occupy from one bit to several words in computer memory. All data units are defined in a declaration statement. A variable declaration includes the name or names being declared and (optionally) the type and a preset value. In CMS-2, all variables must be defined before they are referenced in other parts of the program.

The CMS-2Y language includes six types of variables, three numeric and three non-numeric. The non-numeric types will be discussed in Chapter 3. The three numeric types are integer (I), fixed-point (A) and floating-point (F). Numeric variables may be either signed (i.e., able to take on both positive and negative values) or unsigned (i.e., always positive).

### A. NUMERIC VARIABLE DECLARATIONS

FORMAT

| VRBL | identifier or identifiers | type | P | preset value | $ |
|---|---|---|---|---|---|
| reserved word | programmer supplied names(s) | (optional) the kind of value, size and sign | (optional) preset value to follow | (optional) value to which variables are to be preset | |

Explanation:

VRBL - CMS-2 language keyword, indicating a variable declaration. (See Note 1 on the following page)

identifier or identifiers - programmer devised name(s) by which the variable(s) will be referenced. A maximum of 25 names may appear as identifiers in one variable declaration. Multiple names are separated by commas and the entire list must be enclosed in parentheses.

| | |
|---|---|
| type<br>(optional)<br>(see Note below) | - specifies:<br>(1) the kind of values the variable can assume- integer (I), fixed-point (A) and floating-point (F).<br>(2) the size of the variable (for I and A)<br>(3) the number of fractional bits (for A only)<br>(4) the sign (if any) - (for I and A only) Use S for signed and U for unsigned (means always positive). |
| P<br>(optional) | - indicates that a preset value follows. |
| preset value<br>(optional) | - specifies a value to which the variables are to be preset. The value must be compatible with the type of the variable. The effect of a preset is the same as if the value were assigned to the variable at the beginning of program execution. If P is used, then preset value is no longer optional, but must be used. |

Notes:
1. As used in this manual, keyword means a word with special meaning to the compiler. Most, but not all, keywords are also Reserved Words.
2. Throughout the manual the word optional describes parts of a statement that are not always necessary. The programmer must decide which parts are required for his task.
3. Examples of variables will be shown as each type is discussed.

---

WORK AREA

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. (a) What must be included in a variable declaration?
   (b) What may be included in a variable declaration?

2. Show how you would write a list of identifiers titled KIM1 through KIM5.

3. What must be used at the end of every statement?

4. What is the reserved word for variable?

5. What does the letter P in a variable statement indicate?

6. What does the letter U indicate?

CORRECT ANSWERS

1. (a) the reserved word and the name or names being declared
   (b) type (including kind, size and sign as required) and preset value
2. (KIM1, KIM2, KIM3, KIM4, KIM5)
3. a dollar sign $
4. VRBL
5. a preset value follows
6. the item is unsigned

---

The following example shows how the variable declaration format can be used.

FORMAT

| reserved word | identifier or identifiers | type | preset indicator | preset value | end of statement |
|---|---|---|---|---|---|
| VRBL | COUNT | I 8 U | P | 6 | $ |
| variable statement | programmer devised name | integer 8 bits unsigned | preset to 6 | | |

1. INTEGER VARIABLES (I):

Integer variables may take only whole number values. The number of bits in an integer variable must be between 1 and 64. (It may be as small as one bit or as large as 64 bits). One bit of a signed data type (the left-most bit) is required for the sign; therefore total bit length must be one more than the bit length required for the largest value of the data unit.

Note: Since this is the first mention of bits, the student may wish to refresh his memory on bits and bit handling. These items are discussed in Appendix D, Section 2.

## EXAMPLES

| Variable Declaration | Remarks |
|---|---|
| VRBL COUNT I 8 U $ | COUNT is an integer variable, 8 bits in length, unsigned. |
| VRBL (IND1,IND2,IND3) I 32 S $ | IND1,IND2,IND3 are integer variables, 32 bits in length, signed. |
| VRBL NUM I 8 S P -10 $ | NUM is an integer variable, 8 bits in length, signed. It is preset to minus 10. |

---

## WORK AREA

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. True or false - Numeric variables must always be signed.

2. What is the maximum number of bits permitted in an integer variable?

3. Write a variable declaration given the following information:
   There are two integer variables called PERM and PERM1, they are both signed with 20 bits each and preset to -27.

4. In the problem above, what is the bit length of the value of PERM1?

5. What is the size of a signed data unit that specifies 32 bits?

6. Example: VRBL (SEND1,SEND2) I 16 U $
   In the above example, answer the following questions:
   (a) What type of variables are these?
   (b) What is the name of each variable?
   (c) How long is each variable?

7. Example: VRBL PMA I 16 S P 1500 $
   Explain the meaning of the above statement.

CORRECT ANSWERS

1. false - may be either signed or unsigned
2. 64
3. VRBL (PERM,PERM1) I 20 S P -27 $
4. 19 bits - the 20th bit (the left-most) is the sign.
5. 32 bits-the left-most bit becomes the sign, leaving 31 useful bits
6. (a) integer variables
   (b) SEND1,SEND2
   (c) 16 bits
7. PMA is an integer variable, 16 bits in length, signed. It is preset to 1500.

---

Note:

It is the programmer's responsibility to specify an adequate number of bits to represent the maximum values a data unit is expected to assume. For instance, in the examples on page 19, the variable COUNT with eight unsigned bits specified can assume values in the range of 0 through 255. NUM also has eight bits specified, but one bit represents the sign. Usually it is sufficient to specify a size that is known to be large enough, without being overly concerned with the exact number of bits.

## 2. FIXED-POINT VARIABLES

Fixed-point variables provide a means of representing fractional values. The accuracy to which the value is expressed depends upon the number of fractional bits declared. If an attempt is made to assign a value having more fractional bits than the number of bits declared, the excess low-order (least significant) bits will be truncated.

FORMAT

| )L | identifier(s) | type | P (optional) | preset value $ (optional) |
|---|---|---|---|---|
| ierved 'd | programmer supplied name(s) | always A, size as required, S-signed U-unsigned, number of fractional bits | preset value to follow | the value to be preset |

## EXAMPLES

| Variable Declaration | Remarks |
|---|---|
| VRBL PAY A 19 U 7 $ | PAY is a fixed-point variable, 19 bits in length, unsigned, with 7 fractional bits. There are 12 integer bits. |
| VRBL AREA A 19 S 7 $ | AREA is a fixed-point variable, 19 bits in length, signed, with 7 fractional bits. Since 1 bit is used for the sign, there are 11 integer bits. |
| VRBL RATE A 12 U 7 P 3.75 $ | RATE is a fixed-point variable, 12 bits in length, unsgned with 7 fractional bits. It is preset to 3.75 and there are 5 integer bits. |
| VRBL TEMP A 16 S Ø $ | TEMP is a fixed-point variable, 16 bits in length, signed with no fractional bits. |

---

## WORK AREA

1. What is the purpose of using fixed-point variables?

2. How many integer bits are there in the TEMP example above?

3. Explain the meaning of each of the following declarations:
   (a) VRBL DIF A 16 S 3 $
   (b) VRBL TIM A 12 S 8 $
   (c) VRBL TIM1 A 15 U 9 $
   (d) VRBL TIM2 A 24 S 4 $

4. How many integer bits have been assigned in each of the items in problem 3?

5. Write a variable declaration given the following information: There are two fixed-point variables called TAM and TAM1, signed with 14 bits assigned to each with 7 fractional bits. They are preset to 21.

CORRECT ANSWERS

1. to be able to use fractional data

2. 15

3. (a) DIF is a fixed-point variable, 16 bits in length, signed, with 3 fractional bits.
   (b) TIM is a fixed-point variable, 12 bits in length, signed, with 8 fractional bits.
   (c) TIM1 is a fixed-point variable, 15 bits in length, unsigned, with 9 fractional bits.
   (d) TIM2 is a fixed-point variable, 24 bits in length, signed, with 4 fractional bits.

4. (a) 12 (c) 6
   (b) 3 (d) 19

5. VRBL (TAM,TAM1) A 14 S 7 P 21 $

---

3. FLOATING-POINT VARIABLES (F):

Floating-point variables represent values expressed in the format of the target computer's floating-point hardware. On the AN/UYK-7 computer, for example, a floating-point data unit has 16 bits of accuracy for the characteristic and 31 bits of accuracy for the mantissa. Floating-point data units are advantageous when the magnitude of data can vary over a wide range.

FORMAT

| BL | identifier(s) | type | P (optional) | preset value $ (optional) |
|---|---|---|---|---|
| served rd. | programmer supplied name(s) | always F size, sign and fractions determined by the computer | preset value to follow | the value to be preset |

Floating-point operations normally consume great amounts of computer time and therefore are usually not used in programs with stringent time restraints. The programmer may elect to define fixed-point data units instead.

EXAMPLES

| Variable Declaration | Remarks |
|---|---|
| VRBL FPAY F $ | FPAY is a floating-point variable. |
| VRBL FRATE F P 3.75E-4 $ | FRATE is a floating-point variable. It is preset to 3.75x10**-4. |

---

WORK AREA

1. When are floating-point operations useful?

2. Explain the meaning of the following variable declarations:
   a. VRBL (FPAY,FCST,FPAS) F $
   b. VRBL (NUM1,NUM2) F P 1.15E2 $

3. What is one reason for not using floating-point arithmetic if other methods are possible?

4. On the AN/UYK-7 computer, how many bit positions of accuracy are there to the left of the decimal point? How many to the right?

CORRECT ANSWERS

1. when the magnitude of the data can vary over a wide range

2. (a) FPAY, FCST, FPAS are floating-point variables
   (b) NUM1, NUM2 are floating-point variables preset to 1.15X10**2

3. it uses more computer time than non-floating point arithmetic

4. 16 bits, 31 bits

---

## B. SET STATEMENT

Arithmetic calculations and data transfers are accomplished with an assignment statement called a SET statement. The statement specifies the receiving area followed by the data to be transferred.

FORMAT

| | | | | |
|---|---|---|---|---|
| SET | receiving area | TO | source area | $ |
| reserved word | name of receiving area | reserved word | a specified value | |

Explanation:

| | |
|---|---|
| SET | - reserved word indicating an assignment statement |
| receiving area (receptacle) | - name (or names) of the data unit (or units) to receive the source data |
| TO | - reserved word indicating that a source expression follows |
| source area | - the value to be assigned or the name assigned to that value |

### 1. BASIC USES

The simplest examples of SET statements are the assignment of a constant value to a data unit and the transfer of data from one area to another. When a transfer involves values of different types or different bit sizes, loss of significance may occur. (Details of bit handling and methods of converting various numbering systems are covered in Appendix D).

The following variable declarations are used in the examples of SET statements to show the various methods that are available to the programmer and the effective results based on specific actions.

## Variable Declarations

```
VRBL COUNT I 8 U $
VRBL (IND1,IND2,IND3) I 32 S $
VRBL NUM I 8 S P -1Ø $
VRBL PAY A 19 U 7 $
VRBL TIM A 16 S 8 $
VRBL TIM1 A 15 U 9 $
VRBL FPAY F $
```

## EXAMPLES

| | Statement | Remarks |
|---|---|---|
| 1. | SET COUNT TO Ø $ | Assign the value zero to COUNT. Assigning a zero value to a data unit is sometimes called clearing. |
| 2. | SET IND1,IND2,IND3 TO 15 $ | Assigns the value 15 to all three named variables. |
| 3. | SET IND1 TO IND3 $ | Assigns the value of IND3 to IND1. IND3 remains unchanged. |

---

## WORK AREA

1. Write a statement to assign the value zero to a variable called INDEX.

2. Write a statement to assign the value 12 to two variables called INDEX1 and INDEX2.

3. Given the following variable declarations:

```
VRBL REC I 16 S $
VRBL DEL A 8 U 5 $
VRBL (DEL1,DEL2) A 16 S 8 $
VRBL (HOLD,HOLD1,HOLD2) I 32 S $
```

Write a statement to assign the value in:

a. HOLD2 to HOLD
b. REC to HOLD1
c. DEL to REC
d. DEL2 to DEL1

CORRECT ANSWERS

1. SET INDEX TO 0 $

2. SET INDEX1,INDEX2 TO 12 $

3. a. SET HOLD TO HOLD2 $
   b. SET HOLD1 TO REC $
   c. SET REC TO DEL $ — The fractional bits in DEL will be lost, value will be converted from fixed-point to integer.
   d. SET DEL1 TO DEL2 $

---

## Additional Examples

The following examples will use the variable declarations below:

```
VRBL COUNT I 8 U $
VRBL (IND1,IND2,IND3) I 32 S $
VRBL NUM I 8 S P -10 $
VRBL PAY A 19 U 7 $
VRBL TIM A 16 S 8 $
VRBL TIM1 A 15 U 9 $
VRBL FPAY F $
```

| | Statement | Remarks |
|---|---|---|
| 4. | SET PAY TO TIM1 $ | The value of TIM1 is assigned to PAY truncating the excess fractional bits of TIM1. |
| 5. | SET COUNT TO NUM $<br>SET NUM TO COUNT $ | COUNT and NUM have the same total number of bits, but COUNT is unsigned. Assigning the value of one to the other may have undesirable results. |

For example, a negative value of NUM (-127 say) if assigned to COUNT will assume an apparent positive value of different magnitude (in this case +255). On the other hand, a value of COUNT greater than 127 (say 137) if assigned to NUM will have the appearance of a negative number (in this case -9). Please refer to Appendix D, Section 2 if this example causes any confusion.

| | Statement | Remarks |
|---|---|---|
| 6. | SET COUNT TO IND2 $ | Assigns the value of IND2 to COUNT truncating the extra high-order (most signigicant) bits of IND2. If the value of IND2 were negative or if it exceeded the maximum positive range of COUNT (255), it would not be correctly represented. |

The examples so far have involved only variables of the same numeric type. When data units in an assignment are of different numeric types, the value of the source is converted to the type of the receiving area. Loss of accuracy may occur.

| | | |
|---|---|---|
| 7. | SET IND1 TO PAY $ | The value of the fixed-point variable PAY is converted to integer and assigned to IND1. All fractional bis are lost. PAY is unchanged. |
| 8. | SET FPAY TO PAY $ | The value of the fixed-point variable PAY is converted to floating-point and assigned to FPAY. |

---

WORK AREA

1. What are two types of actions that may cause undesirable results?

2. Using the variable declarations on the previous page, would it be advisable to assign the value of:

   a. PAY to COUNT?
   b. IND1 to IND3?
   c. TIM1 to TIM?
   d. TIM to FPAY?
   e. TIM1 to PAY?

CORRECT ANSWERS

1. (1) setting signed to unsigned (or unsigned to signed) data units
   (2) setting data units of different numeric types

2. a. not advisable - both unsigned and no space problems, but different numeric types
   b. OK - no problem
   c. no - one signed, one unsigned
   d. no - different numeric types
   e. no - TIM1 has 2 additional fractional bits which would be lost

---

## 2. ARITHMETIC COMPUTATIONS USING SET

The SET statement is also used to perform arithmetic computations. Mixed-mode arithmetic (i.e. different numeric types) is permitted. Evaluation of arithmetic expressions is hierarchical (i.e. has defined rank) according to the following table:

| Operator | Operation | Rank (hierarchy) |
|---|---|---|
| + | Unary plus (positive sign) | 1 (high) |
| - | Unary minus (minus sign) | 1 |
| ** | Exponentiation | 1 |
| * | Multiplication | 2 |
| / | Division | 2 |
| + | Addition | 3 (low) |
| - | Subtraction | 3 |

Expression evaluation proceeds from left-to-right until complete or until an operator or parentheses requiring different handling is encountered. An operation will be performed when it is of the same or higher rank than the next operation. Expressions in parentheses are evaluated before the operations (of which they are operands) are performed.

## EXAMPLES

| Expression | Steps in Evaluation |
|---|---|
| 1. A+B-C/D**2 | (1) The first operator (+) and the second (-) have equal rank. The addition A+B is performed. |
| Follow this through with simple numbers: | (2) The second operator (-) has lower rank than the third (/), so the division must be performed before the subtraction. |
| A+B-C/D**2<br>6+4-2/2**2<br>The result will be 9.5 | (3) Before dividing, the next operator ** is examined. It ranks higher than (/) and must be performed first. There are no further operators to consider, so the exponentiation is performed.<br>(4) C is divided by the result of Step 3.<br>(5) The result of Step 4 is subtracted from the result of Step 1. |
| 2. A+(B-C)/D**2 | Note that the parentheses change the evaluation.<br>(1) B-C is performed<br>(2) D is squared |
| Using the same numbers as in problem 1, the result will be 6.5 | (3) The result of Step 1 is divided by the result of Step 2.<br>(4) The result of Step 3 is added to A. |

---

## WORK AREA

1. Show the necessary steps in evaluating the following expressions:

   a. A+B*C-D

   b. (A+B)*(C-D)

   c. 2*(A+B)*C

CORRECT ANSWERS

1. a. (1) the multiplication is accomplished first
(2) + and - are equal, so A is added to the result of Step 1
(3) D is subtracted from the result of Step 2

b. (1) A is added to B
(2) D is subtracted from C
(3) the result of Step 1 is multiplied by the result of Step 2

c. (1) A is added to B
(2) the leftmost multiply is performed (2 times the result of Step 1)
(3) the result of Step 2 is multiplied by C

---

If the student works each problem with small numbers, it will help him to clarify the method of evaluation being used.

Expressions in parentheses may contain further expressions in parentheses. However, if the overall expression becomes very complicated, it is best to break it into smaller pieces.

EXAMPLES

| | Expression | Steps in Evaluation |
|---|---|---|
| 1. | A+((B-C)/D)**2 | (1) B-C is peformed<br>(2) The result of Step 1 is divided by D<br>(3) The result of Step 2 is squared<br>(4) The result of Step 3 is added to A |

The following examples will use the SET statement and will draw on the variable declarations shown directly below.

```
VRBL COUNT I 8 U $
VRBL (IND1,IND2,IND3) I 32 S $
VRBL NUM I 8 S P -1Ø $
VRBL PAY A 19 U 7 $
VRBL RATE A 12 U 7 P 3.75 $
VRBL DIF A 16 S 3 $
VRBL FPAY F $
VRBL FRATE F P 3.75E-4 $
```

Statement — Steps in Evaluation

2. SET IND1 TO NUM*COUNT +3 $

   The values of NUM and COUNT are multiplied, the constant 3 is added to the result and the final value is assigned to IND1.

3. SET IND2 TO (COUNT-NUM**2)/3+IND1*4 $

   The expression in parentheses is evaluated first-the value of NUM is squared and the result subtracted from the value of COUNT. This value is divided by 3 and the result is held. The value of IND1 is multiplied by 4 and added to the value being held. This final value is then assigned to IND2.

4. SET IND2 TO COUNT-NUM**2/3+IND1*4 $

   The only difference between this and the previous statement is in the absence of parentheses, which changes the evaluation. The expression NUM**2/3 will be evaluated and subtracted from the value of COUNT. The remaining evaluation is the same.

---

WORK AREA

1. What is the lowest rank operation in the evaluation of expressions?

2. Given A+B*C What would be evaluated first? Why?

3. Given A+(B*C) From the point of view of evaluation, what is the difference between this expression and the one in problem 2?

4. Show the necessary steps in evaluating the following statements:
   a. SET IND1 TO IND2-3+2*NUM $
   b. SET IND TO IND2-(3+2*NUM) $

CORRECT ANSWERS

1. addition and subtraction

2. B*C - the + is lower rank than the *

3. none - whatever is enclosed in parentheses is always evaluated first

4. a. (1) Subtract 3 from the value of IND2
   (2) Multiply the value of NUM by 2
   (3) Add the result of Step 2 to the result of Step 1

   b. (1) Multiply the value of NUM by 2
   (2) Add 3 to the result of Step 1
   (3) Subtract the result of Step 2 from the value of IND2.

---

## 3. SCALING

When fractional quantities are involved, the scaling (number of fractional bits) of a result is determined by the scaling of the receiving area (receptacle). Scaling of intermediate results may differ. In general, intermediate scaling is designed to preserve the greatest possible accuracy.

Detailed discussion of the CMS-2Y scaling algorithm and programmer control of scaling will not be covered in this manual. These brief examples are shown, so that the student will be aware of the problems that may occur with fractional numbers.

1. The variable declarations on page 30 are still being used for these examples. 2. The final scaling is always determined by the scaling of the receptacle (in this case IND3). The scaling of intermediate values is not always predictable in advance. Caution must be exercised when mixing data of different types or scaling.

EXAMPLES

| Statement | Steps in Evaluation |
|---|---|
| 1. SET IND3 TO RATE+DIF $ | The value of RATE (7 fractional bits) will be aligned with the value of DIF (3 fractional bits) the addition will be performed and the result (3 fractional bits) will be aligned with and assigned to IND3 (no fractional bits). |

| Statement | Steps in Evaluation |
|---|---|
| 2. SET PAY TO DIF*RATE $ | In multiplication, the scaling of the result equals the sum of the scaling of the factors. The value of DIF (3 fractional bits) times the value of RATE (7 fractional bits) gives a result having 10 fractional bits. This is aligned with and assigned to PAY (7 fractional bits) |
| 3. SET DIF TO PAY/RATE $ | In division, the scaling of the result equals the difference of the scaling of the dividend and the scaling of the divisor. The values of the dividend PAY and the divisor RATE will be adjusted so that when the division is performed the result will be the same scaling as DIF and may be assigned without manipulation. |

Note: If any data unit in an expression is of floating-point type, the values of all operands will be converted to their closest floating-point approximations and the operations will be carried out in floating-point. If necessary, the result will be converted to the type of the receptacle.

| | |
|---|---|
| 4. SET FPAY TO FRATE*40 $ | The multiplication will be done in floating-point and the result assigned to FPAY. |
| 5. SET IND3 TO FPAY+PAY $ | The fixed-point value of PAY will be converted to floating-point and added to the value of FPAY. The result will be converted to integer type and assigned to IND3. |

---

WORK AREA

1. When scaling is involved, what determines the number of fractional bits in the result?

For the following questions, use the variable declarations on page 30.

2. Write a statement to multiply the values of COUNT and NUM and to place the result in location IND1.
3. Write a statement to subtract the product of PAY and RATE from DIF, placing the result in IND2.
4. In problem 3 above, what would happen to the fractional bits at the completion of the arithmetic?

## CORRECT ANSWERS

1. the number of fractional bits defined in the receiving area (receptacle)

2. SET IND1 TO COUNT*NUM $

3. SET IND2 TO DIF-PAY*RATE $
   this could also be written: SET IND2 TO DIF-(PAY*RATE) $

4. all fractional bits would be lost

---

## Chapter 2 Summary

This chapter introduces the numeric variables (integer, fixed-point and floating-point). Each of the three types is shown with many examples, definition of rules and problems. The SET statement is then introduced so that actual use of variables can be shown, followed by arithmetic computations using the SET statement. Finally, scaling is touched on briefly.

The variable declaration (VRBL) is used to specify to the compiler the type and structure of the variable to be used and the name (identifier) associated with it. Arithmetic calculations, the transfer of data and the assignment of values to specific data units are accomplished with the SET statement.

## Chapter 2 Quiz

Work these problems on scratch paper, then check your answers with the correct answers given on the following page.

1. In a variable declaration:
   a. to what do the letters I, A and F refer?
   b. to what do the letters P, S and U refer?
   c. What three things may be specified under the heading of "type"?

2. Write a variable declaration containing the following information:
   16 bit, fixed-point, no fractional bits, signed, identified as HOLD1

3. In problem 2 above, how many bits are available to hold the value? Why?

4. Explain the meaning of the following declaration:

```
VRBL (COUNT1,COUNT2) I 32 U $
```

5. When is it necessary to use fixed-point variables?

6. The keyword SET is used for what two types of actions?

7. Explain the meaning of each of the following statements:

```
a.  SET NUM1,NUM2 TO 16 $
b.  SET IND TO NUM1 $
```

8. Of the following arithmetic operators, which one ranks the highest?

```
- * + / **
```

9. Given variables with the following attributes:
   a. 16-bit, signed, fixed-point, with no fractional bits.
   b. The receptacle for each statement is a 32 bit, signed, fixed-point variable, with no fractional bits, called RESULT.

   Write the necessary VRBL and SET statements to define data units and to evaluate the following expressions:

   Expressions:

   1. X**2-2XY-Y**2
   2. (AX+BY)*(X**2-Y)
   3. $\frac{(X/Y+4)}{X+1}$

   In writing the variable statements, use the data names given in the above expressions (A, B, X, Y) so that the answers can be more easily checked with the correct answers given on the following page.

## CORRECT ANSWERS

1. a. I - integer, A - fixed-point, F - floating-point
   b. P - preset value follows, S - signed, U - unsigned
   c. (1) kind of value, (2) size of the variable, (3) number of fractional bits

2. VRBL HOLD1 A 16 S 0 $

3. 15 bits because 1 bit (the leftmost) is required for the sign

4. a variable declaration containing two variables COUNT1 and COUNT2, integer, 32 bits long, unsigned

5. when fractional values are to be used

6. arithmetic calculations and data transfer

7. a. the value of 16 is assigned to both variables NUM1 and NUM2
   b. assign the value of NUM1 to IND

8. ** exponentiation

9. VRBL (A,B,X,Y) A 16 S 0 $
   VRBL RESULT A 32 S 0 $

   1. SET RESULT TO X**2-2*X*Y-Y**2 $
   2. SET RESULT TO (A*X+B*Y)*(X**2-Y) $
   3. SET RESULT TO (X/Y+4)/(X+1) $

# Chapter 3

# NON-NUMERIC VARIABLES AND DECISION MAKING

## A. NON-NUMERIC VARIABLES

The three non-numeric variable types are Boolean (B), character (H) and status (S). (Status variables are implemented for the AN/UYK-7 computer only.)

### 1. BOOLEAN VARIABLES

Some data units in computer programs are used only to represent one of two values: on/off, high/low, true/false, yes/no. In these cases, only a single bit is required for the information. In CMS-2, the Boolean data unit can be used to represent this binary condition. The two possible values are normally called true and false. A true condition is represented with a binary 1 and a false condition is represented with a binary 0. A Boolean variable declaration has the following format:

FORMAT

| | | | | |
|---|---|---|---|---|
| VRBL | name identifier(s) | type always B | P (optional) preset value to follow | preset value (optional) value(s) to which variables are preset $ |

Since only one bit is required for a Boolean data unit to indicate 1 or 0, the number of bits is not specified.

EXAMPLES

| | Declaration | Remarks |
|---|---|---|
| 1. | VRBL (FLG1,FLG2,FLG3) B $ | FLG1,FLG2 and FLG3 are all Boolean variables |
| 2. | VRBL HOLD B $ | HOLD is a Boolean variable |
| 3. | VRBL HOLD1 B P 1 $ | HOLD1 is a Boolean variable preset to 1 |

A Boolean data unit may be the receptacle for the result of any one of three Boolean operations. The operators and their hierarchy (for execution) are shown below:

| Operator | Operation | Hierarchy |
|---|---|---|
| COMP | Logical Complement | 1 |
| AND | Logical Product | 2 |
| OR | Logical Sum | 3 |

The Boolean operators are used to work problems requiring the rules of Boolean algebra. A short discussion of Boolean algebra may be found in Appendix D, Section 3, for the benefit of those who wish to review this area. The following examples show some uses of the Boolean variable and the logical operations related to it.

EXAMPLES

| Statement | Remarks |
|---|---|
| 1. Single Boolean variable | |
| a. SET FLG1 TO 0 $ | FLG1 is assigned the value zero (false). |
| b. SET FLG2 TO FLG3 $ | FLG2 is assigned the value of FLG3. |
| 2. Logical complement | |
| SET FLG3 TO COMP FLG1 $ | The value of FLG1 is changed to its logical complement (0 to 1, 1 to 0) and the result is assigned to FLG3. |

---

WORK AREA

1. SET FLG1 TO COMP FLG3 $ FLG3 has a value of 1. After this statement is executed, what will be the value assigned to FLG1?
2. SET FLG3 TO FLG1 AND FLG2 $ What will be assigned to FLG3 under the following conditions:
   a. FLG1=0,FLG2=1
   b. FLG1=1,FLG2=0
   c. FLG1=1,FLG2=1
   d. FLG1=0,FLG2=0
3. SET FLG3 TO FLG1 OR FLG2 $ What will be assigned to FLG3 under the conditions shown in problem 2?
4. SET FLG1 TO (FLG2 OR FLG3) AND (FLG4 AND FLG5) $ What will be assigned to FLG1, assuming the following values:
   a. FLG2=0,FLG3=0,FLG4=1,FLG5=1
   b. FLG2=1,FLG3=0,FLG4=1,FLG5=1

CORRECT ANSWERS

1. zero
2. a. zero
   b. zero
   c. one
   d. zero
3. a. one
   b. one
   c. one
   d. zero
4. a. zero
   b. one

---

EXAMPLES continued

3. Logical product

```
SET FLG1 TO FLG2 AND FLG3 $
```

The source expression (FLG2 AND FLG3) is evaluated and the result is assigned to FLG1. If FLG2 equals 1 and FLG3 equals 1, the result is a value of 1. Any other combination gives a value of zero.

4. Logical sum

```
SET FLG2 TO FLG1 OR FLG3 $
```

The expression (FLG1 OR FLG3) is evaluated and the result is assigned to FLG2. If both FLG1 and FLG3 equal zero, the result is a value of zero. Any other combination gives a value of one.

5. Combination

```
SET FLG1 TO FLG2 OR (FLG1 AND COMP FLG3) $
```

The source expression is evaluated (beginning with the part in parentheses) and the result is assigned to FLG1.
Assume FLG1 equals 0, FLG3 equals 1. Comp FLG3 would then equal 0; 0 and 0 equals 0. When FLG2 equals 0, FLG1 will be set to 0; when FLG2 equals 1, FLG1 will be set to 1.

A Boolean type data unit cannot be set to the value of any other type data unit, nor to a constant other than 0 or 1. It also cannot be the source in a SET statement whose receptacle is not Boolean.

Examples of Illegal Statements

1. `SET FLG1 TO 5 $` Only zero or one is allowed
2. `SET COUNT TO FLG1 $` Where COUNT has been defined as integer type, the statement is illegal.

## 2. CHARACTER VARIABLES

A character type data unit provides a means of assigning a name to a series of characters (often called a "string" of characters). Character strings can be used to perform a variety of non-numeric actions such as setting up messages and checking for keywords. A character variable declaration is as follows:

FORMAT

| VRBL | name identifier(s) | type always H | number of characters | P (optional) preset value | preset value (optional) value to which variables are preset | $ |
|---|---|---|---|---|---|---|

EXAMPLE

| Declaration | Remarks |
|---|---|
| VRBL LINE H 132 $ | LINE is a character variable, 132 characters in length |

A character data unit may contain a maximum of 132 characters.

Character type data units may be assigned values called character constants. A character constant is specified by the letter H immediately followed by the desired string of characters enclosed by parentheses, as follows:

H(character string)

The value of the character constant is the string that is enclosed in parentheses.

A character constant may contain any CMS-2Y characters or special characters, however, a right parenthesis must be represented by two consecutive right parentheses, otherwise it would be taken for the end-of-string indicator.

Examples of Character Constants:
H(A1/QY7)
H($7.25)
H(FIL(CMS)),GO)

---

WORK AREA

1. What must be the type designation for a character variable declaration?

2. The values assigned to character variables are called...

CORRECT ANSWERS

1. H

2. character constants

---

A character type data unit may be a receptacle in a SET statement. The source expression must be of compatible type. The method of handling is as follows:

1. If the source character string is the same length as the receptacle, its value replaces the value that was in the receptacle.
2. If the source is longer than the receptacle, the leftmost characters will be transferred to the receptacle, while the rightmost excess characters of the source will be truncated and not transferred.
3. If the source is shorter than the receptacle, there are two possible cases:
   a. If the source is a character data unit, it will enter the receptacle left-justified leaving the excess rightmost characters in the receptacle untouched.
   b. If the source is a character constant, it will enter the receptacle left-justified with the excess rightmost characters in the receptacle being filled with spaces.

A preset character constant value in a variable declaration is handled exactly like a character constant in a SET statement.

EXAMPLES

| Receptacle | Source | Remarks |
|---|---|---|
| 12 characters | 12 characters | Source and receptacle the same size value moves into receptacle. |
| 10 characters | 12 characters | Source larger, rightmost 2 characters of source will not be transferred. |
| 14 characters | 12 characters | Source smaller<br>1. if source is data unit, it enters the receptacle left-justified, leaving rightmost 2 characters in receptacle untouched<br>2. if source is character constant, it enters left-justified with rightmost 2 characters in receptacle filled with spaces |

## EXAMPLES

The additional examples below are based on the following variable declarations:

```
VRBL LINE H 132 $
VRBL HEADR H 15 P H(NUMBER OF ITEMS) $
VRBL WORD H 4 $
VRBL DATE H 9 P H(19NOV1980) $
```

| Statement | Remarks |
|---|---|
| 1. SET WORD TO H(TIME) $ | The length of the character constant and the length of the receptacle are the same. WORD is assigned the string TIME. |
| 2. SET WORD TO DATE $ | The receptacle WORD is shorter than the source DATE. The value of DATE will be truncated to 19NO (4 characters) and assigned to WORD. |
| 3. SET LINE TO HEADR $ | The receptacle line is longer than the source HEADR. Since HEADR is a data unit, its string NUMBER OF ITEMS will be assigned to the 15 leftmost locations of LINE and the remaining locations will remain unchanged. |
| 4. SET LINE TO H(NUMBER OF ITEMS) $ | Since the source this time is a character constant, the string NUMBER OF ITEMS is assigned to the 15 leftmost character locations of LINE and the remaining locations will be filled with spaces. |

EXAMPLES continued

5. SET LINE TO H( SUMMARY) $

.... (20 spaces) ....

If it is desired to assign the constant SUMMARY starting with the twenty-first position of the 132 positions in LINE, the only way to do that would be to hit the space bar 20 times following the left parenthesis. The remainder of the positions beyond SUMMARY would automatically be set to spaces.

6. SET LINE TO H( ) $

The space character in parentheses will be extended on the right, so that LINE becomes all spaces.

7. SET HEADR TO H(SUBSET(0))) $

Note that to get a right parenthesis included in the character constant, it is necessary to use two consecutive right parentheses. The third one is the closing delimiter for the character constant.

The result is to set HEADR to the string SUBSET(0) followed by 6 space characters.

WORK AREA

1. Write a variable declaration for HDR1 with 14 characters preset to the character constant SUMMARY REVIEW.

2. What will happen in the following instances:
   a. VRBL HDR2 H 16 P H(HOLDING AREA) $
   b. VRBL HDR3 H 16 P H(MAINTAINED AT 247) $
   c. VRBL HDR4 H 16 P H(UNLOAD AREA 24.5) $

3. Using the following variable declarations, write SET statements and explain what would happen in each case.

```
VRBL HOLD H 5 $
VRBL CORD H 14 P H(CONSTANT VALUE) $
VRBL DATE H 9 P H(23JAN81) $
VRBL HOUR H 5 P H(02:46) $
```

   a. set the value TIME1 into location HOUR.
   b. set the value of variable DATE into location HOLD.
   c. set the value VARIABLE DATA UNIT into location CORD.

4. Using the variables in problem 3 above, write a SET statement to put spaces into CORD.

5. What is wrong with the following statements:
   a. SET HOUR TO H (SETUP) $
   b. VRBL CONTR1 H 16 H(DATA) $

CORRECT ANSWERS

1. VRBL HDR1 H 14 P H(SUMMARY REVIEW) $
2. a. source smaller than receptacle_rightmost 4 characters in receptacle set to spaces.
   b. source larger_rightmost character (the 7) will be truncated.
   c. same size_no problem.
3. a. SET HOUR TO H(TIME1) $ the string TIME1 would move into location HOUR.
   b. SET HOLD TO DATE $ the year (81) would be truncated.
   c. SET CORD TO H(VARIABLE DATA UNIT) $ the string UNIT would be truncated.
4. SET CORD TO H( ) $
5. a. the left parenthesis must immediately follow the H
   b. the P was omitted just in front of the character constant H(DATA)

---

## 3. STATUS VARIABLES

Some computer programs require the usage of different conditions or values. Some examples are:

| | | |
|---|---|---|
| short | Army | UYK-7 |
| medium | Navy | UYK-20 |
| long | Air Force | CP-642 |
| extra long | Marines | UYK-43 |

These values can be programmed using numeric data units and assigning values 1, 2, 3, 4, etc. arbitrarily to each condition. However, the numeric codes have no real meaning and must always be cross-checked to the real values by a programmer. In CMS-2, these data values can be more accurately represented using status type data units.

A status type data unit provides a method of representing programmer specified conditions. The number of bits is not specified in the declaration but is automatically handled by the compiler. A status variable declaration has the following basic format:

FORMAT

| VRBL | name identifier(s) | type always S | list of values | P (optional) | preset value (optional) | $ |
|---|---|---|---|---|---|---|

The values in the list are enclosed in single apostrophes and separated by commas.

EXAMPLE

```
VRBL COMPAR S 'EQ', 'GR', 'LESS' $
```

The values enclosed in single apostrophes are called status constants. The value of each status constant is the string enclosed between the two apostrophes.

EXAMPLES

| Declaration | Remarks |
|---|---|
| VRBL CHECK S 'LOW','MEDIUM','HIGH' $ | CHECK is a status variable. It can assume the values 'LOW', 'MEDIUM', and 'HIGH' and no others. |
| VRBL SIZE S 'SHORT','MEDIUM','LONG','XLONG' $ | Size is a status variable that can assume the above specified values and no others. |

Rules:

1. A status constant may contain no more than 8 characters
2. An apostrophe is not permitted in a status constant
3. The same status constant may appear only once in a given data declaration, but may appear in any number of different declarations. In the examples above, 'MEDIUM' appears in both CHECK and SIZE but has different meanings.
4. A status type data unit may have a maximum of 12 values.
5. Used as the receptacle in a SET statement, a status type data unit accepts only status type source values.
6. Status constants must have been defined in data declarations:

EXAMPLES

```
VRBL CHECK S 'LOW','MEDIUM','HIGH' $
VRBL SIZE S 'SHORT','MEDIUM','LONG','XLONG' $
```

| Statement | Remarks |
|---|---|
| SET CHECK TO 'LOW' $ | The compiler supplied representation for 'LOW' is assigned to CHECK. |
| SET CHECK TO 'BROWN' $ | Illegal statement. 'BROWN' was not declared as a possible value of CHECK. |

The compiler automatically assigns a numeric representation to each of the status constants, moving from left to right and beginning with zero. These numeric representations are only used by the compiler. The actual status constants must be used by the programmer.

---

WORK AREA

1. What is wrong with the following status constants?
   a. 'CONTEMPORARY'
   b. 'CHECK'
2. What is the maximum number of status constants allowed in one data declaration?
3. Using the variable statements above, what is wrong with the following?
   a. SET SIZE TO 2 $
   b. SET CHECK TO '2' $
   c. SET SIZE TO 'LONG' $
   d. SET CHECK TO 'LOWER' $
4. What is wrong with the following declaration?

```
VRBL SIM S 'HI','LO','EXHI','LO','EXLO' $
```

CORRECT ANSWERS

1. a. more than 8 characters in length
   b. nothing wrong
2. twelve
3. a. a number is not allowed - must use a status constant
   b. '2' was not defined as a status constant for this variable
   c. this is OK
   d. 'LOWER' was not defined as a status constant for this variable
4. 'LO' used twice in the same data declaration

---

## B. DECISION MAKING

The execution of previously discussed SET statements would normally flow sequentially from one statement to the next. Often, logic requires the interruption of such smooth execution flow so that control can be given to a totally different set of instructions. In this chapter, the conditional statement and the unconditional branch statement will be discussed.

### 1. UNCONDITIONAL BRANCH STATEMENT

An unconditional branch statement consists of the keyword GOTO and the name of the statement to which program control is to be transferred.

| | EXAMPLE | |
|---|---|---|
| | Statement | Remarks |
| | GOTO AIF $ | Transfer control to the statement labeled AIF. Note that the period |
| various | program statements | used to define a label is omitted |
| AIF. | SET IND1 TO 0 $ | when the statement is referenced. |

It makes no difference whether the specified statement is located before or after the GOTO statement in the program. However, the statement must be a part of the same sub-program as the GOTO.

### 2. RELATIONAL OPERATORS AND EXPRESSIONS

A relational expression is a combination of data units and relational operators to form a condition that can be reduced to a value of true or false. Program decisions are made by evaluating specified relational expressions.

The six relational operators and their meanings in numeric and Boolean relational expressions are shown on the following page.

| Operator | Meaning |
|---|---|
| EQ | is equal to |
| NOT | is not equal to |
| LT | is less than |
| LTEQ | is less than or equal to |
| GT | is greater than |
| GTEQ | is greater than or equal to |

There is no hierarchy among relational operators. Each relational expression is evaluated as it is reached in left-to-right order. The comparison of a Boolean relational expression compares the values of 0 and 1, therefore the relational operators have the same meaning for Boolean as for numeric expressions.

EXAMPLES

```
VRBL COUNT I 8 U $
VRBL (FLG1,FLG2) B $
VRBL (IND1,IND2,IND3) I 32 S $
```

| Partial Statement | Remarks |
|---|---|
| .... COUNT LT 45 ... | The numeric relational expression is true when the value of COUNT is less than 45. |
| .... FLG1 GT FLG2 ... | The Boolean relational expression is true when FLG1=1 and FLG2=0. |
| .... IND1+2 LTEQ IND2-IND3 ... | Either or both operands may be expressions. The comparison is true when the value of the left-hand expression (IND1+2) is less than or equal to the value of the right-hand expression (IND2-IND3). |

---

WORK AREA

1. Write an unconditional branch statement that will send the program to the statement labeled JUMP1.

2. Are the following expressions true or false?

   a. . . .FLG1 LT FLG2. .(1) FLG1=0 FLG2=0
   (2) FLG1=1 FLG2=0
   (3) FLG1=0 FLG2=1

   b. . . .VAR1 GT 12. . (1) VAR1=4
   (2) VAR1=22
   (3) VAR1=12

   c.... VAR1 NOT VAR2+VAR3 .(1) VAR1=4 VAR2=3 VAR3=1
   (2) VAR1=6 VAR2=2 VAR3=5

CORRECT ANSWERS

1. GOTO JUMP1 $
2. a. (1) false (2) false (3) true
   b. (1) false (2) true (3) false
   c. (1) false (2) true

---

## 3. CONDITIONAL STATEMENTS

The evaluation of a conditional statement controls the sequence of program execution. If the value of the condition is true, a statement is given to be executed. An optional statement may be given to cover the false condition. This sequence is often referred to as IF-THEN-ELSE logic.

FORMAT

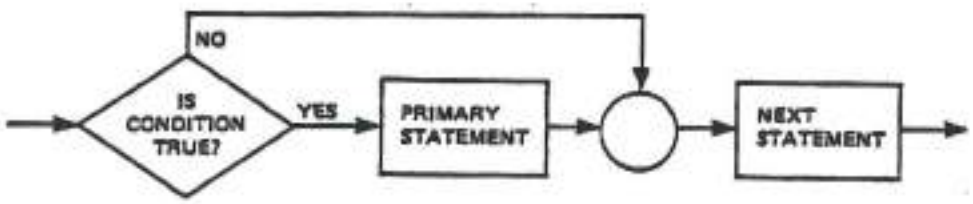
```
IF     condition    THEN      primary statement $

           ELSE          alternative statement $
```

Explanation:

| | |
|---|---|
| IF | Keyword indicating a conditional statement |
| condition | An expression whose value determines the execution of the following statement or the alternative statement. |
| THEN | Keyword indicating that the primary statement follows |
| primary statement | A statement to be executed if the condition is true |
| ELSE | (Optional) keyword indicating that an optional statement follows |
| alternative statement | (optional) a statement to be executed if the condition is false |

The conditional statement could be illustrated as follows:

(a) A conditional statement without an ELSE clause could be flowcharted as shown below:

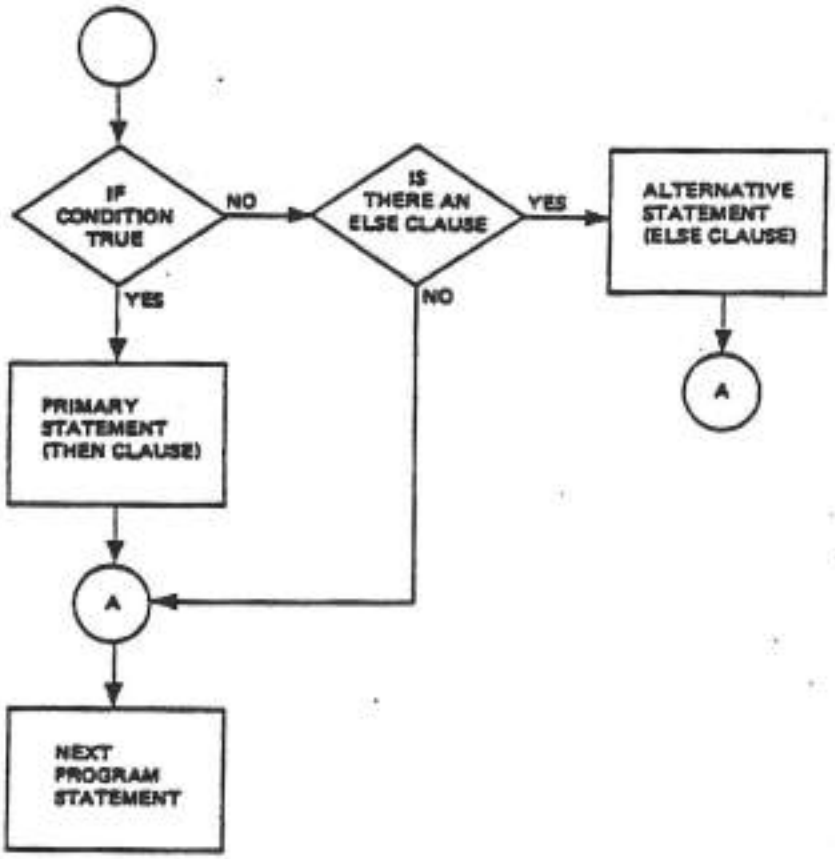



```
IF condition THEN primary statement $
next statement $
```

(b) A conditional statement with an ELSE clause is flowcharted as follows:

```
IF condition THEN primary statement $
     ELSE alternative statement $
next statement $
```

Note that both the IF - THEN and the ELSE statements are completed with the dollar sign as separate statements, although the logic of IF - THEN - ELSE works together.



## EXAMPLES

```
VRBL COUNT I 8 U $

VRBL NUM I 8 S P -10 $

VRBL (IND1,IND2,IND3) I 32 S $

VRBL TIM A 16 S 8 $

VRBL TIM1 A 15 U 9 $

VRBL DIF A 15 S 3 $

VRBL (FRATE,FPAY) F $

VRBL (FLG1,FLG2,FLG3) B $
```

Examples with no ELSE clause:

```
          IF COUNT LTEQ 10 THEN SET IND1 TO COUNT $
NXT.      SET IND2 TO IND3+2 $
```

When the condition is satisfied (i.e. COUNT is less than or equal to 10), the statement after THEN is executed. Since there is no ELSE clause, execution continues with statement NXT.

When the condition is not satisfied, the THEN clause is skipped. Since there is no ELSE clause, execution continues immediately with statement NXT.

```
          IF NUM GT 5 THEN SET IND2 TO IND3 $
NXT.      SET IND1 TO NUM $
```

When the condition is satisfied (COUNT is greater than 5), the statement after THEN is executed. Execution continues with the next statement in sequence. If the condition is not satisfied (COUNT is less than or equal to 5), execution goes immediately to the next statement.

Example with ELSE clause:

```
                IF IND2 EQ IND3
                   THEN SET COUNT TO IND2-1 $
                      ELSE SET COUNT TO 0 $
          AIF.  SET IND1 TO 3 $
```

Remarks: When the condition is satisfied (IND2=IND3), the statement after THEN is executed, the ELSE clause is skipped and execution continues with statement AIF.

When the condition is not satisfied, the THEN clause is skipped, the statement after ELSE is executed, followed by statement AIF.

---

WORK AREA

1. In the variable declarations on page 52:
   a. identify the types
   b. how many are unsigned?
   c. what is the largest bit size?

2. The primary statement is executed only if the condition turns out to be _____.

3. Considering the examples above, which statement would be executed under the following conditions?
   a. COUNT=10
   b. COUNT= 8
   c. COUNT=14
   d. IND2=4 IND3=5
   e. IND2=3 IND3=2
   f. IND2=6 IND3=6

CORRECT ANSWERS

1. a. 3 integer, 3 fixed-point, 1 floating-point and 1 Boolean
   b. two
   c. 32 (programmer controlled)
2. true
3. a. SET IND1 TO COUNT
   b. SET IND1 TO COUNT
   c. NXT. SET IND2 TO IND3+2
   d. SET COUNT TO 0
   e. SET COUNT TO 0
   f. SET COUNT TO IND2-1

---

The values being compared (comparands) in the examples on the previous pages were all of integer type. Other combinations are possible, including comparands of different numeric types.

When fixed-point to fixed-point or fixed-point to integer comparisons are made, the values are aligned before being compared. This may result in loss of accuracy from one or more of the operands, possibly changing the meaning of the comparison, therefore it must be considered very carefully.

Doing mixed-mode comparisons is not recommended for the beginner. It is mentioned here only to show that it is possible.

Comparisons of Boolean values use the numeric representations of true (1) and false (0). The relational operators have the same meanings for Boolean comparisons as for numeric.

EXAMPLE

| Statement | Remarks |
|---|---|
| IF FLG1 GT FLG2 THEN GOTO NXT $ | The condition is satisfied when FLG1=1 and FLG2=0. |

A statement to check for a true condition may be written in full or in a short form which is also recognized by the compiler.

EXAMPLE

```
long form:      IF FLG3 EQ 1 THEN GOTO NXT $

short form:     IF FLG3 THEN GOTO NXT $
```

Remarks: The compiler recognizes both statements as a check for FLG3=1 (true).

Any valid relational expression may be an operand in a Boolean expression.

EXAMPLE

```
long form:        IF FLG2 EQ 1 AND FLG3 EQ 1 THEN GOTO NXT $
short form:       IF FLG2 AND FLG3 THEN GOTO NXT $
```

Remarks: FLG2 is evaluated first. When FLG2=0 the condition cannot be satisfied and execution continues with the next statement. When FLG2=1, FLG3 is evaluated. If FLG3=1 also, the condition is satisfied and the GOTO statement is executed. If FLG3=0 the condition is not satisfied and execution continues with the next statement.

EXAMPLE

```
IF FLG3 OR (IND1**2 EQ IND2/3) THEN GOTO NXT $
```

Remarks: FLG3 is evaluated first. When FLG3=1, the condition is satisfied whatever the value of the right operand and the GOTO statement is executed.
When FLG3=0, the right operand is evaluated. If the two numeric expressions are equal in value, the condition is satisfied and the GOTO statement is executed. If they are not equal, execution continues with the following statement.

---

WORK AREA

1. IF FLG1 GTEQ FLG2 THEN GOTO NXT $
   SET FLG3 TO 0 $
   What will be executed given the following?
   a. FLG1=1 FLG2=0
   b. FLG1=0 FLG2=0
   c. FLG1=1 FLG2=1
   d. FLG1=0 FLG2=1
2. Which bit value (0 and 1) indicates a "true" statement?
3. The following statement is a short form of what full statement?
   IF FLG1 THEN GOTO NXT $
4. IF FLG1 AND FLG2 THEN GOTO NXT $
   SET FLG3 TO 1 $
   What will be executed given the following?
   a. FLG1=0 FLG2=1
   b. FLG1=1 FLG2=0
   c. FLG1=1 FLG2=1
5. IF FLG1 OR FLG2 THEN GOTO NXT $
   SET FLG3 TO 0 $
   What will be executed, given the following?
   a. FLG1=1 FLG2=1
   b. FLG1=0 FLG2=0
   c. FLG1=0 FLG2=1

CORRECT ANSWERS

1. a. the THEN clause
   b. the THEN clause
   c. the THEN clause
   d. the SET statement
2. the value 1 indicates a true statement
3. IF FLG1 EQ 1 THEN GOTO NXT $
4. a. the SET statement
   b. the SET statement
   c. the THEN clause
5. a. the THEN clause
   b. the SET statement
   c. the THEN clause

---

The relational operators described on page 49 have the same meanings for status type comparands as for numeric comparands.

The instructions generated for the comparison use the numeric values assigned to the status constants. The programmer must use the status constant names, and should think of the comparison in terms of the order in which the status constants appear in the variable declaration (i.e. an early appearing name would be "less than" a late appearing name).

Status type comparands may be compared only to other status type comparands.

EXAMPLES

```
VRBL CHECK S 'LOW','MEDIUM','HIGH' $
VRBL SIZE S 'SHORT','MEDIUM','LONG','XLONG' $
```

| | Statement | Remarks |
|---|---|---|
| 1. | IF CHECK EQ 'HIGH' THEN GOTO NXT $ | The condition is satisfied when the value of CHECK matches the value 'HIGH'. |
| 2. | IF CHECK NOT 'LOW' THEN GOTO NXT $ | The condition is satisfied when the value of CHECK is either 'MEDIUM' or 'HIGH'. |
| 3. | IF SIZE LT 'LONG' THEN GOTO NXT $ | The condition is satisfied when the value of SIZE is either 'SHORT' or 'MEDIUM'. |
| 4. | IF SIZE GT 'LONG' THEN GOTO NXT $ | The condition is satisfied only when the value of SIZE is 'XLONG'. |

When relational operators are used with character type operands, the meanings are slightly changed as shown below:

| Operator | Meaning |
|---|---|
| EQ | is the same as |
| NOT | is not the same as |
| LT | collates before |
| GT | collates after |
| LTEQ | collates before or is the same as |
| GTEQ | collates after or is the same as |

Comparisons are performed on a character-by-character basis, beginning with the left-most characters. The first inequality found by this process determines the result of the comparison. If no inequality is found, the comparands are the same.

If two comparands are of different lengths, the number of characters compared is determined by the shorter, unless the shorter is a character constant. When the shorter comparand is a character constant, it is extended on the right with space characters to the length of the longer comparand.

EXAMPLES

```
VRBL LINE H 132 $
VRBL HEADR H 15 P H(NUMBER OF ITEMS) $
VRBL WORD H 4 $
VRBL DATE H 9 P H(19OCT1980) $
VRBL NAME1 H 5 P H(WELLS) $
VRBL NAME2 H 5 P H(WELCH) $
```

| | Statements | Remarks |
|---|---|---|
| 1. | IF LINE EQ HEADR THEN GOTO NXT $ | The number of characters to be compared is determined by the shorter comparand HEADR. The first 15 characters of LINE will be compared with HEADR. If they are the same, the condition will be deemed satisfied whatever the remaining characters in LINE may be. |
| 2. | IF LINE EQ H(NUMBER OF ITEMS) THEN GOTO NXT $ | The 15 characters in the character constant are extended on the right with space characters to the 132-character length of LINE. All 132 characters must match, character-by-character, to satisfy the condition. |

3. IF NAME1 LT NAME2 THEN GOTQ NXT $

NAME1 and NAME2 are the same length and will be compared character-by-character. Using their present values for the comparison, an inequality is found in the 4th character position. Since the L in NAME2's value collates after the C in NAME2's value, the condition is not satisfied and execution continues with the next statement.

4. IF LINE GT NAME1 THEN GOTO NXT $

The 5 characters of NAME1 will be compared to the first 5 characters of LINE. If the characters of LINE are GT (i.e. come after the characters of NAME1 alphabetically) the condition is satisfied.

WORK AREA

Using the following variable statements:

```
VRBL CHECK S 'LOW','MEDIUM','HIGH' $
VRBL SIZE S 'SHORT','MEDIUM','LONG','XLONG' $
```

1. Write an IF statement to send the program to the statement labeled FETCH if:
   a. CHECK is greater than 'LOW'
   b. SIZE is equal to 'MEDIUM'
   c. SIZE is less than 'LONG'
   d. CHECK is equal to 'MEDIUM'

2. May status type comparands be compared to numeric type comparands?

3. What is the meaning of each of the following relational operators?
   a. GTEQ
   b. LTEQ
   c. NOT

4. Given the following character constants, what is the relationship of A to B?
   a. A. H(JK576AFG)
      B. H(JK573AFG)
   b. A. H(JOAN)
      B. H(JOHN)
   c. A. H(NEVER)
      B. H(NEVER)

5. Write a statement to check the contents of variable LINE for the entry BOWLING LEAGUE AVERAGES BY DIVISION, exactly, and to go to a label called FORMIT when the condition is satisfied.

CORRECT ANSWERS

1. a. IF CHECK GT 'LOW' THEN GOTO FETCH $
   b. IF SIZE EQ 'MEDIUM' THEN GOTO FETCH $
   c. IF SIZE LT 'LONG' THEN GOTO FETCH $
   d. IF CHECK EQ 'MEDIUM' THEN GOTO FETCH $
2. No - only to other status type comparands
3. a. greater than or equal to
   b. less than or equal to
   c. not equal to

4. a. A is greater than (collates after) B
   b. A is less than (collates before) B
   c. A equals (is the same as) B
5. IF LINE EQ H(BOWLING LEAGUE AVERAGES BY DIVISION)
   THEN GOTO FORMIT $

---

## 4. BEGIN BLOCKS

Navy standards prohibit the use of branching statements (GOTO) when there is a logical and efficient way to avoid it. In many instances this can be accomplished by use of BEGIN blocks.

The BEGIN block is a programming device that allows a group of statements to appear in places that normally call for a single statement. One place would be after THEN in an IF statement.

FORMAT

| | |
|---|---|
| BEGIN $ | This keyword indicates that the following group of statements are to be treated as a single unit. |
| various statements | |
| END $ | This keyword indicates the end of a block of statements. |

BEGIN blocks are not the only programming devices that require an END statement. Other types will be discussed later.

Blocks may be nested (i.e. blocks-within-blocks). The programmer must provide an END statement for each BEGIN statement. To avoid losing track, it is a good policy to add a note to each BEGIN and its matching END statement.

EXAMPLE

```
BEGIN ''1'' $          first block started     Note that block
                                               2 is nested
BEGIN ''2'' $          second block started    within block 1.
                                               they may not
END ''2'' $            second block ended      overlap.
END ''1'' $            first block ended
```

Two examples of the same IF statement are shown below, one using GOTO and the other using BEGIN blocks.

EXAMPLE 1: GOTO statement

```
IF FLG1 THEN GOTO CALC1 $
```
When FLG1 is true, control transfers to CALC1.

```
SET IND1 TO IND2**2 $
```
When FLG1 is not true, execution continues with the following statement.

```
SET COUNT TO COUNT-1 $
GOTO CONTINUE $
```
This transfers program control to the statement after alternate processing at CALC1 is complete.

```
CALC1.  SET IND1 TO IND3**2 $
```
When FLG1 is true, control transfers here.

```
        SET COUNT TO NUM+1 $
CONTINUE.  SET FLG1 TO 0 $
```

EXAMPLE 2: BEGIN blocks

```
IF FLG1 THEN
    BEGIN ''YES'' $
      SET IND1 TO IND3**2 $
      SET COUNT TO NUM+1 $
    END ''YES'' $
ELSE BEGIN 'NO'' $
      SET IND1 TO IND2**2 $
      SET COUNT TO COUNT-1 $
    END ''NO'' $
SET FLG1 TO 0 $
```
When FLG1 is true, the BEGIN block after THEN is executed, the block after ELSE is skipped and execution continues with SET FLG1 TO 0.

When FLG1 is not true, the block after THEN is skipped, the block after ELSE is executed and execution continues with SET FLG1 TO 0.

When BEGIN blocks are used, the full logic is visible without chasing from one point to another in the program listing.

---

WORK AREA

1. In example 2 above, what will be executed under the following conditions?
   a. if FLG1=0
   b. if FLG1=1
2. Is there anything wrong with the following? If so, what is wrong?

```
BEGIN $
BEGIN $
END $
```

CORRECT ANSWERS

1. a. the BEGIN block after ELSE and SET FLG1 to 0
   b. the BEGIN block after THEN and SET FLG1 to 0
2. yes - one END was omitted

---

EXAMPLE

Problem Statement:

Given the equation for the area of a circle, A=πr**2, define data units and write a statement to calculate the area of a circle under the following conditions:

1. The radius has a maximum of 12 bits of integer value and 3 bits of fractional value.
2. The area has a maximum of 24 bits of integer value and 6 of fractional value.
3. π=3.14

Step 1: Study the formula. It says, Area equals π (3.14) times the radius squared. Note the size of the radius and the size of the area.

Step 2: AREA and RADIUS must be defined based on the given information.

Note that the total number of bits for each variable is the sum of the values of the integer and fractional portions, therefore:

```
VRBL RADIUS A 15 U 3 $
VRBL AREA A 30 U 6 $
```

Step 3: Now that RADIUS and AREA have been defined, a SET statement must be written to incorporate the original formula.

```
SET AREA TO 3.14*RADIUS**2 $
```

Chapter 3 Summary

The three non-numeric type variables (Boolean, Character and Status) are discussed and demonstrated. Decision making is introduced with the unconditional branch statement (GOTO), followed by conditional branch (IF-THEN-ELSE) statements. Relational operators are introduced and basic program organization using BEGIN blocks is shown.

Before starting the Chapter Quiz, study the following problem example and work the two problems that follow.

## WORK AREA

Problem Statement:

1. Define variables and write a statement to calculate the area of a rectangle, AREA=LNGTHxHEIGHT

   (1) The length and height each have 10 bits of integer value and 4 bits of fractional value.

   (2) The area has 20 bits of integer value and 8 bits of fractional value.

2. Mr. James is employed by CALTRAN as a highway maintenance worker. His salary is $5.85 an hour with time-and-a-half for overtime. Federal income tax is withheld at the rate of 14%. To make it simpler to check the solution against the correct answer, the programmer devised names are given here. Regular hours and overtime hours are needed to calculate pay. Also, net pay and gross pay will be needed. Hours worked are expected to be integer values. Use the following:

   REGHRS
   OVRTIM
   NET
   GROSSPAY

Step 1: Write the variable declarations

Step 2: Set gross pay to the number of hours worked times the salary per hour.

Step 3: Test for overtime-
   a. If there was overtime, add to gross pay
   b. If no overtime, or when overtime calculation is finished-

Step 4: Calculate net pay

## CORRECT ANSWERS

1.
```
VRBL (LNGTH,HEIGHT) A 14 U 4 $
VRBL AREA A 28 U 8 $

SET AREA TO LNGTH*HEIGHT $
```

This was almost exactly like the problem example and should have caused no trouble.

2.
```
VRBL REGHRS I 7 U $
VRBL OVRTIM I 7 U $
VRBL NET F $
VRBL GROSSPAY F $

SET GROSSPAY TO REGHRS*5.85 "REGULAR PAY" $
IF OVRTIM NOT 0
    THEN BEGIN ''EXTRA'' $
        SET GROSSPAY TO GROSSPAY+(OVRTIM*5.85*1.5) $
        END ''EXTRA'' $
SET NET TO GROSSPAY-GROSSPAY*.14 $
```

The correct answer given here is not the only way that this problem can be solved. Most problems have many possible solutions. To check the accuracy of a solution that is different from the one given, use actual numbers and see if the numeric result is the same in both cases.

## Chapter 3 Quiz

1. Name the three non-numeric variable types.

2. Given the following variable declaration: VRBL (FLAG1, FLAG2) B $
   With FLAG1 containing a value of zero and FLAG2 containing a value of one, what would result: (As you work the following problems, consider each result as the start for the next SET statement)
   a. SET FLAG1 TO 1 $
   b. SET FLAG1 TO COMP FLAG2 $
   c. SET FLAG2 TO FLAG1 AND FLAG2 $
   d. SET FLAG1 TO FLAG1 OR FLAG2 $

3. A character constant or character data unit is specified by what letter?

4. In a character type assignment, what will result if:
   a. the source is longer than the receptacle?
   b. the source is shorter than the receptacle?

5. What is the maximum number of status constants that may be defined in a status type variable?

6. What is the maximum length of a status constant?

7. What is the meaning of the following symbols?
   a. LTEQ
   b. GTEQ

8. Write an unconditional branch statement to send the program to SPEC.

9. A conditional statement always starts with what keyword?

10. IF FLG1 LTEQ FLG2 THEN GOTO NEXT $
        SET FLG3 TO 1 $
    What will be executed given the following:
    a. FLG1=1 FLG2=0
    b. FLG1=0 FLG2=1
    c. FLG1=1 FLG2=1

11. A set of instructions that are to be treated as a single unit are contained between what two keywords?

## CORRECT ANSWERS

1. Boolean, character, status

| | | FLAG1 | FLAG2 |
|---|---|---|---|
| | Start | 0 | 1 |
| 2. a. the 0 in FLAG1 changes to 1 | a. | 1 | 1 |
| b. FLAG1 changes from 1 to 0 as COMP FLAG2=0 | b. | 0 | 1 |
| c. FLAG2 changes to 0 | c. | 0 | 0 |
| d. stays 0 | d. | 0 | 0 |

3. H

4. a. rightmost excess characters are not transferred
   b. (1) if source is character constant, excess rightmost characters of receptacle filled with spaces
      (2) if source is character data unit,the excess rightmost characters in the receptacle will be untouched

5. 12

6. 8 characters

7. a. less than or equal to
   b. greater than or equal to

8. GOTO SPEC $

9. IF

10. a. the SET statement
    b. the THEN clause
    c. the THEN clause

11. BEGIN - END

# Chapter 4

# TABLES

# Tables

## A. TABLE DESCRIPTION

Chapters 2 and 3 showed how to define and manipulate individual pieces of data using the six types of CMS-2 variables. However, much of the data for CMS-2 programs consists of a number of items of related information.

As an example, the following partial table shows some of the information a college coach might keep on members of the basketball team.

| Player's Name | Age | Height Nearest Inch | Weight Nearest Pound | Year | Point 4.0 Scale | Team Rank | Is Player Eligible | Is Player on Scholarship |
|---|---|---|---|---|---|---|---|---|
| J. Saxon | 20 | 75 | 170 | 3 | 2.75 | Sec. | Yes | No |
| R. Treat | 19 | 80 | 190 | 2 | 3.32 | First | Yes | Yes |
| J. Walls | 21 | 84 | 205 | 3 | 2.38 | First | No | Yes |

In CMS-2, data such as the above is defined by a group of declarations called a table block. In the CMS-2 table the lines of the printed table are called items and the columns are called fields. Each item has the same configuration of fields as each other item in the table. In the table shown above, the player's name, age, height, weight, and so on, are each fields. All of the information for one player (as J. Saxon) is an item.

## B. TABLE DECLARATIONS

A table block consists of a table declaration, the declarations of any associated data units (fields, for example), and an end-table declaration.

A table declaration specifies the name of the table, its type, how data in the items are to be allocated and the number of items in the table.

FORMAT

```
TABLE   name   type      item          table subscript   $
                        allocation    declaration
```

Explanation:

TABLE - keyword indicating a table declaration

name - name by which the table will be referenced.

type - either H (horizontal) or V (vertical) See paragraph 1, next page.

item allocation - NONE, MEDIUM, or DENSE See paragraph 2, next page.

table subscript declaration - the number of items (lines) in the table. Must be a positive whole number.

H (horizontal) and V (vertical) are directions for the compiler and control the physical arrangement of table items in computer memory. Horizontal arrangement can result in more efficient code, but if a table contains floating-point or character type fields, it must be of vertical type.

NONE, MEDIUM and DENSE are also compiler directions specifying how the fields within an item will be arranged. NONE uses the most computer memory for data while giving the most efficient code for accessing the fields. DENSE uses the least computer memory for data, but results in the least efficient code for accessing. MEDIUM (as the word suggests), is a compromise that usually requires memory space for data between that of NONE and DENSE and results in code of intermediate efficiency. For most purposes, the MEDIUM specification works out best.

When the details of field arrangement are handled by the compiler according to a NONE, DENSE or MEDIUM specification, the table is referred to as compiler-packed. Packing is the assignment of fields to specific memory sites.

A table block is closed by an end-table declaration naming the table to be ended.

FORMAT

```
END-TABLE name $
```

END-TABLE - keyword indicating the end of a table block.
name - the name appearing in the end-table declaration must be the same as the name in the table declaration.

EXAMPLE

Declare a table TEAM to contain the basketball team data illustrated above, assuming that the team consists of 20 members.

```
TABLE TEAM V MEDIUM 20 $
```

TEAM is a vertical table of 20 items with MEDIUM field allocation. Vertical type is chosen because at least one of the fields (the player's name) will contain character type data.

---

WORK AREA

1. In the table example on page 68:
   a. how many items are shown?
   b. how many fields are shown?
2. What are the two types of tables?
3. Which table type must be used if there are floating-point fields in the table?
4. What does compiler-packed mean?
5. Write a table declaration called COMN, containing 32 items.

CORRECT ANSWERS

1. a. 3 items
   b. 9 fields
2. horizontal and vertical
3. vertical
4. the compiler handles the allocation of data items into computer memory
5. TABLE COMN H MEDIUM 32 $ Since there was no mention of character or floating-point data, H was used; however, V would also be correct.

---

## C. FIELD DECLARATIONS

A field is a defined data area (a column) within each of the items of a table. A field declaration specifies the name and properties of the field and (optionally) preset values.

FORMAT

```
FIELD    name    type    P    preset values    $
```

Explanation:

FIELD - keyword indicating a field declaration

name - name by which the field will be referenced

type - (optional) field types are the same as those for variables (I,A,F,B,H,S). If the type is left out, the default type in effect for fields is assumed.

P - (optional) indicates preset value (or values) to follow

preset value(s) - (optional) one or more values to which the field is to be initialized in successive items. Presetting fields will be discussed on page 73.

EXAMPLES

| Declaration | Remarks |
|---|---|
| FIELD NAME H 28 $ | NAME is a character type field of 28 characters |
| FIELD AGE I 7 U $ | AGE is an integer type field of 7 bits, unsigned. |

Field names must be unique within a table, but the same field name may be used in multiple tables or for other data units.

## EXAMPLES

The following is an example of a full table block containing fields to represent each column of the printed table on page 68.

| Declaration | Remarks |
|---|---|
| `TABLE TEAM V MEDIUM 20 $` | TEAM is a vertical, compiler-packed table of 20 items. |
| `FIELD NAME H 28 $` | NAME is a character type field of 28 characters. |
| `FIELD AGE I 7 U $` | AGE is an integer type field of 7 bits, unsigned. |
| `FIELD HEIGHT I 7 U ''TO NEAREST INCH'' $` | Notes are used to clarify meaning and to explain the program. |
| `FIELD WEIGHT I 9 U ''TO NEAREST LB'' $` | |
| `FIELD YEAR I 3 U $` | |
| `FIELD GRADEPNT A 9 U 6 ''4.0 SCALE'' $` | GRADEPNT is a fixed-point field of 9 bits, 6 of which are fractional, unsigned. |
| `FIELD RANK S '1STTEAM','2NDTEAM','JV' $` | RANK is a status type field with 3 possible values. |
| `FIELD ELIGIBLE B $` | ELIGIBLE is a Boolean field; 1 means eligible, 0 means ineligible. |
| `FIELD FNCLAID B $` | 1 means player is receiving financial aid. |
| `END-TABLE TEAM $` | End of table block TEAM. |

---

## WORK AREA

1. In a field declaration, how many field types are possible?

2. Is it permissible to use the same field name twice in a single table?

3. Is it permissible to use a particular field name in two or more different tables?

4. What must be in every field declaration?

CORRECT ANSWERS

1. six: I,A,F,B,H,S - the same as in variable declarations
2. no - the compiler could not differentiate between them
3. yes - this would cause no problem since the table names would be different
4. the keyword FIELD, the programmer devised name and the closing dollar sign

---

## D. REFERENCING DATA WITHIN A TABLE

To find a particular piece of information in a printed table, one "counts down" to the desired line and then "moves over" to the proper column. In a CMS-2 statement a similar reference is achieved by specifying the name followed by an item subscript and a field name, the latter two being enclosed in parentheses and separated by a comma.

FORMAT (within a statement)

```
----table name(item subscript, field name)----
```

Explanation:

| | |
|---|---|
| table name | - name of the table being referenced |
| item subscript | - a positive integer value specifying an item within the table |
| field name | - name of a field within the item indicated by the item subscript. |

A major difference between printed tables and CMS-2 tables is in the numbering of lines and items. Normally, lines of a printed table would be numbered from 1 through n (where there are n lines in the table). The items of a CMS-2 table, however, are numbered from 0 through n-1. For instance, the number of items of table TEAM on the previous page is 20 and the valid item numbers (i.e. item subscripts) are 0 through 19.

The following examples use table TEAM, defined on page 71, as well as the declared variables.

```
VRBL CNTR I 15 U $
VRBL GPNT A 9 U 6 $
```

| Name | Age | Height | Weight | Year | Grade Point | Team Rank | Eligible? | Scholarship |
|---|---|---|---|---|---|---|---|---|
| J. Saxon | 20 | 75 | 170 | 3 | 2.75 | Sec. | Yes | No |
| R. Treat | 19 | 80 | 190 | 2 | 3.32 | First | Yes | Yes |
| J. Walls | 21 | 84 | 205 | 3 | 2.38 | First | No | Yes |

## EXAMPLES

| Statements | Remarks |
|---|---|
| 1. SET GPNT TO TEAM(CNTR,GRADEPNT) $ | The value of field GRADEPNT in the item of TEAM specified by the value of CNTR is assigned to variable GPNT. If CNTR equals 2 (indicating J. Walls), GPNT would be set to 2.38. |
| 2. SET TEAM(CNTR,RANK) TO 'JV' $ | The value of the status constant 'JV' is assigned to field RANK in the item specified by CNTR. |
| 3. IF TEAM(CNTR,NAME) EQ H(J.SAXON) THEN SET TEAM(CNTR,ELIGIBLE) TO 1 $ | Field NAME of the item specified by CNTR will be compared to the character constant H(J.SAXON). If they match, field ELIGIBLE of the item will be assigned the value 1. |

In some cases, it is possible to reference a table by name alone.

| | |
|---|---|
| 4. SET TEAM TO Ø $ | This statement would result in the entire area of the table being set to zero. Such a statement might be used to clear old information out of an area before starting to build up new data from other sources. |

---

## WORK AREA

1. Set up a compiler-packed (medium) table block called SUP with 35 items, containing a field named SERV, integer type with 31 unsigned bits.

2. Clear all of table SUP.

3. What are the valid item subscripts for table SUP?

4. Using VRBL CNTR I 15 U $, write a statement to square the value in field SERV of item CNTR of SUP and place the new value into the field.

5. Write a statement to transfer control to NXTLINE if field SERV of item CNTR is greater than or equal to 1250 octal.

## CORRECT ANSWERS

1. TABLE SUP V MEDIUM 35 $ (type could also be H)
   FIELD SERV I 31 U $  Every table block must be closed
   END-TABLE SUP $  by an end-table declaration.

2. SET SUP TO 0 $

3. 0 through 34

4. SET SUP(CNTR,SERV) TO SUP(CNTR,SERV)**2 $

5. IF SUP(CNTR,SERV) GTEQ O(1250) THEN GOTO NXTLINE $

---

## EXAMPLES

The following additional examples of referencing data within a table are offered to show that an item subscript can be something other than a variable.

1. Change J. Saxon's weight to 225.
   SET TEAM(0,WEIGHT) TO 225 $

2. What player is described by item 1?
   R. Treat

3. Change R. Treat from the first team to the second team.
   SET TEAM(1,RANK) TO '2NDTEAM' $

4. Change J. Walls from not eligible to eligible.
   SET TEAM(2,ELIGIBLE) TO 1 $

   Note: In a Boolean field, the true condition is indicated by a 1 and the not-true condition is indicated by 0.

5. Change R. Treat from scholarship to no-scholarship.
   SET TEAM(1,FNCLAID) TO 0 $

## 1. PRESETTING FIELD DECLARATIONS

The preset portion of a field declaration may specify initial values to which that field is to be set in successive items of the table. The first value is used to preset the field in item 0, the second value is used to preset the field in item 1, etc.

To preset the same value into fields in several consecutive items, enclose the value in parentheses and precede the left parenthesis with a positive integer repeat count(n) representing the number of items to be preset. The effect is the same as writing the value n times.

It is not necessary to preset the field in every item, but items may not be skipped. The total number of preset values, counting each repeat as n values, may not exceed the smaller of the number of items in the table and 256.

Note that the preset value is established when the program is loaded into the computer, but may subsequently be changed by the program, if desired.

EXAMPLES

```
TABLE TEAM V MEDIUM 20 $

FIELD ELIGIBLE B P 20(1) $
          Field ELIGIBLE will be set to 1 in each of the 20 items
          of TEAM.

FIELD GRADEPNT A 9 U 6 P 3.6,3.3,2.7,3(2.8),3.1,2.6,2.5,2.9 $

END-TABLE TEAM $
          Field GRADEPNT will be set to 3.6,3.3,2.7,2.8,2.8,2.8,
          3.1,2.6,2.5,2.9 successively in the first ten items of
          TEAM.  The field is not preset in the remaining ten
          items.
```

---

WORK AREA

1. In the example above, how long is field GRADEPNT?

2. In how many items is field GRADEPNT preset?

3. Write a field declaration to preset an integer field called SEC (containing 6 unsigned bits) of table SECURE (which has 7 items) to 23 in all items.

4. Write the table declaration for the table in problem 3 above, given MEDIUM allocation.

CORRECT ANSWERS

1. 9 bits
2. 10 (items 0 through 9)
3. FIELD SEC I 6 U P 7(23) $
4. TABLE SECURE V (or H) MEDIUM 7 $

---

## E. ADDITIONAL DECLARATIONS

### 1. ITEM-AREA DECLARATIONS

An item-area is a data unit the same size as one item of its associated table and with the same attributes (i.e., fields). It lies outside of the table bounds and is usually used as a working area by the programmer. The item-area declaration must be within a table block and the table named in the table block is referred to as the parent table of the item-area.

The item-area declaration simply specifies the name (or names) of the item-area (or areas) being declared.

FORMAT

```
ITEM-AREA    name(s)  $
```

Explanation:
ITEM-AREA - keyword indicating an item-area declaration

name(s) - name(s) by which the item-area(s) will be referenced. Multiple names are separated by commas.

EXAMPLE

| Declaration | Remarks |
|---|---|
| TABLE TEAM V MEDIUM 20 $ | |
| field declarations | |
| ITEM-AREA MEMBER $ | MEMBER is an item-area of parent table TEAM. MEMBER contains the same fields as every item of TEAM. |
| END-TABLE TEAM $ | |

An item-area is referenced by name and (optionally) a field specification.

EXAMPLES

1. SET MEMBER(NAME) TO H(M. JOHNSON) $
   Field NAME of item-area MEMBER is set to the character constant, filled on the right with space characters.

2. SET TEAM(CNTR) TO MEMBER $
   The entire item of TEAM specified by CNTR is set to the item-area MEMBER. Since an item-area, by definition, has the same field arrangement as an item of the parent table, this is a short way to assign the value of every field in the item to its corresponding field in the item-area.

References to entire tables, table-items and item-areas without a field specification may be made in appropriate assignment statements, but are not legal in IF statements.

3. Legal statement:
   IF MEMBER(ELIGIBLE) EQ Ø THEN GOTO DROP $

   Illegal statement:
   IF TEAM (CNTR) NOT Ø THEN GOTO NXT $
   The reference is ambiguous because the item contains several fields.

---

WORK AREA

1. Where is an item-area declaration placed within a program?

2. What is the common use for an item-area?

3. Write a declaration for two item areas (called SAL1 and SAL2).

4. Write statements to clear SAL1 and SAL2.

5. Write a statement to place the value 3.19 into field GRADEPNT of item-area MEMBER.

CORRECT ANSWERS

1. within a table block
2. as a working area
3. ITEM-AREA SAL1,SAL2 $
4. SET SAL1 TO 0 $ SET SAL2 TO 0 $ or SET SAL1,SAL2 TO 0 $
5. SET MEMBER(GRADEPNT) TO 3.19 $

---

## 2. LIKE-TABLE DECLARATIONS

A like-table declaration specifies the name of a table whose items have the same attributes as its parent table and (optionally) the number of items in the like-table. A like-table is identical to its parent table in every way unless a different number of items is specified. It occupies a completely separate area of computer memory.

FORMAT

| LIKE-TABLE | name | table subscript $ declaration |
|---|---|---|

Explanation:

LIKE-TABLE - keyword indicating a like-table declaration

name - the name by which the like-table will be referenced

table subscript declaration - (optional) specifies the number of items in the like-table. When the table subscript declaration is missing, the like-table has the same number of items as its parent table.

The only place that a LIKE-TABLE declaration may be placed is within a table block. It has the same table type and the same item allocation as its parent table.

The like-table is just another table in every respect, but since the item allocation is identical to the parent table, it simply saves having to write the field definitions all over again.

For example (using the TEAM table defined earlier), if two other team tables were to be established with the identical configuration of the original team, one having the same number of items and the other having 90 items instead of the 20 indicated in the original table, it would be written as follows:

```
TABLE TEAM V MEDIUM 20 $

     (field declarations)

LIKE-TABLE TEAMB $          TEAMB is a compiler packed, vertical ta-
                            ble with the same field configuration and
                            the same number of items as TEAM (which
                            is 20).

LIKE-TABLE TEAMF 90 $       TEAMF is a compiler packed, vertical ta-
                            ble with the same field configuration as
                            TEAM, but with 90 items.

END-TABLE TEAM $
```

Like-table data is referenced in exactly the same manner as table data is referenced. Use of the unique table name alerts the compiler that the particular like-table is being addressed.

---

WORK AREA

1. Write a statement to set:
   a. table TEAM to like-table TEAMB.
   b. table TEAM to like-table TEAMF.

2. Write declarations for a table TAB1 of 12 items with fields FX(floating-point) and CROSSREF (I 4 U), and a like-table TAB2 (same number of items).

3. Using VRBL CNTR I 15 U $ as the item subscript, write a statement to set field CROSSREF of item CNTR in table TAB2 to 5.

CORRECT ANSWERS

1. a. `SET TEAM TO TEAMB $` — Same number of items. All of TEAMB is transferred to TEAM.

   b. `SET TEAM TO TEAMF $` — TEAMF has 90 items, TEAM only 20. The first 20 items of TEAMF will be transferred. The last 70 items of TEAMF will not transfer.

2. Tables with floating point fields must be vertical.

```
TABLE TAB1 V MEDIUM 12 $
    FIELD FX F $
    FIELD CROSSREF I 4 U $
    LIKE-TABLE TAB2 $
    END-TABLE TAB1 $
```

3. `SET TAB2(CNTR,CROSSREF) TO 5 $`

---

## 3. SUB-TABLE DECLARATIONS

A sub-table is a part of a larger (parent) table and consists of a consecutive sub-set of items of the parent table. The name must be unique and the additional necessary information to be given in the sub-table declaration describes the location of the sub-table within the larger parent table. Unlike like-tables and item areas, a sub-table lies wholly within the parent table. A table may have any number of sub-tables, which may be overlapped, adjoining, or completely separate from one another.

FORMAT

```
SUB-TABLE    name  starting item    table subscript declaration    $
```

Explanation:

SUB-TABLE - Keyword indicating a sub-table declaration

name - name by which the sub-table will be referenced

starting item - positive integer value indicating which item of the parent table corresponds to item zero of the sub-table

table subscript declaration - positive integer value indicating the number of items in the sub-table. The number must be such that the sub-table will lie completely within its parent table at program execution time.

A sub-table declaration may only be used within a table block. It has the same table type and the same item allocation as its parent table.

The first item (item zero) of the sub-table is allocated to the same memory location as the specified starting item of the parent table. Successive items of the sub-table are allocated in order to the items of the parent table following the starting item.

Sub-tables may be especially useful in tables whose items are ordered in some manner. For instance, a table containing rainfall records, ordered by years of occurrence, might be divided into sub-tables representing decades, or quarter centuries.

EXAMPLES

1.
```
TABLE VTBL V MEDIUM 10 $
                    Parent table declaration
FIELD VEL1 A 15 U 6 $

FIELD VEL2 A 15 U 6 $

SUB-TABLE VTBLS 2 4 $
                    VTBLS is a table with the same attributes as
                    its parent table VTBL.  It begins in item 2
                    of VTBL and consists of 4 items in all.
END-TABLE VTBL $
                    End-table declaration for the parent table.
```

2.
```
TABLE HTBL H MEDIUM 7 $
                    Parent table declaration
FIELD BEL1 A 15 U 6 $
SUB-TABLE HTBLS 0 4 $
                    HTBLS is a table with the same attributes as
                    its parent table HTBL.  It consists of the
                    first 4 items of HTBL.
END-TABLE HTBL $
                    End-table declaration for the parent table.
```

Sub-tables are referenced exactly like other tables.

---

WORK AREA

1. Set up a compiler-packed (medium) vertical table called PORT with 10 items.

2. Set up a sub-table called PORT1 with 5 items starting in item 3 of the parent table.

3. Close the parent table.

4. Is it necessary to close the sub-table?

5. What two attributes of a sub-table are the same as its parent table?

CORRECT ANSWERS

1. TABLE PORT V MEDIUM 10 $
2. SUB-TABLE PORT1 3 5 $
3. END-TABLE PORT $
4. no
5. type and item-allocation

---

## 4. ITEM-TYPED TABLE

Sometimes a table which has only one field in each item may be needed. In this case, instead of defining a field, the field type can simply be applied to the items. A table which has a specified type applied to its items is called item-typed.

An item-typed table is similar to FORTRAN arrays, providing a table of homogeneous data. Item-typing makes it possible to reference the table data with a specified type, while not having to write a field name in the reference.

An item-typed table is declared by specifying the desired data type in parentheses as the item allocation portion of the table declaration. The types are the same as those for variables and fields.

FORMAT

```
TABLE  name   type  (data type)    number of $
                                       items
```

Explanation:

TABLE - keyword indicating a table declaration

name - name by which the table will be referenced

type - either H or V

data type - same types as for variables and fields

number of items - the number of items in the table

EXAMPLES

1. TABLE XCORDS H (A 18 S 7) 25 $
   END-TABLE XCORDS $

   Table XCORDS consists of 25 items, each of type A 18 S 7.

The single-valued specification for each item removes any ambiguity from an item-only reference and this form is used without field specification.

```
VRBL CNTR I 15 U $

IF XCORDS(CNTR) LT 123.8 THEN GOTO INLIMITS $
```

The fixed-point value of the item indicated by CNTR will be compared to 123.8.

2. TABLE DONORS V (H 28) 10 $
   END-TABLE DONORS $

   Table DONORS consists of 10 items, each of character type, 28 characters in size.

```
VRBL CNTR1 I 15 U $

SET DONORS (CNTR1) TO H(RACHEL VIRGINIA SWANSON) $
```

It is also possible to define an item-typed table with field declarations. This would permit referencing pieces of each item separately, or an entire item with an overall typing.

It should be noted that an item-typed table with no fields cannot be preset. If presets are desired, a field must be defined and preset.

---

WORK AREA

1. Define a vertical table called TAB with 7 items, each of character type, 9 characters in size.

2. Write the end-table declaration for table TAB.

3. Write a statement to set the fourth item of TAB to the character constant TUESDAY.

CORRECT ANSWERS

1. TABLE TAB V (H 9) 7 $

2. END-TABLE TAB $

3. SET TAB(3) TO H(TUESDAY) $

---

Chapter 4 Summary

The method of defining tables is discussed. Associated data units (fields, sub-tables, like-tables and item-areas) are described and explained. The method of referencing data within a table is shown along with the use and allocation of different types of table declarations. The dynamic statements previously introduced are used to demonstrate table accessing and data manipulation.

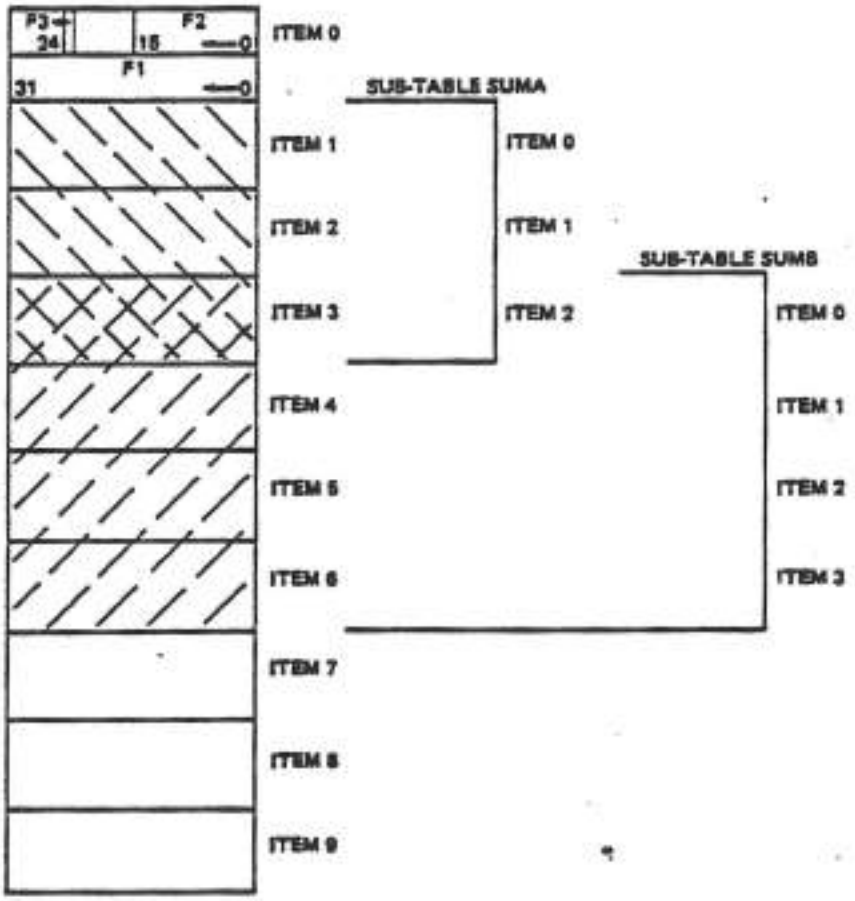
---

TABLE REVIEW

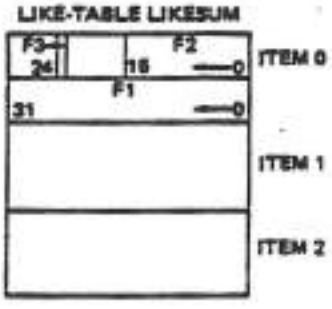
The following data and illustrations are provided to clarify the way tables, fields, sub-tables, like-tables and item-areas appear in computer memory. Note that sub-tables appear inside the parent table, but item-areas and like-tables are allocated outside the parent table.

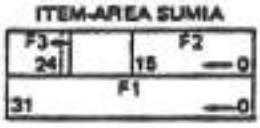
```
TABLE SUMMARY V MEDIUM 10 $

FIELD F1 A 32 S 0 $

FIELD F2 A 16 S 0 $

FIELD F3 B $

SUB-TABLE SUMA 1 3 $

SUB-TABLE SUMB 3 4 $

LIKE-TABLE LIKESUM 3 $

ITEM-AREA SUMIA $

END-TABLE SUMMARY $
```

TABLE SUMMARY




OUTSIDE THE PARENT TABLE



## Organizing Problem Solutions for Compilation

A number of programming problems will be presented throughout the manual. The students that have facilities available for compiling may follow the basic outline presented below to organize the problem solution for compilation. The parts of the outline will be explained in a future chapter.

Steps required to get an actual compile vary from installation to installation and are not included in this manual.

```
DOPROB      SYSTEM $
            OPTIONS (SM,SCR) $
            END-HEAD $
PROB        SYS-PROC $
            LOC-DD $

         insert data declarations here

            END-LOC-DD $
            PROCEDURE PROB $

         insert program statements here

            END-PROC PROB $
            END-SYS-PROC PROB $
            END-SYSTEM DOPROB $
```

## CHAPTER 4 PROBLEMS

1. A large legal firm wishes to have a weekly listing of scheduled court appearances by member attorneys. The necessary data is to be entered into a table called CORTDATE.

   Define table CORTDATE to contain 150 items and the following fields:
   CLIENT, LEADLAWR,ASSIST1,ASSIST2,JUDGE each to be a character type field with a maximum of 30 characters.
   DOCKETNR to contain a numeric value whose range is from 1000 through 9999.
   POSITION to be a status type with the possible values of PRELIMH,INPROG, CONTINUE,APPEAL.
   TIME to be a character type with a maximum of 14 characters; form: month/day/year hour (00/00/00 00:00).
   CST2DATE to contain a numeric value whose range is from $ 000,000.00 through $ 999,999.99.
   CONFLICT to be a Boolean type.

   Table CORTDATE also has two item-areas, called CRTDATA and CRTDATB, and sub-table called FIRSTDAT, that includes the first 50 items of CORTDATE. There is a like-table called ONESDATE which has 10 items.

2. After completing the necessary declarations for problem 1 above, rewrite the appropriate field declarations to include presetting the following information:

   a. The first two clients are William K. Mullen and Lucy B. Hefner.
   b. The first four docket numbers are 1027, 1032, 1103, and 1114.
   c. The judge for each of the first three cases is George L. Swan.
   d. The first three cases are in position INPROG.

## CORRECT ANSWERS

1.
```
TABLE CORTDATE V NONE 150 $
FIELD CLIENT H 30 $
FIELD LEADLAWR H 30 $
FIELD ASSIST1 H 30 $
FIELD ASSIST2 H 30 $
FIELD JUDGE H 30 $
FIELD DOCKETNR I 14 U $
FIELD POSITION S 'PRELIMH','INPROG','CONTINUE','APPEAL' $
FIELD TIME H 14 $
FIELD CST2DATE A 29 U 9 $
FIELD CONFLICT B $
ITEM-AREA CRTDATA,CRTDATB $
SUB-TABLE FIRSTDAT 0 50 $
LIKE-TABLE ONESDATE 10 $
END-TABLE CORTDATE $
```

The table definition could also have been written with MEDIUM or DENSE packing.

2. a.
```
FIELD CLIENT H 30 P H(WILLIAM K. MULLEN),
   H(LUCY B. HEFNER) $
```

b.
```
FIELD DOCKETNR I 14 U P 1027,1032,1103,1114 $
```

c.
```
FIELD JUDGE H 30 P 3(H(GEORGE L. SWAN) ) $
```

Note: the value to be repeated is enclosed in parentheses. Since the value is a character constant, at least one space must come between the right parenthesis ending the constant and the parenthesis ending the repeated value.

d.
```
FIELD POSITION S 'PRELIMH', 'INPROG', 'CONTINUE',
   'APPEAL' P 3('INPROG') $
```

## CHAPTER 4 QUIZ

1. In a table declaration, what is the meaning of the letters H and V?

2. How are table items numbered if the table contains 30 items?

3. Using the variable declaration and the table below, explain statements a and b.

```
VRBL CNTR I 15 U $
VRBL IND1 I 32 S $
TABLE TEAM V MEDIUM 20 $
   FIELD NAME H 28 $
   FIELD AGE I 7 U $
   FIELD HEIGHT I 7 U $
   FIELD WEIGHT I 9 U $
   FIELD YEAR I 3 U $
   FIELD GRADEPNT A 9 U 6 $
   FIELD RANK S '1STEAM', '2NDTEAM', 'JV' $
   FIELD ELIGIBLE B $
   FIELD FNCLAID B $
END-TABLE TEAM $
```

```
a.  SET IND1 TO TEAM(CNTR,AGE) $
b.  SET TEAM(CNTR,YEAR) TO 3 $
```

4. What declarations may appear in a table block?

5. A table block is always closed with what declaration?

6. What type of table can have floating-point fields?

7. When the compiler allocates the fields of a table, the item allocation in the declaration is _____.

8. Within a table, field names must be unique. Is this a true statement?

9. What is given in an item-area declaration and where must the declaration appear?

10. If a field is to be referenced, what must also be specified?

11. What may be different in a like-table from its parent table?

12. What must be specified in a sub-table declaration?

## CORRECT ANSWERS

1. horizontal and vertical

2. 0 through 29

3. a. The value of field AGE of item CNTR is assigned to IND1.
   b. The value of 3 is assigned to field YEAR of item CNTR.

4. field, item-area, like-table and sub-table declarations

5. end-table declaration

6. vertical type

7. NONE, MEDIUM or DENSE

8. yes

9. the name(s) of the area(s) being declared placed within a table block

10. the table and the item or the item-area

11. the number of items in the table and the name of the table

12. the sub-table name, the item in which the sub-table begins and the number of items in the sub-table

# Chapter 5

# LOOP BLOCKS

## Loop Blocks

A group of statements that a programmer wishes to be repeated a number of times is called a loop. The statements are repeated as many times as required before continuing with the program. The keyword used for this operation is VARY and the method used to indicate the number of times through the loop is specified in the basic format.

### A. BASIC FORMAT

The basic loop block consists of the VARY statement, followed by the statement or statements comprising the loop body, finishing with an END statement. The VARY statement has the following format:

FORMAT

| VARY | loop index | FROM | initial value | THRU | final value | BY | change value | $ |
|---|---|---|---|---|---|---|---|---|

Explanation:

| | |
|---|---|
| VARY | - keyword indicating the beginning of a loop block |
| loop index | - data unit used as an index during execution of the loop body |
| FROM | - (optional) keyword indicating that an initial value for the loop index follows |
| initial value | - (optional) numeric expression specifying the initial value of the loop index. Must be present if FROM is present. |
| THRU | - (optional) keyword indicating that the final value for the loop index follows |
| final value | - (optional) numeric expression specifying the final value of the loop index. Must be present if THRU is present. |
| BY | - (optional) keyword indicating that an increment value for the loop index follows |
| change value | - (optional) a numeric expression specifying an amount by which the loop index is to be changed each time through the loop. Must be present if BY is present. |

The END statement has the following format:

FORMAT

| END $ |
|---|

VARYing may be performed either forward (positive increment) or backward (negative increment). The following examples will explain both methods. First, a table will be established as a base upon which to use the VARY statement. This will be a table containing all the grades for a class of 20 students.

```
TABLE GRADES V MEDIUM 20 $
   FIELD NAME H 30 $
   FIELD QUIZ1 A 8 U 5 $
   FIELD QUIZ2 A 8 U 5 $
   FIELD QUIZ3 A 8 U 5 $
   FIELD TERMP A 8 U 5 $
   FIELD FINAL A 8 U 5 $
   FIELD AVGQ A 8 U 5 $
   FIELD ENDGRADE A 8 U 5 $
END-TABLE GRADES $
VRBL CNTR I 8 U $
```

## 1. VARYING FORWARD

Varying forward means starting at the initial value and incrementing the loop counter by a positive value each time through the loop.

EXAMPLE

To initialize the end grade to 0 for all 20 students:

```
VARY CNTR FROM 0 THRU 19 BY 1 $
    SET GRADES(CNTR,ENDGRADE) TO 0 $
END $
```

Steps:
1. The loop index CNTR is given the initial value 0.
2. The set statement comprising the loop body is executed.
3. The value of CNTR is increased by the change value 1.
4. The new value of CNTR is compared to the specified final value 19.
   (a) If CNTR is less than or equal to 19, return to Step 2.
   (b) if CNTR is greater than 19, terminate loop execution.

---

WORK AREA

1. What keyword is used to cause a loop to be established?
2. If the change value in the example above had been a 5:
   a. after the first execution of the loop, CNTR would be on what number?
   b. how many executions of the loop would there be altogether?
3. In the example above, what is the:
   a. initial value?
   b. change value?

CORRECT ANSWERS

1. VARY
2. a. 5
   b. 4 (CNTR values 0,5,10,15)
3. a. 0
   b. 1

---

When varying forward, the FROM and/or BY clauses may be omitted. If FROM is omitted, an initial value of zero is assumed. If BY is omitted, a change value of +1 is assumed. The VARY statement in the previous example could be written:

```
VARY CNTR THRU 19 $
```

This automatically assumes a FROM 0 and BY 1.

## 2. VARYING BACKWARD

Sometimes it might be desirable to start at the end and work in the other direction by decrementing the loop. Decremental indexing, or varying backward, is possible by placing a minus sign in front of the change value and adjusting the control statements appropriately. The same end result would occur if the previous loop block were written as follows:

```
VARY CNTR FROM 19 THRU 0 BY -1 $
    SET GRADES(CNTR,ENDGRADE) TO 0 $
END $
```

Steps:
1. The loop index CNTR is given the initial value 19.
2. The set statement comprising the loop body is executed.
3. The value of CNTR is decreased by the change value 1.
4. The new value of CNTR is compared to the final value 0:
   (a) If CNTR is greater than or equal to zero, return to Step 2.
   (b) If CNTR is less than zero, terminate loop execution.

In the above examples, CNTR was used as both the loop index and the subscript value for table GRADES in the loop body. In each case, the result of executing the loop was to set field ENDGRADE to 0 in each item of table GRADES. The example on page 93 began with the first item of the table and progressed forward, while the example above began with the last item and proceeded backwards.

The range of values specified for a loop index is inclusive. Therefore, 0 THRU 19 BY 1 or 19 THRU 0 BY -1 equals 20 repetitions.

Notes:

1. In loop blocks such as the above, with no beginning of loop checks, the loop body is always executed at least once.

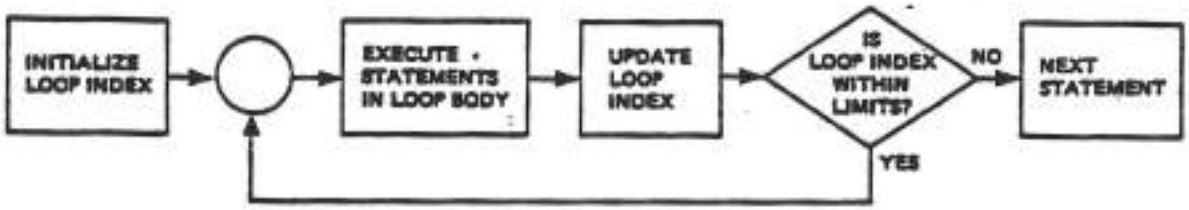
2. The compiler does not check loop controls for logic. If the programmer accidentally writes VARY index FROM 0 THRU 9 BY -1, the index will be initialized to 0 and decremented by 1 on each repetition, with undefined results.

EXAMPLE

```
VRBL POWER A 32 S 0 $
VRBL ROOT A 9 S 0 $

SET POWER TO ROOT $
VARY CNTR FROM CNTR -2 THRU 0 BY -1 $
SET POWER TO POWER*ROOT $
END $
```

The above statements represent a solution to the equation (expressed in words) POWER equals ROOT raised to the power CNTR.
(1) POWER is initialized to the value of ROOT.
(2) For the loop index CNTR must be reduced by 2 (by one, because the first power of ROOT has already been placed in POWER; by one more, because the loop index count is inclusive).

BASIC LOOP BLOCK FLOW




---

WORK AREA

1. Given the following:
```
VARY CNTR FROM 0 THRU 19 BY 1 $
    SET GRADES(CNTR,ENDGRADE) TO 0 $
END $
```
Write loop blocks to cause 10 repetitions:
   a. starting with the first item (item 0).
   b. starting with the last item (item 19).

2. What would each of these loop blocks accomplish?

CORRECT ANSWERS

1. a.
```
VARY CNTR THRU 9 $ (or)
VARY CNTR FROM 0 THRU 9 BY 1 $
    SET GRADES(CNTR,ENDGRADE) TO 0 $
END $
```
   b.
```
VARY CNTR FROM 19 THRU 10 BY -1 $
    SET GRADES(CNTR,ENDGRADE TO 0 $
END $
```

2. a. Sets field ENDGRADE to 0 in the first 10 items of the table
   b. Sets field ENDGRADE to 0 in the last 10 items of the table

---

## 3. USE OF LABEL

If a VARY statement has a label, the label name must appear in the corresponding end statement.

EXAMPLE

```
CLRNAME.  VARY CNTR FROM CNTR THRU 19 $
             SET GRADES(CNTR,NAME) TO H () $
          END CLRNAME $
```

In this example, the initial value of the loop index is whatever value CNTR contains at that point in program execution.

## B. ADDITIONAL CAPABILITIES

### 1. USE OF WHILE AND UNTIL

Additional capabilities available in loop blocks include top-of-loop and/or bottom-of-loop condition testing. The VARY statement format is expanded as follows:

FORMAT

```
VARY   loop    FROM   initial   THRU   final   BY   change
       index          value             value       value

       WHILE   top test   UNTIL  bottom test  $
```

Explanation:

WHILE - (optional) keyword indicating that a top-of-loop test follows.

top test - a conditional expression to be evaluated prior to each execution of the loop body. Must be present if WHILE is used.

UNITL - (optional) keyword indicating that a bottom-of-loop test follows.

bottom test - a conditional expression to be evaluated after each execution of the loop body. Must be present if UNTIL is used.

The following variables and tables are established for use with the examples to follow.

```
VRBL (CNTR,CNTR1) I 8 U $
VRBL (XX,YY) A 14 S 6 $

TABLE VECTORS V MEDIUM 11 $
   FIELD XV1 A 14 S 6 $
   FIELD XV2 A 14 S 6 $
   FIELD YV1 A 14 S 6 $
   FIELD YV2 A 14 S 6 $
END-TABLE VECTORS $

TABLE VALS V MEDIUM 11 $
   FIELD AVGX A 14 S 6 $
   FIELD AVGY A 14 S 6 $
END-TABLE VALS $
```

EXAMPLES

1.
```
VWHIL.VARY CNTR FROM CNTR THRU 0 BY -1
          WHILE CNTR1 GT 0 $
      SET XX TO CNTR1*3 $
      SET CNTR1 TO CNTR1-1 $
      END VWHIL $
```

Steps:

1. Evaluate WHILE condition (No initialization of CNTR needed.)
   (a) If CNTR1 is greater than zero, do Step 2.
   (b) If CNTR1 is less than or equal to zero, terminate the loop and go to the statement after END.

2. Execute loop body

3. Decrement CNTR by the change value 1

4. Compare the new value of CNTR to the final value zero
   (a) If CNTR is greater than or equal to zero, go to Step 1
   (b) If CNTR is less than zero, terminate loop and go to the statement after END.

Note: A loop with a WHILE clause will not execute the loop body even once if the WHILE clause is false on the first evaluation.

```
2.  VUTIL.  VARY CNTR THRU 10 UNTIL XX LT
              VALS(CNTR,AVGX) $
            SET VALS(CNTR,AVGX) TO (VECTORS(CNTR,XV1)+
              VECTORS(CNTR,XV2))*2 $
            END VUTIL $
```

Steps:

1. Initialize CNTR to 0

2. Execute loop body

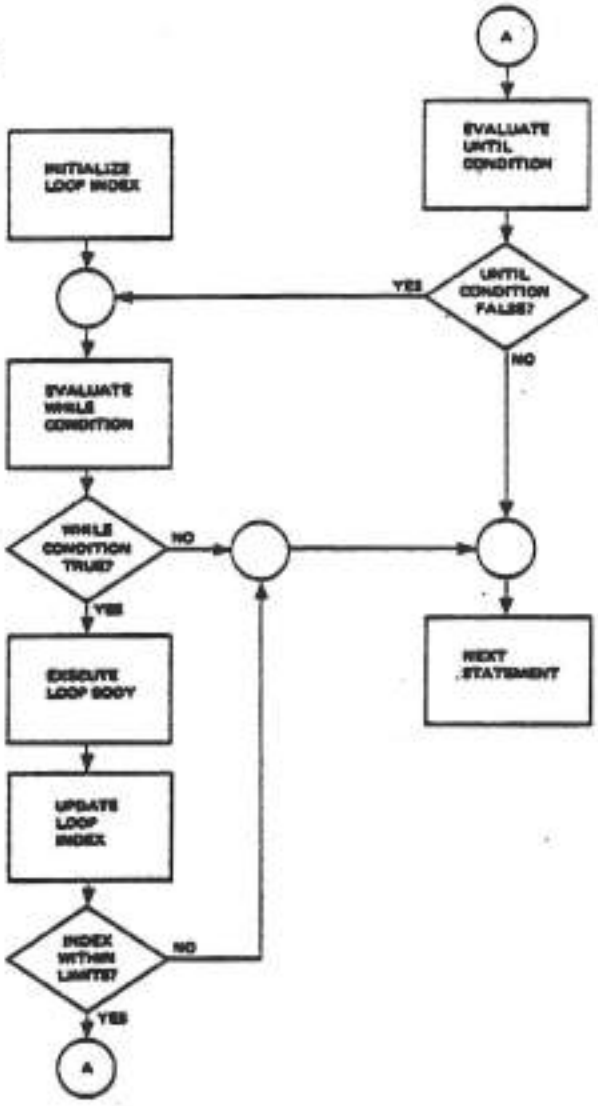
3. Increment CNTR by 1

4. Compare the new value of CNTR to the final value 10
   a. CNTR LTEQ 10 go to step 5
   b. CNTR GT 10 terminate loop and go to the statement after END

5. Evaluate the UNTIL clause
   a. If XX is greater than or equal to VALS(CNTR,AVGX) go to step 2
   b. If XX is less than VALS(CNTR,AVGX) go to the statement after END

Note: When a loop contains top-and/or bottom-of-loop clauses in addition to loop index controls, it is not always obvious which condition caused termination of loop execution.

## LOOP LOGIC WITH WHILE AND UNTIL




---

WORK AREA

```
VRBL CNTR I 15 U $
TABLE POINTS V MEDIUM 50 $
FIELD LABEL H 8 $
FIELD XVAL A 10 S 4 $
FIELD YVAL A 10 S 4 $
FIELD XYSQ A 24 S 8 $
END-TABLE POINTS $
```

1. Write a vary loop to place the value of field XVAL squared plus the value of field YVAL squared into the field XYSQ for each item of the table, while field LABEL is not equal to the string NOMOREIN.

CORRECT ANSWERS

```
1.  VARY CNTR THRU 49 WHILE POINTS(CNTR,LABEL)
        NOT H(NOMOREIN) $
    SET POINTS(CNTR,XYSQ) TO POINTS(CNTR,XVAL)**2+
        POINTS(CNTR,YVAL)**2 $
    END $
```

---

## 2. RESUME PHRASE

A resume phrase specifies that end-of-loop index processing is to be executed. In other words, no more of the loop body is to be executed, but the program will go directly to the index processing (i.e. the END statement) and if the loop is not complete, resume to top-of-loop.

FORMAT

```
RESUME  statement name  $
```

Explanation:

RESUME - keyword indicating end-of-loop processing is to be done

statement name - (optional) If present, the end-of-loop processing for the named loop block is executed. Otherwise, the end-of-loop processing for the most recent loop declared will be executed.

EXAMPLE

```
VARY CNTR THRU 10 $
IF VALS(CNTR,AVGX) GT VALS(CNTR,AVGY)
     THEN RESUME $
SET VALS(CNTR,AVGX) TO VALS(CNTR,AVGY)+2 $
END $
```

When the IF statement is true, execution moves to the END statement, omitting the set statement.

## Chapter 5 Summary

The features available for initiating and controlling loop blocks were explained, including varying backward and forward, the use of labels and special clauses. Examples were confined to straightforward, unnested blocks of statements. More complex loops will be discussed in Appendix C.

## CHAPTER 5 PROBLEMS

The following problems use the table block CORTDATE that was defined at the end of Chapter 4 and, in addition, require defining specified variables.

```
TABLE CORTDATE V MEDIUM 150 $
FIELD CLIENT H 30 $
FIELD LEADLAWR H 30 $
FIELD ASSIST1 H 30 $
FIELD ASSIST2 H 30 $
FIELD JUDGE H 30 $
FIELD DOCKETNR I 14 U $
FIELD POSITIONS 'PRELIM','INPROG','CONTINUE','APPEAL' $
FIELD CST2DATE A 29 U 9 $
FIELD TIME H 14 $
FIELD CONFLICT B $
ITEM-AREA CRTDATA,CRTDATB $
SUB-TABLE FIRSTDAT 0 50 $
LIKE-TABLE ONESDATE 10 $
END-TABLE CORTDATE $
```

### PROBLEM 1

Problem statement:

Write a block of statements to locate each item of CORTDATE in which Mildred R. Marlow appears as the leading lawyer and transfer the items, in first-to-last order, to like-table ONESDATE.

Analysis and hints:

First-to-last order is specified, so it will be necessary to proceed through table CORTDATE, starting with item 0 and checking field LEADLAWR of each item for a match with the given name. The easiest way to do this is to establish a loop block with an index that controls loop execution and can also be used as the item-subscript to CORTDATE. Since the table has 150 items, the index must be able to have values in the range 0 through 149.

Problem 1 continued

When a match is found, the data in that item of CORTDATE is to be placed in ONESDATE. This requires an item-subscript for ONESDATE with a range of possible values of 0 through 9. Both this item-subscript data unit and ONESDATE should be initialized to zero before the loop is declared.

If more than 10 matching items were expected ever to be found, then ONESDATE should be defined with more than 10 items. However, to insure that extra items are not matched and transferred-overlaying other data or instructions, the loop should be designed to terminate if either of two conditions is satisfied:

1. The entire table CORTDATE has been searched

2. ONESDATE has been filled.

For the two index variables needed for this problem, use the names INX1 and INX2. This will simplify the checking of the correct answer against the student's solution.

PROBLEM 2

Problem statement:

CRTDATA is used as a buffer into which is placed update information for table CORTDATE.

Write a loop block that will search CORTDATE for the item with the same docket number as that in CRTDATA and replace the item with the data in CRTDATA.

The previously defined variables may be used as appropriate.

## CORRECT ANSWERS

Problem 1: Comments are included to help explain the instructions

```
VRBL INX1 I 8 U $
VRBL INX2 I 4 U $

01   SET ONESDATE TO 0 ''CLEAR ANY PREVIOUS DATA'' $

02   SET INX2 TO 0 ''INITIALIZE INDEX FOR ONESDATE'' $

03   VARY INX1 THRU 149 UNTIL INX2 EQ 10 $

04   IF CORTDATE(INX1,LEADLAWR) EQ H(MILDRED R. MARLOW)
         THEN BEGIN ''CASE FOUND'' $

05         SET ONESDATE(INX2) TO CORTDATE(INX1)''EXTRACT
             ITEM'' $

06         SET INX2 TO INX2+1''INCREMENT ONESDATE INDEX'' $

07       END ''CASE FOUND'' $

08   END $
```

Another solution to the above problem is shown below to point up the fact that very few problems have only one possible solution. The first three statements do not change, then:

```
04   IF CORTDATE(INX1,LEADLAWR) NOT H(MILDRED R. MARLOW)
         THEN RESUME ''CONTINUE LOOP IF NOT A MATCH'' $

05   SET ONESDATE(INX2) TO CORTDATE(INX1)''ITEM FOUND,
         EXTRACT'' $

06   SET INX2 TO INX2+1''INCREMENT ONESDATE INDEX'' $

07   END $
```

Problem 2:

```
VARY INX1 FROM 149 THRU 0 BY -1 $
IF CORTDATE(INX1,DOCKETNR) EQ CRTDATA(DOCKETNR)
     THEN SET CORTDATE(INX1) TO CRTDATA $
END $
```

In the above loop block all items of CORTDATE are examined. It is clear that once a match is found, it is unnecessary to continue the search (only one case with a given docket number is to be expected). This would appear to be an ideal place for a WHILE or UNTIL clause. Unfortunately, either would result in terminating the loop before the desired data is transferred.

One way to bypass unneeded repetitions is to use a GOTO to exit the loop directly, as follows:

```
         VARY INX1 FROM 149 THRU 0 BY -1 $
         IF CORTDATE(INX1,DOCKETNR) EQ CRTDATE(DOCKETNR)
             THEN BEGIN $
             SET CORTDATE(INX1) TO CRTDATA $
             GOTO FINI $
         END $
FINI.    SET CRTDATA TO 0 $
```

## Student Work Area

This space may be used for notes, comments or responses to the quiz.

## REVIEW QUIZ

At the half-way point in the manual, this quiz will attempt to emphasize the major areas that have been covered in the preceeding chapters.

1. What are the restrictions on identifiers?

2. VRBL (IND1,IND2) I 32 S $
   SET IND1 TO IND2 $
   If IND1 contained the number 5 and IND2 contained a 3, what would be in each after the instruction has been executed?

3. SET IND1 TO A+(B*C)**2 $
   What is the sequence of steps in evaluating the expression above?

4. What are the non-numeric variables and the values each may take?

5. A conditional statement (usually portrayed in a flowchart with a decision block) uses what basic keyword?

6. List the six relational operators.

7. What is the purpose of BEGIN blocks and what keywords are required?

8. Define a table HIGHTEMP that will contain the highest recorded temperature in San Diego for each year from 1900 through 1981, in order. The table has the following associated data units:
   (a) field TFAR - temperature in degrees and 10th's Fahrenheit, range 0.0F through 140.9F
   (b) field TCEL - temperature in degrees and 10th's Celsius, range 0.0 through 60.5C
   (c) field DATE - date on which temperature was recorded, form XX:YYY:ZZ (for example, 31:JUL:57)
   (d) sub-table HALFCEN - includes items for the years 1900 through 1949
   (e) like-table MORETEMP - same number of items
   (f) item-area AYEAR

9. Write a sequence of statements to extract the item of HIGHTEMP containing the highest recorded temperature and place the item in AYEAR. For simplicity's sake, the possibility of duplicate high entries may be ignored.
   (a) a variable TX will be needed as an index, maximum value 81
   (b) think metric-use the Celsius temperature field for the comparison
   (c) to get a base value for comparison, set AYEAR to the first item of HIGHTEMP

## CORRECT ANSWERS

1. (a) cannot be over 8 characters long, (b) must start with alpha and (c) delimiters are not allowed.

2. IND1 equals 3, IND2 equals 3

3. (a) (B*C) evaluated first - within the parentheses
   (b) ** evaluated second - exponent
   (c) A+ evaluated last

4. (a) Boolean - 0 or 1 only
   (b) character - any string of characters
   (c) status - status constants (a string of no more than 8 characters enclosed in single apostrophes)

5. IF

6. EQ, NOT, LT, GT, LTEQ, GTEQ

7. (a) to avoid branching statements and for the facility of using a group of statements where normally only one would be used
   (b) BEGIN END

8.
```
TABLE HIGHTEMP V MEDIUM 82 $
FIELD TFAR A 13 U 5 $ or FIELD TFAR F $
FIELD TCEL A 11 U 5 $ or FIELD TCEL F $
FIELD DATE H 9 $
SUB-TABLE HALFCEN 0 49 $
LIKE-TABLE MORETEMP $
ITEM-AREA AYEAR $
END-TABLE HIGHTEMP $
```

9.
```
VRBL TX I 7 U $
SET AYEAR TO HIGHTEMP(0) $
VARY TX FROM 1 THRU 81 ''THE FIRST ITEM HAS BEEN TAKEN FOR THE
    BASE IN AYEAR AND NEED NOT BE INCLUDED IN THE SEARCH''$
IF HIGHTEMP(TX,TCEL) GT AYEAR(TCEL)
    THEN SET AYEAR TO HIGHTEMP(TX) $
END $
```

# Chapter 6

# FUNCTIONS AND PROCEDURES

## A. PURPOSE OF SUB-PROGRAMS

Large computer programs are usually written in separate parts or segments for the following reasons: (1) to avoid costly duplication of program statements, (2) to make it easier to follow the logic and (3) to simplify the debugging process. The smallest program parts in CMS-2 are the sub-programs called functions and procedures.

A sub-program usually performs a fairly limited task that makes up only a small part of the overall program's job. The total program will normally consist of a controlling (sometimes called an executive) procedure and any number of sub-programs. Each sub-program is called upon when its task is to be accomplished.

## B. FUNCTION DECLARATIONS

A CMS-2Y function is a sub-program of specialized and/or frequently used statements that can be referenced from other sub-programs. Functions must be defined and are then used as an operand in an expression. The definition is accomplished with a function declaration and the use is handled with a function call. When the function is executed it generates a value which replaces the function call as the operand.

The three basic parts of a function definition are the function declaration, the return statement (taking it back to the flow of the program) and the end-function declaration. The statements between the function and end-function declarations are called the function body and the entire sequence is called a function block.

FORMAT

```
FUNCTION    name     (formal input      function   type   $
                      parameter(s))

  function body including at least one:
  RETURN    (return value)  $

END-FUNCTION   name  $
```

Explanation:

| | |
|---|---|
| FUNCTION | - keyword indicating a function declaration |
| name | - name by which the function is to be referenced |
| formal input parameter(s) | - names(s) of the data unit(s) whose value(s) will be replaced by actual value(s) supplied when the function is referenced<br>(1) there must be at least one formal input parameter |

(2) may have a maximum of 25 formal input parameters (AN/UYK-7 only)
(3) formal input parameters must be enclosed in parentheses
(4) the data units that may be used are variables, system indexes and tables

function type - (optional) the type of value returned by the function
(1) the possible types are the same as the types for variables
(2) if the type is missing, the default type in effect for variables is assumed

RETURN - keyword indicating termination of function execution and return of control to the referencing sub-program.
(1) a function must contain at least one return statement and may contain several

return value - the value that is to be returned to the calling sub-program
(1) a return value, enclosed in parentheses, is required for each function return statement

END-FUNCTION - keyword indicating the end of a function block

name - name of the function being ended
(1) the name in the end-function declaration must be the same as the name in the function declaration

EXAMPLES

1.
```
VRBL (FXP1,FXP2) A 22 S 6 $
FUNCTION FNCT1(FXP1,FXP2) B $
   IF FXP1 GTEQ FXP2
      THEN RETURN (1) $
      ELSE RETURN (0) $
END-FUNCTION FNCT1 $
```

FNCT1 is a function with two formal input parameters and a return value of Boolean type. It returns the value 1 (true) when the value of the first input parameter is greater than or equal to the value of the second input parameter. Otherwise it returns the value 0 (false).

---

WORK AREA

1. What are the three basic parts of a function?
2. What is the maximum number of formal input parameters allowed?
3. What two pieces of information must be in every function block (in addition to keyword, name and end)?
4. List the possible function types for function declarations.
5. Indicate whether the following function declarations are correct or incorrect. If incorrect, what is wrong?
   a. FUNCTION ABLE(ALPHA,BETA,GAMMA) H 6 $
   b. FUNCTION BAKER H 7 $
   c. FUNCTION CARRY(HOLD) I 16 S,A 32 S 4 $

CORRECT ANSWERS

1. declaration, return statement, end declaration

2. 25

3. formal input parameter and return statement

4. I, A, F, B, H, S - same as for variables

5. a. correct
   b. incorrect - has no input parameter
   c. incorrect - may have only one output

---

EXAMPLES continued

2.
```
VRBL (ARG1,ARG2) I 32 S $
VRBL MINRMAX S 'MIN','MAX' $

FUNCTION MINMAX(ARG1,ARG2,MINRMAX) I 32 S $
   IF ARG1 EQ ARG2 THEN RETURN(0)''NO MIN OR MAX IF EQUAL'' $
   IF MINRMAX EQ 'MIN'
        THEN BEGIN''MINIMUM REQUESTED'' $
          IF ARG1 LT ARG2 THEN RETURN(ARG1) $
                          ELSE RETURN(ARG2) $
          END''MIN'' $
        ELSE BEGIN''MAXIMUM'' $
          IF ARG1 GT ARG2 THEN RETURN(ARG1) $
                          ELSE RETURN(ARG2) $
          END ''MAX'' $
END-FUNCTION MINMAX $
```

MINMAX determines which of two input parameters is larger or smaller, depending upon the value of the third input parameter. Should the compared values be equal, a value of zero is returned.

## 1. CALLING A FUNCTION

There are two classes of functions in CMS-2Y:

a. User functions are declared and defined by the programmer as illustrated on the previous pages.

b. Intrinsic functions are an integral part of the language.

Functions are all referenced in the same manner as an operand in an expression. (A function call is more formally referred to as a function reference.) When the expression is evaluated, the function reference is replaced by its value. The type of value and the method of determination depend upon the definition of the function.

Examples in the following discussion will be limited to user function references. However, the same principles apply to intrinsic function references, which will be covered in Chapter 10.

A function reference (function call) has the following format:

FORMAT

```
... function name (actual input parameter(s))...
```

Explanation:

function name - the name of the function being referenced

actual input parameter(s) - (optional) value(s) that will be transferred to the corresponding formal input parameter(s) before executing the function body. An actual input parameter may be an expression.

A function reference (function call) is a part of a statement. It places specified constants, data units, etc., into the data units specified by the formal input parameters, then passes control to the function. When processing has been completed, a single value (much like the result of an expression) is returned to the statement that called the function.

---

WORK AREA

How is a function reference used and what happens?

CORRECT ANSWER

It is used as an operand in an expression. When the function is evaluated, its value replaces the function call in the statement.

---

## 2. EVALUATING A FUNCTION REFERENCE (CALL)

The evaluation of a function reference takes three steps:

(1) The value of each actual input parameter is assigned to the corresponding formal input parameter, if necessary.
   (a) The first actual input parameter corresponds to the first formal input parameter, the second actual to the second formal, etc.
   (b) The values of the actual input parameters must be compatible with the types of their corresponding formal input parameters.
   (c) If an actual input parameter and the matching formal input parameter is the same, no transfer occurs.

(2) The body of the function is executed. Execution of the function body ends with completion of a function return statement.

(3) The value specified by the function statement becomes the value of the function reference.

EXAMPLES

The examples that follow refer to the functions shown on pages 111 and 112.

```
VRBL FLG1 B $
VRBL (CNTR,CNTR1) I 15 U $
VRBL (RAD1,RAD2,RD3) A 12 S 3 $

TABLE TT H MEDIUM 100 $
   FIELD FX1 A 15 S 0 $
   FIELD FX2 A 15 S 0 $
   LIKE-TABLE TT2 $
   ITEM-AREA TTIA $
END-TABLE TT $
```

1. `SET FLG1 TO FNCT1(RAD1,RAD2) $`
   (1) The values of RAD1 and RAD2 are assigned to FXP1 and FXP2.
   (2) The function body is executed, returning a value of 1 if RAD1 is greater than or equal to RAD2, otherwise 0.
   (3) The value is assigned to FLG1.

2. SET TTIA(FX1) TO MINMAX(CNTR,CNTR1,'MIN') $
   (1) The values of CNTR,CNTR1 and 'MIN' are assigned to ARG1,ARG2 and MINRMAX, respectively.

   (2) The function body is executed, returning the smaller of CNTR and CNTR1, or zero if they're equal.

   (3) The value is assigned to field FX1 of TTIA.

3. IF MINMAX(TT(CNTR,FX1),TT(CNTR,FX2),'MAX')
   LT 1500 THEN GOTO BLEWIT $

   (1) The function reference is evaluated and returns the value of the larger of the two fields, or zero if the values are equal.

   (2) The value is then compared to 1500.

Note that a function reference may normally be used anywhere that a single value of the same type is appropriate.

---

WORK AREA

Problem: Study the statements below, then write out a brief explanation of what is happening through this sequence of instructions.

```
IF FNCT1(CNTR,CNTR1)THEN
   BEGIN ''1'' $
     VARY CNTR FROM CNTR-1 THRU 0 BY -1 $
     SET TT2(CNTR,FX2) TO
        MINMAX(TT(CNTR,FX1),TT(CNTR,FX2),'MAX') $
     END $
   END '' 1 '' $
   ELSE BEGIN '' 2 '' $
     SET CNTR,CNTR1 TO 0 $
     IF FNCT1(RAD1,RAD2) THEN SET TT2 TO TT $
     ELSE GOTO BLEWIT $
   END '' 2 '' $
```

CORRECT ANSWERS

(1) When CNTR is greater than or equal to CNTR1, do the first BEGIN block.
   (a) Set field FX2 in the items of table TT2 specified by the values of CNTR to the larger of fields FX1 and FX2 in the corresponding item of table TT.

(2) When CNTR is less than CNTR1, do the second BEGIN block.
   (a) Clear CNTR,CNTR1.
   (b) If RAD1 is greater than or equal to RAD2, set the entire table TT2 to table TT. If not, something is wrong, go to an error processing location called BLEWIT.

It may be of some value to the student to assign actual values to the fields and then follow the flow of events numerically. This method not only aids the ability to understand the instructions, but also tends to prove the logic of the instruction sequence.

---

C. PROCEDURES

A procedure is the most often used method of processing within a CMS-2 program. It is a self-contained sequence of statements that appears only once in a program, but may be called (entered) from any number of other subprograms, whenever it is desired to execute that sequence of statements. When the procedure finishes processing, it returns to the statement following the one that "called" the procedure.

A procedure declaration specifies the name of the procedure and (optionally) its formal parameters. The statements to be executed when a procedure is referenced are called the procedure body. A procedure is ended by an end-proc declaration and the entire sequence of statements, beginning with the procedure declaration and ending with the end-proc declaration, is called a procedure block.

FORMAT

```
PROCEDURE   name   $

        procedure body

END-PROC    name   $
```

Explanation:

PROCEDURE - keyword indicating a procedure declaration

name - name by which the procedure will be referenced

procedure body - statements to be executed each time the procedure is referenced (called)

END-PROC - keyword indicating the end of a procedure block

name - name of the procedure being ended - must match the name in the procedure declaration.

EXAMPLE

```
     VRBL (IND1,IND2,IND3) I 32 S $

     TABLE TT H MEDIUM 100 $
        FIELD FX1 A 15 S 0 $
        FIELD FX2 A 15 S 0 $
        LIKE-TABLE TT2 $
        ITEM-AREA TTIA,TTIB $
     END-TABLE TT $

PROCEDURE INIT $
COMMENT CLEARS WORK AREAS $
SET IND1,IND2,IND3 TO 0 $
SET TTIA,TTIB TO 0 $
SET TT TO 0 $
SET TT2 TO 0 $
END-PROC INIT $
```

INIT is declared as a procedure with no formal parameters. Each time INIT is referenced (called), the specified data units are cleared to zero.

---

WORK AREA

1. What are the parts of a procedure block?

2. What happens when procedure execution is complete?

3. Write statements to define a procedure called CHNGTBLS that sets table TT2 to table TT.

CORRECT ANSWERS

1. a. procedure declaration (keyword PROCEDURE, name given to the procedure)
   b. procedure body
   c. end-procedure declaration (END-PROC keyword, same name as in the original declaration)
2. program processing returns to the calling sub-program
3. PROCEDURE CHNGTBLS $
   SET TT2 TO TT $
   END-PROC CHNGTBLS $

---

## 1. FORMAL PARAMETERS

Procedures may have three kinds of formal parameters (input,output,exit). Any of them may be omitted, but if they are used, they must be in the prescribed order as shown in the format below. The basic procedure declaration is expanded as follows:

FORMAT

```
PROCEDURE  name  INPUT   formal input   OUTPUT  formal output
                         parameter(s)           parameter(s)

     EXIT     formal exit   $
              parameter(s)
```

Explanation:

| | |
|---|---|
| INPUT | - (optional) keyword indicating that formal input parameter(s) follow |
| formal input parameters | - (optional) name or names of the data unit(s) whose value(s) will be replaced by the value(s) of actual input parameter(s) when the procedure is called |
| OUTPUT | - (optional) keyword indicating that formal output parameter(s) follow |
| formal output parameters | - (optional) the name or names of the data unit(s) whose values(s) will be transferred to actual output parameter(s) at the end of normal procedure execution |
| EXIT | - (optional) keyword indicating that formal exit parameter(s) follow (EXIT refers to the AN/UYK-7 computer only) |
| formal exit parameters | - (optional) a name or names that can be used in a procedure return statement (the formal exit names are matched to statement names provided by the calling sub-program.) |

Rules:

(1) EXIT parameters are implemented for the AN/UYK-7 computer only.
(2) A procedure may have a maximum of 25 input, 25 output and 10 exit parameters (AN/UYK-7 only).
(3) The data units that may be used as formal input and output parameters are variables (including item-areas), tables and system indexes.
(4) If one of the keywords (INPUT,OUTPUT or EXIT) appears in a procedure declaration, it must be followed by an appropriate formal parameter name.
(5) The data units used as formal parameters must have been defined prior to their use.

2. PROCEDURE RETURN

A procedure return terminates execution of the procedure and gives control back to the calling sub-program. A procedure with only one exit needs no return statement, since the end-proc statement accomplishes the same thing.

When a return statement is used, it may be either simple or it may specify the name of a formal exit parameter (see below).

FORMAT

| RETURN | name | $ |
|---|---|---|

Explanation:

RETURN - (optional) keyword indicating return of control to the calling sub-program

name - (optional) name of a declared formal exit parameter

---

WORK AREA

1. Considering only the keywords, which of the following would be incorrect?

   a. PROCEDURE --- INPUT --- $
   b. PROCEDURE --- OUTPUT --- EXIT --- $
   c. PROCEDURE --- OUTPUT --- INPUT --- $
   d. PROCEDURE --- INPUT --- EXIT --- $
   e. PROCEDURE --- EXIT --- $

2. What is the largest number of input and output parameters allowed for a procedure?

3. What are the two statements to cause a return to the flow of a program after a procedure has been executed ?

## CORRECT ANSWERS

1. all correct except item c INPUT must appear before OUTPUT

2. 25 of each

3. RETURN statement
   END-PROC declaration

---

## EXAMPLE

```
          TABLE WA H (A 32 S Ø) 1ØØ $
          END-TABLE WA $

          VRBL(LX,NUMS) I 8 U $
          VRBL RESULT A 32 S Ø $

Ø1        PROCEDURE AVRG INPUT WA,NUMS OUTPUT RESULT $

Ø2        COMMENT THIS PROCEDURE RETURNS THE AVERAGE
                  OF NUMS NUMBER OF VALUES (WITH A
                  MAXIMUM OF 1ØØ) FROM TABLE WA $

Ø3        SET RESULT TO Ø $

Ø4        VARY LX FROM NUMS-1 THRU Ø BY -1 $

Ø5        SET RESULT TO RESULT+WA(LX) $

Ø6        END $

Ø7        SET RESULT TO RESULT/NUMS $

Ø8        END-PROC AVRG $
```

The detailed analysis of this procedure may be found on the following page.

Statement-by-statement analysis of procedure AVRG:

01 AVRG is declared as a procedure with 2 formal input parameters and 1 formal output parameter.

a. Including a specification for the number of values to be averaged (NUMS) makes the procedure more general, as it permits the calling sub-program to transfer any number of values (up to the maximum of 100) declared for table WA.

02 The comment statement at the beginning of the procedure provides a summary of what the procedure accomplishes..

03 Initialize (set to zero) the data unit to be used in the summation.

04 Establish a loop to vary through the specified number of values.

05 Sum the values into variable RESULT.

06 End of loop block.

07 Calculate average and place in RESULT.

08 End of AVRG procedure block.

The final statement causes control to be returned to the calling sub-program. The value of RESULT will then be assigned to the actual output parameter.

---

WORK AREA

1. What, if anything, is wrong with the following statements?

a. PROCEDURE COMPAR OUTPUT (A26,A27) $

b. PROCEDURE INPUT HOLD,HOLD1 OUTPUT AB,AC $

c. PROCEDURE SIG1 INPUT SORD OUTPUT FIX,FIX1 $

d. PROCEDURE SIG2 OUTPUT BAT1,BAT2,EXIT IND $

e. PROCEDURE COMP1 INPUT REG,REG1 EXIT NXT $

CORRECT ANSWERS

1. a. parentheses not allowed, it should be: ... OUTPUT A26,A27 $
   b. no name given to the procedure
   c. nothing wrong
   d. there should not be a comma after the second output parameter
   e. nothing wrong

---

## 3. LOCAL INDEXES

A local index is a temporary variable whose scope is the particular sub-program in which it is declared. It is quite useful for variables which are used constantly throughout a procedure.

When the target computer has hardware index registers, the value of the local indexes are held in these hardware index registers throughout the execution of the sub-program. When there are no hardware index registers or more local indexes were declared than the number of registers available, the indexes for which there are no registers will be assigned to memory.

A local index declaration specifies the name or names of integer variables whose values are to be held in index registers (if available) during the execution of a sub-program.

FORMAT

```
PROCEDURE     name   $
LOC-INDEX     name or names   $

(procedure continued)

END-PROC   name   $
```

When used, the local index declaration must be the first statement, other than comments, following the sub-program declaration (function or procedure). The local index name is valid only within the sub-program in which it was declared.

Any number of local indexes may be declared. If the number of available index registers is fewer than the number of local indexes, the extra names will be assigned to memory location.

EXAMPLE

```
PROCEDURE PROC1 $
LOC-INDEX AB,BB,CB $


END-PROC PROC1 $
```

AB,BB and CB are declared as local indexes within the PROC1 procedure block. If three index registers are available each local index will be assigned to a register. Any local indexes in excess of available index registers will be assigned to memory locations.

---

WORK AREA

1. What is the scope of a local index?

2. Local indexes are declared in what sub-programs?

3. Is it OK for a programmer to use more local indexes than there are available index registers? If so, what happens?

4. Using the given table definition, write a procedure called DOGRADES to satisfy the following conditions:

   a. Table GRADES is both a formal input and a formal output parameter.
   b. DOGRADES has a local index LX.
   c. For each item of table GRADES, the average of the three quizzes is to be calculated and placed into field AVGQ.
   d. If either field TERMP or field FINAL of an item is found to be zero, field INCOMP is set to 1 and processing moves to the next item.
      If both TERMP and FINAL contain non-zero values, the average of fields AVGQ,TERMP and FINAL is placed into field ENDGRADE.

TABLE DEFINITION

```
TABLE GRADES V MEDIUM 20 $
FIELD NAME H 32 $
FIELD QUIZ1 A 15 U 12 $
FIELD QUIZ2 A 15 U 12 $
FIELD QUIZ3 A 15 U 12 $
FIELD AVGQ A 15 U 12 $
FIELD TERMP A 15 U 12 $
FIELD FINAL A 15 U 12 $
FIELD ENDGRADE A 15 U 12 $
FIELD INCOMP B $
LIKE-TABLE SECTION1 $
LIKE-TABLE SECTION2 $
END-TABLE GRADES $
```

## CORRECT ANSWERS

1. only used in the particular sub-program in which it is declared

2. function or procedure

3. yes - others assigned to memory locations

4. problem solution:

```
01  PROCEDURE DOGRADES INPUT GRADES OUTPUT GRADES $

02  LOC-INDEX LX $

03  VARY LX FROM 19 THRU 0 BY -1 or VARY LX THRU 19 $

04  SET GRADES(LX,AVGQ) TO (GRADES(LX,QUIZ1)+
        GRADES(LX,QUIZ2)+GRADES(LX,QUIZ3))/3 $

05  IF GRADES(LX,TERMP) EQ 0 OR GRADES(LX,FINAL)
        EQ 0 THEN BEGIN $
06          SET GRADES(LX,INCOMP) TO 1 $

07          RESUME $

08          END $

09  SET GRADES(LX,ENDGRADE) TO (GRADES(LX,AVGQ)+
        GRADES(LX,TERMP)+GRADES(LX,FINAL))/3 $

10  END $

11  END-PROC DOGRADES $
```

To understand better the way the sub-program works, move the following two sets of data through it.

| NAME | QUIZ1 | QUIZ2 | QUIZ3 | AVGQ | TERMP | FINAL | ENDGRADE |
|---|---|---|---|---|---|---|---|
| Brown | 2.3 | 3.9 | 2.0 | (2.73) | 2.8 | 0 | (INC) |
| Cary | 3.8 | 2.7 | 3.0 | (3.16) | 3.4 | 3.0 | (3.19) |

## 4. CALLING A PROCEDURE

A procedure call transfers control to the named procedure and assigns actual values (which are specified within the call) to the formal input parameters specified in the procedure statement if necessary.

A procedure is called by its name, optionally followed by actual parameters when appropriate. The keyword PROCEDURE is not used.

FORMAT

```
procedure  INPUT  actual input  OUTPUT  actual output   EXIT
 name             parameters            parameters

                actual exit  $
                parameters
```

Explanation:

| | |
|---|---|
| procedure name | - the name of the procedure being called |
| INPUT | - (optional) keyword indicating that one or more actual input parameters follow |
| actual input parameters | - (optional) an expression or data unit whose value will be transferred to the matching formal parameter of the referenced procedure prior to execution of the sub-program |
| OUTPUT | - (optional) keyword indicating that one or more actual output parameters follow |
| actual output parameters | - (optional) the name of a receptacle to receive the value of the matching formal output parameter after normal completion of sub-program execution |
| EXIT | - (optional) keyword indicating that one or more actual exit parameters follow |
| actual exit parameters | - (optional) the name of a statement to which program control passes if execution of the sub-program is terminated by the matching abnormal exit |

Rules:

(1) A procedure must be defined before it can be called.

(2) Actual parameters are matched one-to-one to formal parameters in left-to-right order.

(3) If formal input parameters were declared, the word INPUT must appear in the procedure call statement even if all actual input parameters are omitted.

(4) If formal output parameters were declared, the word OUTPUT must appear in the procedure call statement even if all actual output parameters are omitted.

(5) The omission of actual input or output parameters implies that for this procedure call the value of that particular formal parameter is irrelevant. When more than one formal input or output parameter is involved the omitted parameters must be represented by commas to assure proper matching.

(6) If formal exit parameters were declared, the word EXIT and a name for each exit parameter must appear in the procedure call statement. Actual exit parameters may never be omitted.

(7) Actual input and output parameters must be of compatible data types with the corresponding formal parameters.

(8) If an actual input or output parameter is the same as the corresponding formal parameter, no transfer of data occurs.

The following steps are taken in the execution of a procedure call:

(1) The value of each actual input parameter is transferred to the corresponding formal input parameter, if necessary.

(2) Control is transferred to the called procedure and the procedure body is executed. Execution of the procedure body is terminated by the execution of a procedure return statement or an end-procedure declaration.

(3) (a) If the called procedure was terminated by executing a normal return statement or an end-procedure declaration, the value of each formal output parameter is transferred to the corresponding actual output parameter, if necessary.
(b) If the called procedure was terminated by executing a return statement with a formal exit name, output parameters are not transferred and program control passes to the corresponding statement named in the procedure call.

EXAMPLE

Problem: Using the table data declarations on page 121, write a statement that will call procedure DOGRADES to calculate final grades for SECTION1.

Solution:

```
DOGRADES INPUT SECTION1 OUTPUT SECTION1 $
```

Analysis:

a. The data in actual input parameter table SECTION is transferred to formal input parameter table GRADES.
b. Procedure DOGRADES is executed.
c. The data in formal output parameter table GRADES is transferred to actual output parameter table SECTION1.

---

WORK AREA

1. Given the following declarations:

```
TABLE FACS V MEDIUM 25 $
   FIELD AMT F $
   FIELD WHN S 'ONE','TWO','THREE' $
   FIELD GONOGO B $
   FIELD KNT A 12 U 3 $
   ITEM-AREA FACSIA $
END-TABLE FACS $
VRBL (ENTA,ENTB,ENTC) A 15 S Ø $
VRBL (FLG1,FLG2) B $

PROCEDURE SWPT INPUT FACSIA,ENTA OUTPUT FLG1 $
```

a. Write a statement to call SWPT with ENTC,FLG2, and the item of FACS indicated by ENTB as actual inputs and outputs as appropriate.

b. Suppose a call were written as follows, with an actual input parameter identical to the formal parameter. What would happen?

```
SWPT INPUT FACSIA,ENTC OUTPUT FLG2 $
```

## CORRECT ANSWERS

1. a. SWPT INPUT FACS(ENTB),ENTC OUTPUT FLG2 $

   b. When the actual and formal parameters are identical, the compiler recognizes that no transfer of data is required.

---

## Chapter 6 Summary

The declarations and referencing (calling) of functions and procedures are explained, including methods of calling and evaluating functions, the use of procedure returns, formal parameters and methods of calling and evaluating procedures. The most suitable use of each type of routine is discussed and local-indexes are introduced.

## CHAPTER SIX PROBLEMS

1. Write a function that accepts as input the number of miles driven and the number of gallons of gasoline used and returns the miles per gallon, each to the nearest tenth.

   The maximum value for miles driven is 25,000.0 and the maximum value for gallons used is 9,000.0. It is not anticipated that miles per gallon exceed 100.0.

   The following names are provided so that the solution should be compatible with the the correction answer given.

   variables: NROFMILS (number of miles driven)
   GALSUSED (number of gallons used)
   MPG (miles per gallon)
   function: GETMPG

2. Fifteen sub-compact cars and trucks participated in a fuel economy test.

   a. Define a table to hold the following information about each vehicle:
      (1) A 12 character code name.
      (2) Distance driven - maximum 25,000.0 miles.
      (3) Gasoline used - maximum 9,000.0 gallons.
      (4) MPG - maximum 100.0 miles per gallon.

   b. Define a suitable loop index and write a loop that will call function GETMPG and result in calculating and storing in the appropriate field, the miles per gallon figure for each vehicle.

## CORRECT ANSWERS

1.
```
VRBL NROFMILS A 20 U 5 $
VRBL GALSUSED A 19 U 5 $
VRBL MPG A 12 U 5 $

FUNCTION GETMPG(NROFMILS,GALSUSED) A 12 U 5 $
SET MPG TO NROFMILS/GALSUSED $
RETURN (MPG) $
END-FUNCTION GETMPG $
```

The function body could be written as one statement, cancelling the need for variable MPG.

```
RETURN(NROFMILS/GALSUSED) $
```

2. a.
```
TABLE SCOMPACT V MEDIUM 15 $
FIELD CODENAME H 12 $
FIELD MILES A 20 U 5 $
FIELD GALLONS A 19 U 5 $
FIELD MPG A 12 U 5 $
END-TABLE SCOMPACT $
```

b.
```
VRBL LX I 7 U $

VARY LX THRU 14 $
SET SCOMPACT(LX,MPG) TO GETMPG(SCOMPACT(LX,MILES),
      SCOMPACT(LX,GALLONS)) $
END $
```

The VARY statement could also be written as follows:

```
VARY LX FROM 14 THRU 0 BY -1 $
```

3. The following problem is presented and executed. The student is to analyze the procedure (statement by statement) in the same manner that the analysis was accomplished on page 121.

Problem Statement:

An experiment was performed on different occasions, producing two tables of results. Using the given data declarations, write a procedure that will calculate the difference between corresponding entries in the first and second tables and will then call on procedure AVRG (page 118) to calculate the average difference.

Data Declarations:

```
TABLE RUFDATA1 H (A 20 S 0) 50 $
   LIKE-TABLE RUFDATA2 $
   LIKE-TABLE DIFFS $
END-TABLE RUFDATA1 $
VRBL AVGDIFF A 32 S 0 $
```

Procedure:

```
01  PROCEDURE FINDDIFF $

02  LOC-INDEX DIFX $

03  VARY DIFX FROM 49 THRU 0 BY -1 $

04  SET DIFFS(DIFX) TO RUFDATA1(DIFX)-
        RUFDATA2(DIFX) $
05  END $

06  AVRG INPUT DIFFS,50 OUTPUT AVGDIFF

07  END-PROC FINDDIFF $
```

CORRECT ANSWER

3. Analysis:

01 FINDDIFF is declared as a procedure with no formal parameters.

02 DIFX is declared as a local-index whose value is to be held in an index register (if available) during execution of procedure FINDDIFF.

03 Establish a loop to vary through all the values in the tables.

04 Calculate the difference between corresponding values in table RUFDATA1 and RUFDATA2 and place in the same numbered item in table DIFFS.

05 End of loop body.

06 Call AVRG to get the average of values in table DIFFS.

(a) The data in actual input parameter DIFFS is transferred to formal input parameter WA and the value of actual input parameter numeric constant 50 is transferred to formal input parameter NUMS.
(b) Procedure AVRG is executed.
(c) Control is returned to FINDDIFF and the value of formal output parameter RESULT is assigned to the actual input parameter AVGDIFF.

07 End of FINDDIFF procedure block.

# CHAPTER 7

# MORE ABOUT TABLES

## A. USER PACKED TABLES

An advantage of the compiler-packed tables introduced in Chapter 4 is that the programmer need not be concerned with the actual, physical layout of the table data in memory. At times, however, program constraints or debugging requirements make it necessary to know, or even to control, how the data appears in memory.

In Chapter 4, the items of a CMS-2 table were equated to lines of a printed table and the fields to columns. When such a table is assigned to computer memory, hardware and software addressing techniques make it advantageous that it occupy an integral number of computer words. (Computer word sizes and addressing may be reviewed in Appendix D, Section 2.)

CMS-2 tables consist of a specified number of items, each of which contains an integral number of words, determined either by the compiler or the programmer. Compiler-packed tables were discussed in Chapter 4. This chapter will be concerned with user-packed tables, in which the user has the option of specifying the number of words per item as well as the number of items.

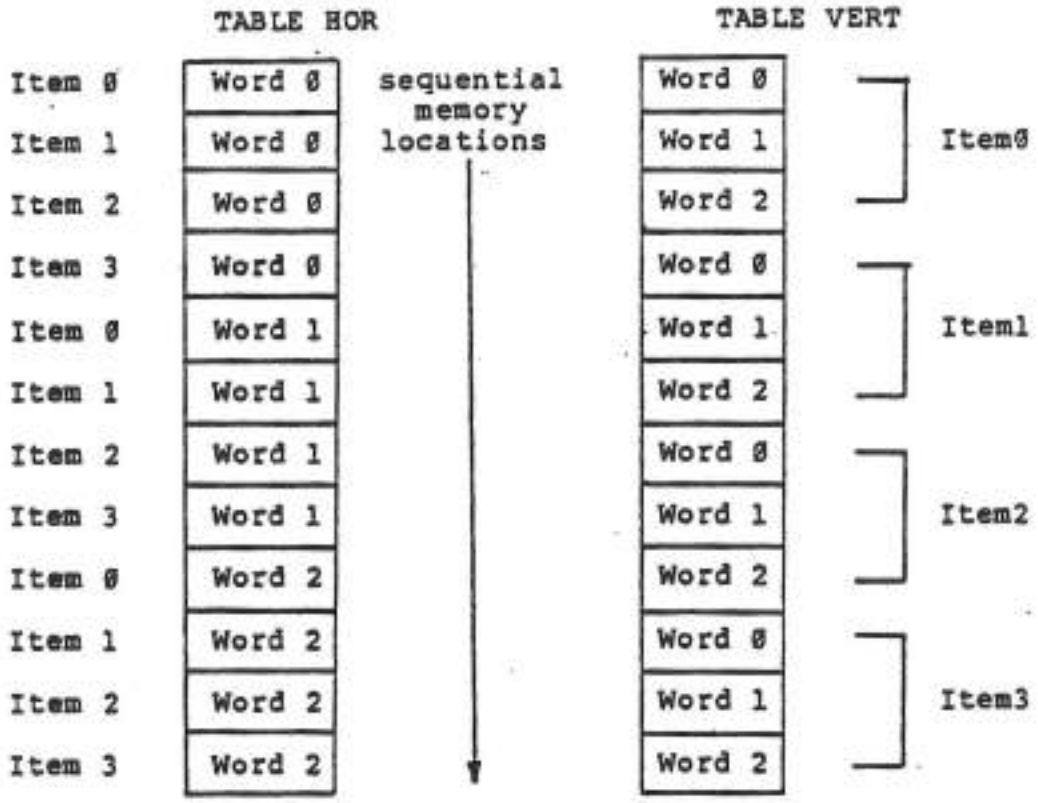
EXAMPLES

| Declaration | Remarks |
|---|---|
| TABLE HOR H 3 4 $ | HOR is the name given to a horizontal table with 4 items of 3 words each |
| TABLE VERT V 3 4 $ | VERT is a vertical table with 4 items of 3 words each |

All tables are allocated to blocks of sequential memory addresses in the computer. Items are numbered from Ø through the number of items -1 and the words of an item are also numbered from Ø through the number of words -1.

Horizontal tables are allocated so that the first word of the first item is followed by the first word of the second item and so on, until all first words are allocated. Then all second words, all third words, etc., are allocated until all words are assigned. See table HOR on the following page.

Vertical tables are allocated so that all words of the first item are followed by all words of the second item, etc., to the end. See table VERT on the following page.




When debugging programs, it is sometimes necessary to examine data as it appears in memory. To do so may require an understanding of how the data is allocated by the compiler. This is the reason for discussing the method the compiler uses in allocating memory segments to tables.

---

WORK AREA

1. Given a table of 3 items with 5 words each:
   a. how would the items be numbered?
   b. how would the words be numbered?

2. In what way is item allocation different in vertical and horizontal tables?

CORRECT ANSWERS

1. a. item 0,1,2
   b. word 0,1,2,3,4

2. vertical - all words of each item are allocated together (item 0, word 0,1,2, etc., then item 1, word 0,1,2, etc)
   horizontal - all first words are allocated, then all second words, etc. (item 0, word 0; item 1, word 0; item 2, word 0, etc.)

---

## 1. SUB-TABLES

The allocation of vertical and horizontal tables has very different results. Sub-tables of vertical tables constitute a discrete block in memory, while sub-tables of horizontal tables are composed of blocks that may be interspersed with other parts of the table.

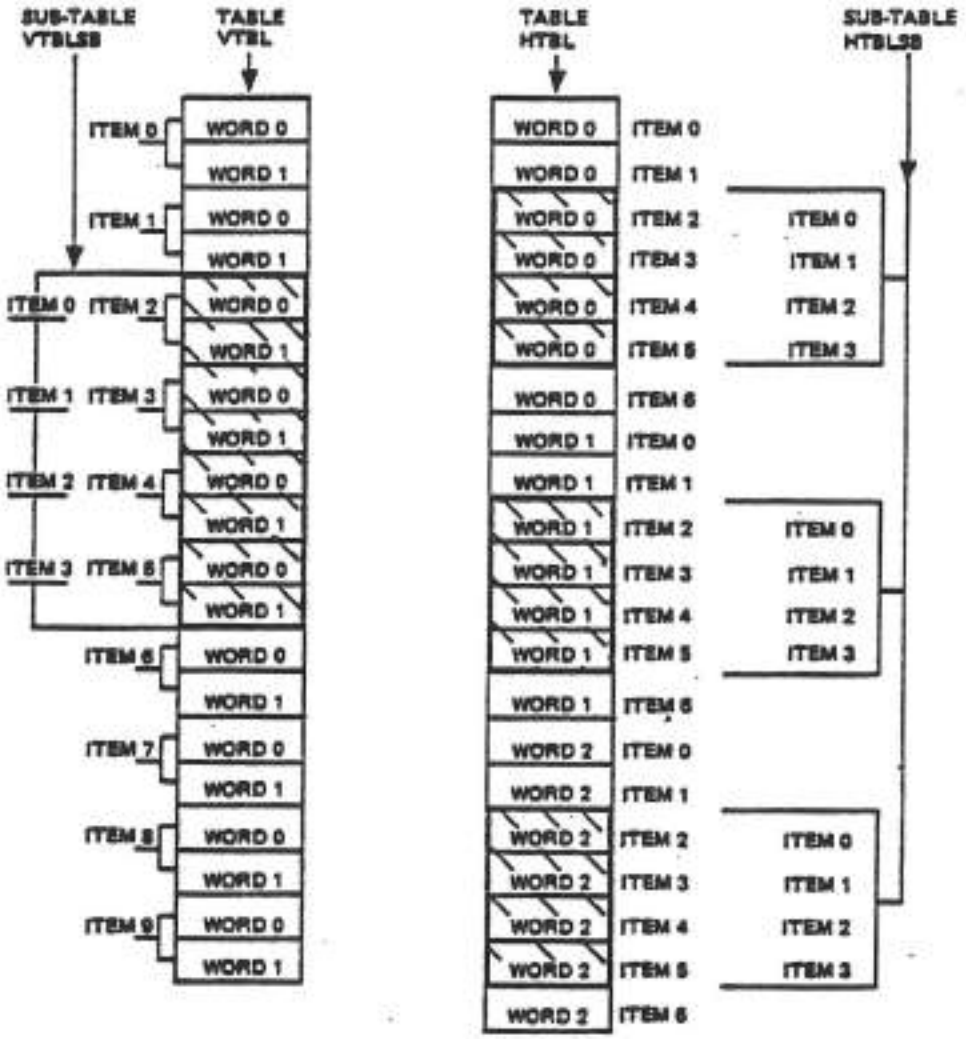
Please study the pictorial representations on the following page of the examples below.

EXAMPLES

```
TABLE VTBL V 2 10 $
SUB-TABLE VTBLSB 2 4 $
END-TABLE VTBL $

TABLE HTBL H 3 7 $
SUB-TABLE HTBLSB 2 4 $
END-TABLE HTBL $
```

Note that sub-table VTBLSB starts in item 2 of table VTBL and picks up 4 items (2,3,4,5).

Sub-table HTBLSB looks entirely different as it picks up 4 items each time it locates item 2.




---

WORK AREA

```
TABLE SIGNAL V 6 10 $
SUB-TABLE SIG1 3 6 $
```

1. Where does sub-table SIG1 begin?

2. How many items are in this sub-table?

CORRECT ANSWERS

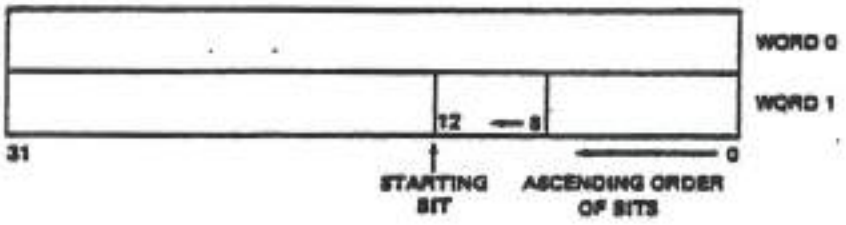
1. Item 3 of table SIGNAL
2. six

---

## 2. FIELDS OF USER-PACKED TABLES

When the programmer chooses to specify the number of words per item in a table, it becomes necessary to specify the actual physical location of any fields declared. This is done by expanding the field declaration to include the number of the word within the item that contains the starting bit of the field and the number of the bit position of that word in which it starts.

The starting bit is defined as the left-most bit of the field. The bits of the 32 bit computer word, however, are numbered from 0 through 31 beginning from the right. It is vital not to confuse the two numbering procedures. For example, an integer field of 5 bits beginning in word 1, bit position 12 (left-most bit), would occupy bits 12,11,10,9 and 8 of word 1.




FORMAT

| FIELD | name | type | starting word | starting bit | P | preset values(s) | $ |
|---|---|---|---|---|---|---|---|

Explanation:

starting word - number of the word within the item in which the left-most bit of the field is positioned
starting bit - the number of the bit position in the starting word that is the left-most bit of the field

EXAMPLE

In Chapter 4 a table called TEAM was defined. To demonstrate how user-packed fields are declared, a table TEAM1 with the same names and type fields will be defined.

| Declarations | Remarks |
|---|---|
| TABLE TEAM1 V 9 20 $ | TEAM1 is a user-packed table with 20 items of 9 words each. |

| Declarations | Remarks |
|---|---|
| FIELD NAME H 28 0 31 $ | NAME is a character type field, 28 characters in length, starting in word 0, bit position 31. A character type field that begins in one word and ends in another must start in bit position 31 of the initial word. |
| FIELD HEIGHT I 7 U 7 30 $ | HEIGHT is an integer type field of 7 bits, unsigned, starting in word 7, bit position 30. |
| FIELD GRADEPNT A 9 U 6 7 17 $ | GRADEPNT is a fixed-point type field of 9 bits, 6 of which are fractional, unsigned, starting in word 7, bit position 17. |
| FIELD WEIGHT I 9 U 7 8 $ | WEIGHT is an integer type field of 9 bits, unsigned, starting in word 7, bit position 8. |
| FIELD AGE I 7 U 8 30 $ | AGE is an integer type field of 7 bits, unsigned, starting in word 8, bit position 30. |
| FIELD YEAR I 3 U 8 18 $ | YEAR is an integer type field of 3 bits unsigned, starting in word 8, bit position 18. |
| FIELD RANK S '1STTEAM','2NDTEAM','JV' 7 19 $ | RANK is a status type field with 3 possible states, starting in word 7, bit position 19. |
| FIELD ELIGIBLE B 8 1 $ | ELIGIBLE is a Boolean type field in word 8, bit position 1. |
| FIELD FNCLAID B 8 8 $ | FNCLAID is a Boolean type field in word 8, bit position 8. |
| FIELD LASTWORD I 32 S 8 31 $ | LASTWORD is an integer type field of 32 bits, signed, starting in word 8, bit position 31. LASTWORD is inserted to illustrate that fields may overlap. |
| END-TABLE TEAM1 $ | |

---

WORK AREA

1. A 10 position integer field starting in word 1, bit 31, would occupy what bit positions?
2. Write definitions for the following:

   A vertical table called STATS having 365 items of 9 words each and fields DAY, integer, 11 unsigned bits, starting in bit position 15 of word 0; MONTH, integer, 5 unsigned bits, starting in bit position 4 of word 0; NEWHI, Boolean, in bit position 16 of word 0; NEWLO, Boolean, in bit position 24 of word 0; HIGH, LOW, PREVHI, PREVLO, floating-point, starting in bit position 31 of words 1, 3, 5 and 7, respectively.

CORRECT ANSWERS

1. bits 31 through 22 in word 1

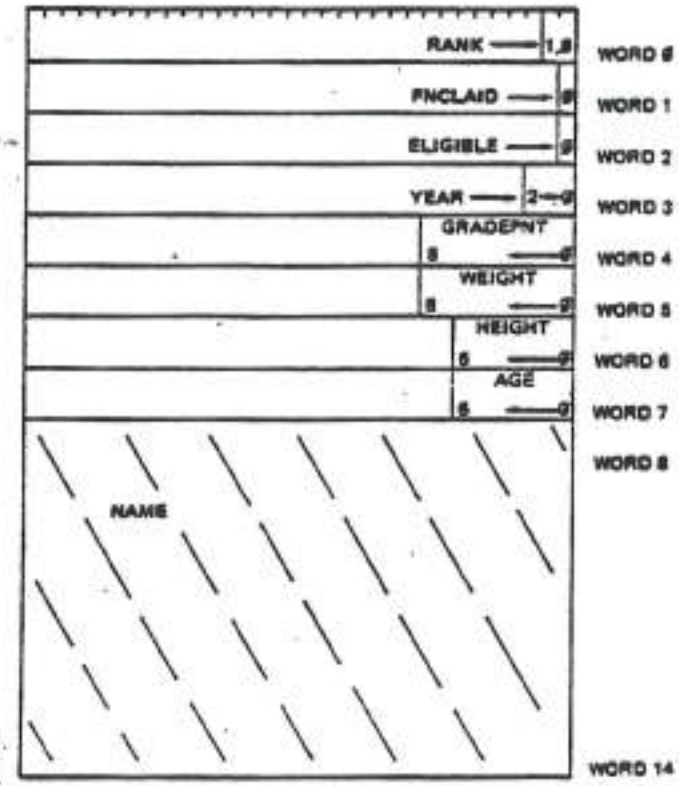
2.
```
TABLE TSTATS V 9 365 $
    FIELD DAY I 11 U 0 15 $
    FIELD MONTH I 5 U 0 4 $
    FIELD NEWHI B 0 16 $
    FIELD NEWLO B 0 24 $
    FIELD HIGH F 1 31 $
    FIELD LOW F 3 31 $
    FIELD PREVHI F 5 31 $
    FIELD PREVLO F 7 31 $
END-TABLE TSTATS $
```

---

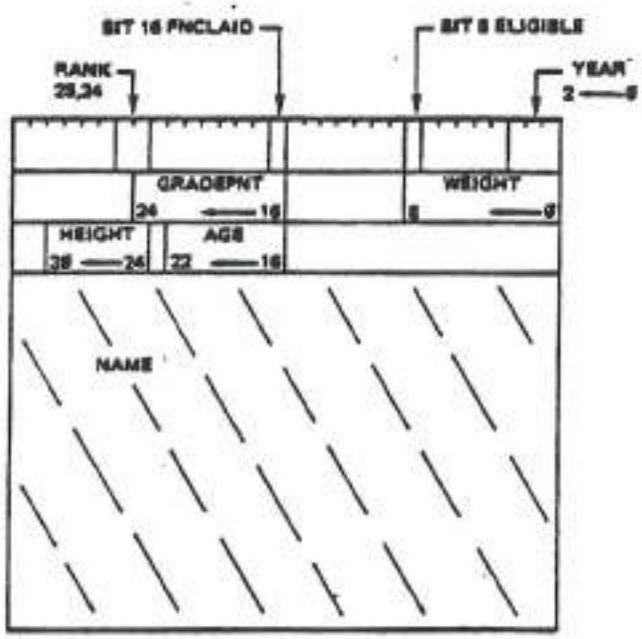
## 3. COMPARATIVE ALLOCATION OF DATA

The following illustrations show the allocation of fields in table TEAM (from Chapter 4) according to NONE, MEDIUM and DENSE allocation. Also, user-packed table TEAM1 is shown.

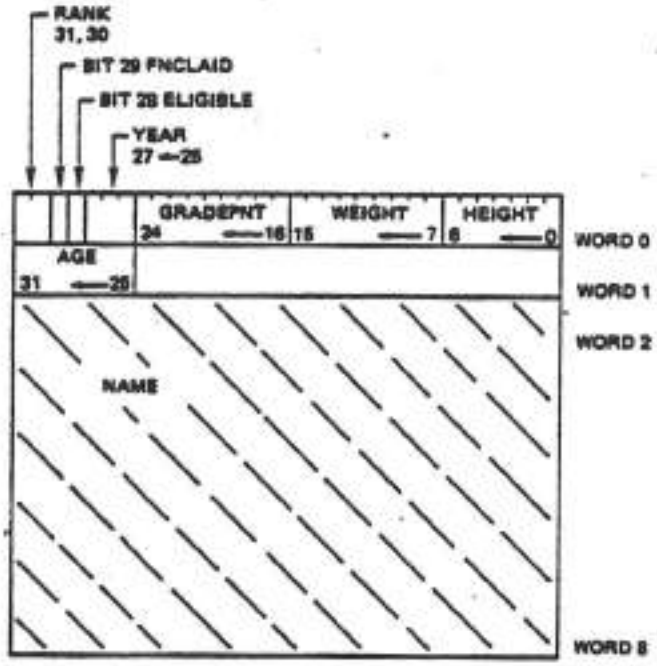
NONE



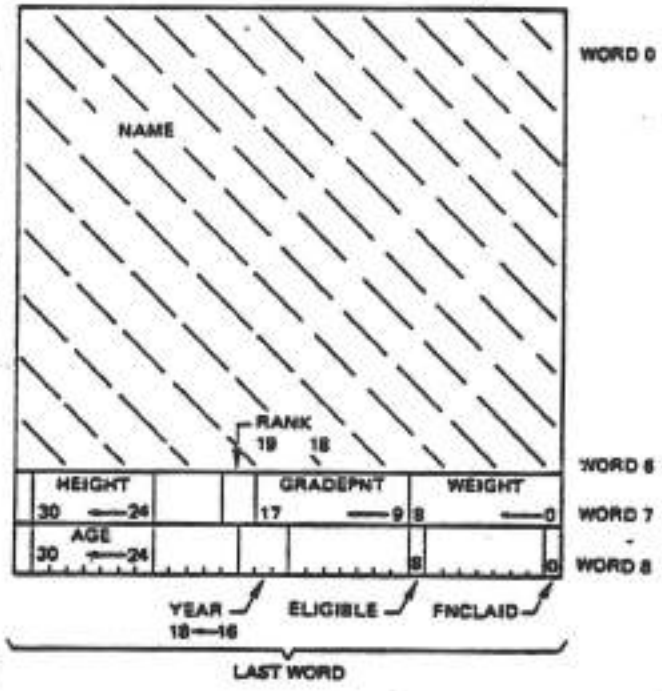

MEDIUM



DENSE




USER-PACKED




All bits need not be allocated and fields may overlap.

Dense packing and user packing required the same number of words, but by making use of byte boundaries the user packing allows more efficient accessing. The DENSE packing algorithm, by definition, does not allow such scattering of fields.

---

WORK AREA

1. FIELD HIGH I 9 U 3 8 $ Based on this statement:
   a. how many bits are needed to hold this field?
   b. which word of the table holds field HIGH?
   c. the field starts in which bit position?
   d. the field is contained within which bit positions?

2. A statement such as the one in problem 1 is written only when declaring what kind of table-block?

3. What "type" is specified for a:
   a. character field?
   b. Boolean field?
   c. numeric field with no fractions?
   d. status type field

CORRECT ANSWERS

1. a. 9
   b. word 3
   c. bit 8
   d. bits 8 through 0

   Starting and ending bits can be confusing since the starting bit to be specified in the statement is the left-most bit, whereas the numbering of any word starts at the right-most position and progresses to the left.

2. a user-packed table
3. a. H  c. I or A with zero fractional bits
   b. B  d. S

---

## B. ARRAYS

An array is a multi-dimensioned table. To relate this to the previously described tables, it might be said that those previously discussed were one-dimensional (consisting of one group of items with specified attributes). Arrays consist of multiple groups of items, each successive group of which contains all the others.

An array declaration specifies the name of the array, the subscripts to be used in accessing its items and the manner of allocating it to memory.

FORMAT

| TABLE | name | A | item allocation | subscript declaration | $ |
|---|---|---|---|---|---|

TABLE - keyword indicating a table declaration

name - the name of the array being declared

A - keyword indicating a multi-dimensioned table (array) is being declared

item allocation - specification of the way an item of the table is to be allocated. The specifications are the same as those for a one-dimensioned table, i.e. NONE, MEDIUM, DENSE, a type declaration, or number of words (user-packed)

Specifications:

1. An array must contain at least two (but no more than 7) subscript declarations.
2. The number of subscript declarations is the dimension of the table.
3. The number of words allocated for an array is the product of the number of words per item and the values of the subscript declarations.

4. An array is always of vertical table type. An example of a three-dimensioned array may be found below.
5. The first item of an array is the item corresponding to all subscripts equal to zero. The items of an array are allocated to sequential memory locations with the first subscript varying most rapidly, the second subscript next most rapidly, etc.
6. Like-tables and sub-tables are not allowed in arrays.

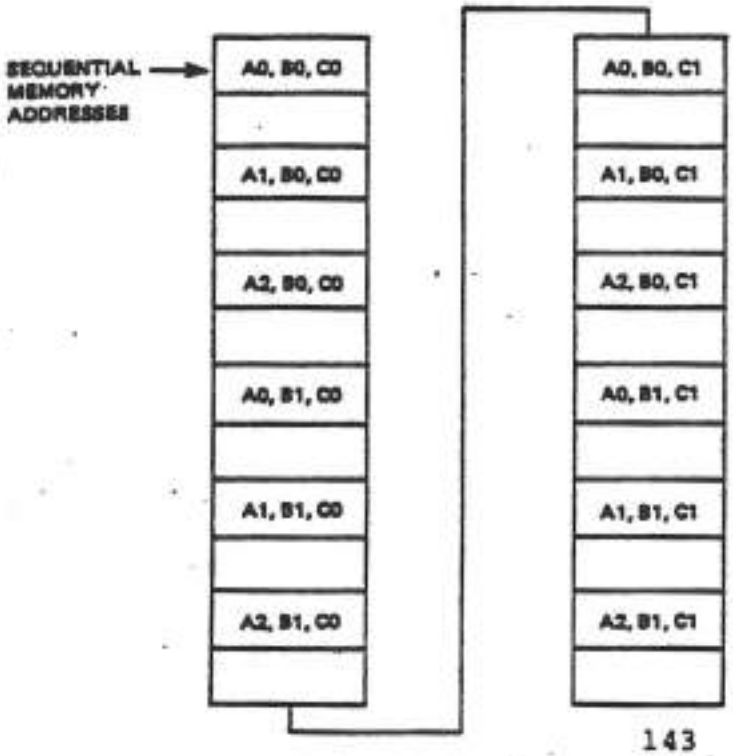
EXAMPLE

| Declaration | Remarks |
|---|---|
| TABLE ANARRAY A (F) 3,2,2 $<br>END-TABLE ANARRAY $ | ANARRAY is a three dimensioned array of floating-point type items. The first dimension consists of three items, the second dimension of two items and the third dimension of two items. |

Referencing an array:
An array is referenced by the array name and (in parentheses) an item index for each dimension and (if needed) a word or field specification.

The items of an array (like the items of a table) are numbered from zero through the number of items minus one. For instance, in the three- dimensional array below, the valid item subscripts are (left-to-right) 0 through 2, 0 through 1 and 0 through 1.

Pictorial Representation of Table ANARRAY




Note: If the three dimensions of the array are called A,B and C with values of 3, 2, 2 respectively, then A1, B1, C0 represents the second item of A in the second item of B in the first item of C. A reference to this item in a CMS-2 statement would appear as:

```
ANARRAY(1,1,0).
```

Using index names to specify items, the reference would appear as:

```
ANARRAY(INDEXA,INDEXB,
 INDEXC).
```

## C. INDIRECT TABLES

An indirect table is a table whose name is used as a stand-in for other addressable data. The attributes of the indirect table may be regarded as a descriptive template that will (in effect) be "laid over" the specified addressable data.

The compiler allocates only one computer word for an indirect table. This word, after initialization by a sub-program, contains no table data, but instead the information needed to access the specified addressable data.

An indirect table is declared by inserting the keyword INDIRECT between the item allocation and the table subscript declaration in the table declaration.

FORMAT

| TABLE | name | type | item allocation | INDIRECT | table subscript declaration | $ |
|---|---|---|---|---|---|---|

EXAMPLE

| Declaration | Remarks |
|---|---|
| TABLE HOLDADR V (H8) INDIRECT 64 $<br>END-TABLE HOLDADR $ | HOLDADR is declared as an indirect table. When the address of a data area is assigned to HOLDADR, that area is looked at as if it were a vertical table consisting of 64 character typed items of 8 characters each. |

### 1. USING INDIRECT TABLES

Before an indirect table may be referenced, it must be assigned a memory address. This is done by using a CORAD intrinsic (part-of-the-language) function reference.

FORMAT

| .... CORAD(addressable name).... |
|---|

Explanation:

CORAD - keyword indicating a memory address function reference.

addressable name - any name that is assigned to a memory location and can be referenced in a CMS-2 program.

A CORAD function reference returns a value that is the memory address of name specified as the actual input parameters. The only method of assigning an address to an indirect table is to use a CORAD set statement.

FORMAT

| | | | | |
|---|---|---|---|---|
| SET | CORAD(indirect table name) | TO | address expression | $ |

Explanation:

address expression - an expression whose value is an address in computer memory. May include a CORAD function reference.

The SET statement above means: equate the starting address of the specified indirect table to the value of the address expression.

EXAMPLE

```
TABLE HOLDADR V (H8) INDIRECT 64 $
END-TABLE HOLDADR $
TABLE BUFFIN V 1 128 $
END-TABLE BUFFIN $
VRBL HOLDX I 8 U $

SET CORAD(HOLDADR) TO CORAD(BUFFIN) $
```

Any references to HOLDADR in statements executed after the above statement will cause the data beginning at the memory address of BUFFIN to be accessed. The descriptive attributes of HOLDADR (not those of BUFFIN) will apply. This condition continues until the execution of another CORAD function reference assigns a different address to HOLDADR.

```
SET HOLDADR(HOLDX) TO H(BEGINCOM) $
```

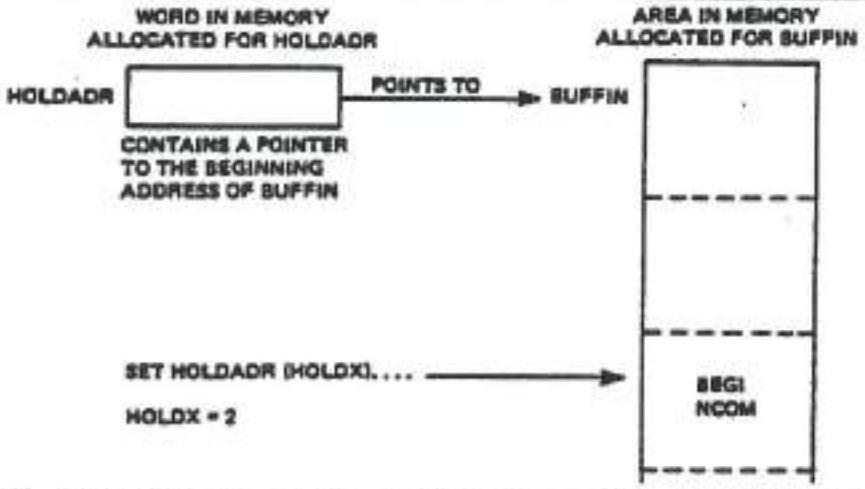
A pictorial representation of the SET statements is shown on the following page.

---

WORK AREA

1. Write table-block declarations for an indirect table TASKCOST containing eleven floating-point items.

1. 
```
TABLE TASKCOST V (F) INDIRECT 11 $
END-TABLE TASKCOST $
```




Chapter 6 introduced and discussed two procedures, AVRG and FINDDIFF. The two procedures are rewritten on the following page to show one way of using indirect tables. The Chapter 6 versions are shown here as a source for comparison.

```
TABLE WA H (A 32 S 0) 100 $
END-TABLE WA $
VRBL(LX,NUMS) I 8 U $
VRBL RESULT A 32 S 0 $

PROCEDURE AVRG INPUT WA, NUMS OUTPUT RESULTS $
    SET RESULT TO 0 $
VARY LX FROM NUMS -1 THRU 0 BY -1 $
    SET RESULT TO RESULT + WA(LX) $
END $
    SET RESULT TO RESULT / NUMS $
END-PROC AVRG $

TABLE RUFDATA 1 H(A 20 U 0) 50 $
    LIKE-TABLE RUFDATA2 $
    LIKE-TABLE DIFFS $
END-TABLE RUFDATA1 $
VRBL AVGDIFF A 32 S 0 $

PROCEDURE FINDDIFF $
LOC-INDEX DIFX $
VARY DIFX FROM 49 THRU 0 BY -1 $
    SET DIFFS(DIFX) TO RUFDATA1(DIFX)-RUFDATA2(DIFX) $
END $
AVRG INPUT DIFFS,50 OUTPUT AVGDIFF $
END-PROC FINDDIFF $
```

```
01          TABLE WA H (A 32 S 0) INDIRECT 100 $
            END-TABLE WA $
            TABLE RUFDATA1 H (A 20 U 0) 50 $
               LIKE-TABLE RUFDATA2 $
               LIKE-TABLE DIFFS $
            END-TABLE RUFDATA1 $
            VRBL (RESULT,AVGDIFF) A 32 S 0 $
            VRBL NUMS I 7 U $
```

Table WA is re-defined as an indirect table. Only one word in memory will be reserved instead of one hundred.

```
02          PROCEDURE AVRG INPUT CORAD(WA),NUMS
               OUTPUT RESULT $
            LOC-INDEX LX $
            SET RESULT TO 0 $
            VARY LX FROM NUMS-1 THRU 0 BY -1 $
               SET RESULT TO RESULT+WA(LX) $
            END $
               SET RESULT TO RESULT/NUMS $
            END-PROC AVG $

            PROCEDURE FINDDIFF $
            LOC-INDEX DIFX $
            VARY DIFX FROM 49 THRU 0 BY -1 $
               SET DIFFS(DIFX) TO RUFDATA1(DIFX)-
               RUFDATA2(DIFX) $
            END $
```

The first formal input parameter of AVRG is changed from a table to the memory address of the indirect table.

```
03          AVRG INPUT CORAD(DIFFS),50 OUTPUT
               AVGDIFF $
            END-PROC FINDDIFF $
```

To match the formal input parameter, the actual input parameter is changed to the memory address of the table.

The original versions resulted in the fifty values specified in the procedure call being transferred from DIFFS to WA before execution of AVRG.

In the new versions (on this page), the memory address of DIFFS is assigned to WA and when AVRG is executed, the values are accessed in place.

## D. VARY WITHIN STATEMENT

The VARY statement was discussed in Chapter 5 in connection with loop blocks. VARY WITHIN is a special form of VARY in which the compiler determines the termination value of the loop index based upon the number of items in the specified table. The VARY WITHIN feature is only available for the AN/UYK-7 target computer.

FORMAT

| VARY | loop index | FROM | initial value | WITHIN | table name | BY | change value | $ |
|---|---|---|---|---|---|---|---|---|

Explanation:

WITHIN - (optional) keyword indicating that a table name follows. The number of items in the table will be used to determine the termination value and possibly the initial value of the loop index.

Table name - (optional) the name of a table for which the loop index must be a valid subscript.

For forward varying, the termination value is the number of table items -1. For backward varying, the termination value is zero. If FROM is omitted, the compiler will also supply the initial loop index value, zero for forward varying, number of items -1 for backward varying.

EXAMPLE

```
TABLE VEHICLES V MEDIUM 90 $
   FIELD CODENAME H 12 $
   FIELD MILES A 20 U 5 $
*  FIELD GALLONS A 19 U 5 $
   FIELD MPG A 12 U 5 $
END-TABLE VEHICLES $
VRBL PRX I 7 U $


VARY PRX WITHIN VEHICLES $
END $
```

The compiler will automatically generate code to vary loop index PRX from zero through the number of items in the table -1. In the above example, from 0 through 89 - a total of 90 times.

```
VARY PRX WITHIN VEHICLES BY -1 $
END $
```

The compiler will generate code to vary backward from the number of items -1 through zero.

## E. FIND STATEMENT

A FIND statement specifies a search of a table for an item satisfying a given condition and is followed by one or two statements to be executed depending on the result of the search. FIND statements are only valid for the AN/UYK-7 computer.

FORMAT

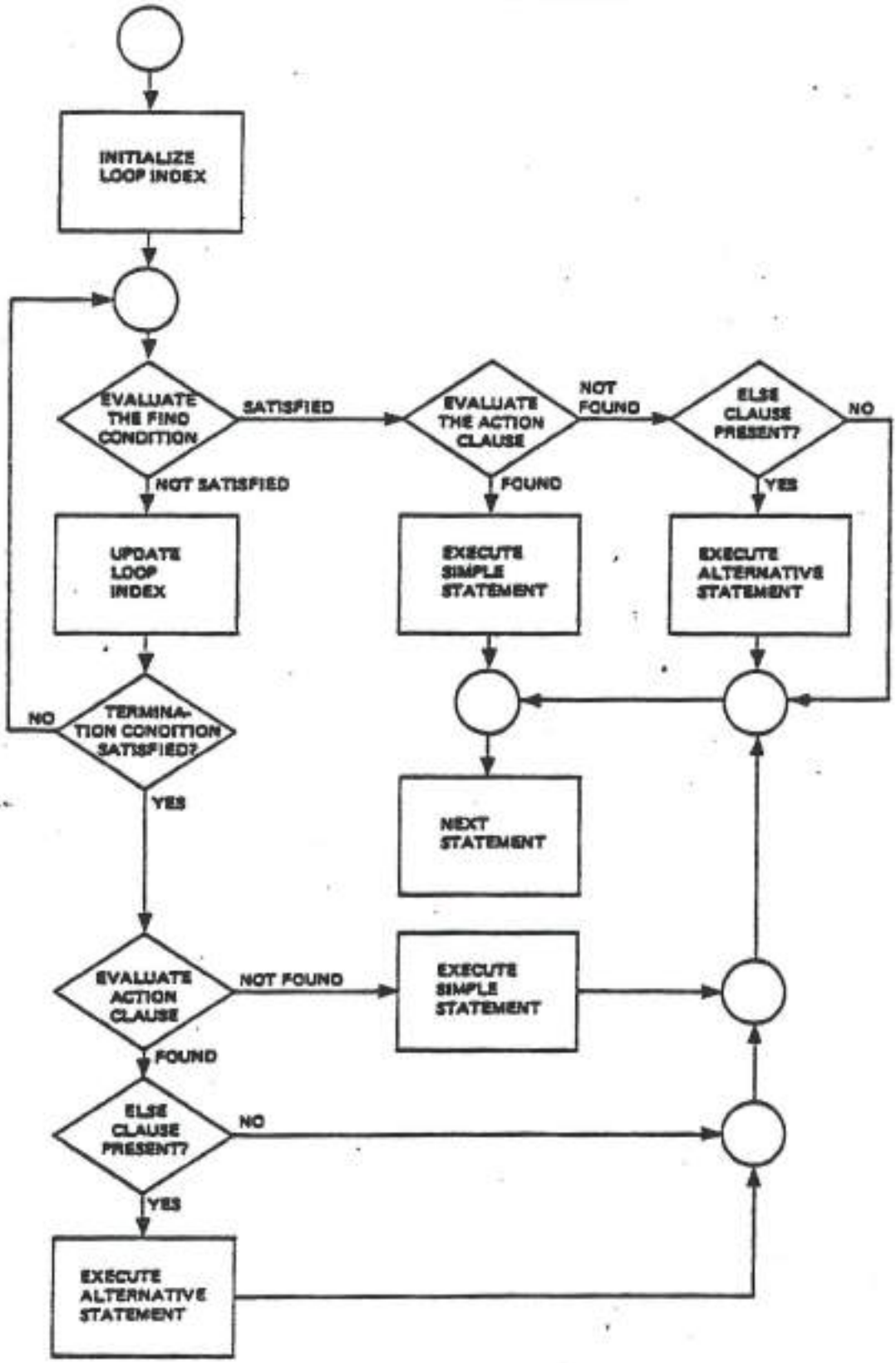
```
FIND    find  VARYING   index clause $
     condition
IF DATA FOUND THEN   simple statement $  else clause $
          or
IF DATA NOTFOUND THEN   simple statement $  else clause $
```

Explanation:

| | |
|---|---|
| FIND | - keyword indicating a FIND statement |
| find<br>condition | - expression specifying a condition on an item of the table that must be met for the search to be satisfied |
| VARYING | - (optional) keyword indicating that an index clause for the table index follows |
| index clause | - (optional) specification of constraints on the index of the table as the search is performed. The keywords FROM, BY, THRU and WITHIN have the same uses here as described for the VARY statement. WHILE and UNTIL may not be used in a find index clause. |
| IF DATA FOUND THEN | - keywords indicating that the following statement is to be executed if a table item satisfying the find condition is found |
| IF DATA NOTFOUND THEN | - keywords indicating that the following statement is to be executed if no table item satisfying the find condition is found |
| simple statement | - a statement to be executed depending on the result of the table search |
| else clause | - (optional) specification of an alternative statement to be executed if the result of the table search is not that specified by the IF DATA clause |

A flowchart of the FIND evaluation is shown on the following page, then a series of statements specifying the steps involved in the execution of a FIND statement.

FIND Evaluation



Evaluation of a FIND statement takes the following steps: (Follow each step through the flowchart.)

1. Initialize the loop index.

2. Evaluate the FIND condition.
   a. If the condition is not satisfied, go to step 3.
   b. If the condition is satisfied, further execution depends on the form of the action (IF DATA) clause.
      (1) If the action clause is IF DATA FOUND: execute the simple statement, completing the FIND statement execution.
      (2) If the action clause is IF DATA NOT FOUND and no ELSE clause is present: FIND statement execution is complete with no further action taken.
      (3) If the action clause is IF DATA NOT FOUND and an ELSE clause is present: execute the alternative statement, completing FIND statement execution.

3. Update loop index and test loop termination condition.
   a. If the termination condition is not met, go to step 2.
   b. If the termination condition is met, further execution depends of the form of the action clause.
      (1) If the action clause is IF DATA FOUND and no ELSE clause is present: FIND execution is complete with no further action.
      (2) If the action clause is IF DATA FOUND and an ELSE clause is present: execute the alternative statement, completing the FIND statement execution.
      (3) If the action clause is IF DATA NOTFOUND: execute the simple statement, completing FIND statement execution.

---

WORK AREA

Are the following statements true or false?

1. The VARY WITHIN allows the programmer to vary forward only.
2. The WITHIN clause replaces the THRU clause in the format of a VARY statement.
3. If WITHIN is used, a table name must be specified.
4. A FIND statement is used to locate an item of data in a procedure.
5. A FIND statement causes a search to be made.
6. A FIND statement is concluded by an END declaration.

CORRECT ANSWERS

1. False
2. True
3. True
4. False
5. True
6. False

---

## 1. FIND Rules

a. The left comparand of the FIND conditional expression must be a subscripted reference, and the first subscript must be a numeric type variable (including indexes). This variable becomes the loop index of the loop implied by the FIND statement.

b. If the optional varying clause is present, the index clause must specify the implied loop index. If the termination clause is omitted, a termination clause of WITHIN the table is implied.

c. If the optional varying clause is omitted, the FIND clause implies a loop whose index is varied FROM 0 BY 1 WITHIN the table named in the first comparand of the statement.

EXAMPLES

```
        TABLE VEHICLES V MEDIUM 90 $
        FIELD CODENAME H 12 $
        FIELD MILES A 20 U 5 $
        FIELD GALLONS A 19 U 5 $
        FIELD MPG A 12 U 5 $
        ITEM-AREA CAR $
        LIKE-TABLE WORKAREA $
        END-TABLE VEHICLES $
        VRBL (PRX,PRX1) I 7 U $

FIND VEHICLES(PRX,CODENAME) EQ H(FORDMOD1SS5G) $
IF DATA FOUND THEN
        SET CAR TO VEHICLES(PRX) $
ELSE
        GOTO MODELERR $
```

Comments:

a. If a find is made, since the action clause is IF DATA FOUND, the SET statement is executed. This completes FIND statement execution and the sub-program continues with the statement following the ELSE clause.

b. If a find is not made, since the action clause is IF DATA FOUND and an ELSE clause is provided, the ELSE clause statement is executed, completing FIND statement execution.

2. Backward Search

Backward searching is possible with a FIND statement. The previous example may be written:

EXAMPLE

```
FIND VEHICLES(PRX,CODENAME) EQ H(FORDMOD1SS5G)
        VARYING PRX BY -1 $
```

The compiler will automatically initialize the implied loop index to 89 and search backward through zero.

3. Use of a Label

By placing a label on the FIND statement, it is possible to resume the table search after each find until loop termination is satisfied, as shown in the example below:

EXAMPLE

```
            SET PRX1 TO 0 $
FINMILES.FIND VEHICLES(PRX,MILES) GT 5000 $
            IF DATA FOUND THEN BEGIN $
               SET WORKAREA(PRX1) TO VEHICLES(PRX) $
               SET PRX1 TO PRX1+1 $
               RESUME FINMILES $
               END $


            continue program
```

When a find is made, the action clause will be executed and the RESUME phrase will return control to the loop index update, which continues the search with the next item of the table. When the loop index termination condition is satisfied, execution of the FIND statement is complete and control passes to the statement following the END statement. In order to resume a FIND statement, the statement must be labelled and that name must appear in the RESUME statement.

Use the following space for notes or comments.

WORK AREA

Given (1) table VEHICLES and (2) a procedure with the following definition:

```
PROCEDURE PUTOUT INPUT BUFF24 $
   where BUFF24 is a 24-word item-area.
```

Write a procedure GETFAILS that will search table VEHICLES, using a FIND statement, for each vehicle that failed to meet a standard of 22.5 miles per gallon. When a find is made, call procedure PUTOUT to print the item and then resume the search. When the entire table has been searched, the procedure is complete.

```
TABLE VEHICLES V MEDIUM 90 $
   FIELD CODENAME H 12 $
   FIELD MILES A 20 U 5 $
   FIELD GALLONS A 19 U 5 $
   FIELD MPG A 12 U 5 $
   ITEM-AREA CAR $
   LIKE-TABLE WORKAREA $
END-TABLE VEHICLES $
```

CORRECT ANSWER

```
01          PROCEDURE GETFAILS $
02          LOC-INDEX LX1 $
03  FAILS.  FIND VEHICLES(LX1,MPG) LT 22.5 VARYING LX1 BY -1 $
04          IF DATA FOUND THEN BEGIN $
05             PUTOUT INPUT VEHICLES(LX1) $
06             RESUME FAILS $
07             END $
08          END-PROC GETFAILS $
```

Line 03 could also have been written: FIND VEHICLES(LX1,MPG) LT 22.5 $

---

## Chapter 7 Summary

User packed tables are explained, including sub-tables, fields of user packed tables and the allocation of data in memory for both compiler packed and user packed tables. Arrays and indirect tables are also covered.

Table searches using the FIND statement are discussed in some detail, including FIND rules backward searches and the use of labels. The VARY WITHIN statement is also touched upon.

## CHAPTER 7 PROBLEMS

1. Write declarations for the following data units

   a. Table LINKDATA, horizontal, indirect, having 15 items of 3 words each and the following fields:

      (1) GRIDA - fixed-point, signed, 12 bits of which 7 are fractional, beginning in bit 15 of word 0

      (2) GRIDB - same as GRIDA, but beginning in bit 31 of word 0

      (3) DBLGD - integer, signed, all of word 0

      (4) STATBITS - integer, 18 bits, unsigned, beginning in bit 23 of word 1

      (5) CHAN - integer, unsigned, 6 bits, beginning in bit 25 of word 2

      (6) EFCT - integer, unsigned, 15 bits, beginning in bit 14 of word 2

2. VRBL LNKX I 4 U $ is to be used as the index in the following problem solution:

   In table LINKDATA (problem 1) it is desired that field CHAN be set to 15 in all items in which GRIDA and GRIDB are identical.

   a. Write a block of statements to accomplish the above, using a vary within, varying forward.
   b. Write a block of statements to accomplish the above, using a find statement called GRIDEQ, varying backward, and using an if data found with a begin block.

3. Given a table INLINK whose definition is compatible with LINKDATA, write a statement to assign the address of INLINK to LINKDATA.

## CORRECT ANSWERS

1.
```
TABLE LINKDATA H 3 INDIRECT 15 $
   FIELD GRIDA A 12 S 7 0 15 $
   FIELD GRIDB A 12 S 7 0 31 $
   FIELD DBLGD I 32 S 0 31 $
   FIELD STATBITS I 18 U 1 23 $
   FIELD CHAN I 6 U 2 25 $
   FIELD EFCT I 15 U 2 14 $
END-TABLE LINKDATA $
```

2.
```
Solution a.
     VARY LNKX WITHIN LINKDATA $
     IF LINKDATA(LNKX,GRIDA) EQ LINKDATA(LNKX,GRIDB)
        THEN SET LINKDATA(LNKX,CHAN) TO 15 $
     END $

Solution b.
GRIDEQ. FIND LINKDATA(LNKX,GRIDA) EQ LINKDATA
        (LNKX,GRIDB) VARYING LNKX WITHIN
        LINKDATA BY -1 $
        IF DATA FOUND THEN BEGIN $
           SET LINKDATA(LNKX,CHAN) TO 15 $
           RESUME GRIDEQ $
           END $
```

3.
```
SET CORAD(LINKDATA) TO CORAD(INLINK) $
```

# CHAPTER 8

# ORGANIZATION OF A CMS-2 SYSTEM

# Organization of a CMS-2 System

## A. INTRODUCTION

Every programming language has characteristics that encourage or require the user to organize his source statements according to some pattern. How these characteristics will be used depends (in part) upon the nature of the program.

Many small or medium sized programs, developed and maintained by a single programmer or a small group of programmers, may be conveniently organized as a single unit. However, computer programs increasingly tend to become very large, complex systems, developed and maintained by a large number of programmers. It then becomes very desirable that the overall program be organized in several units that can be worked upon independently and then joined to form the final product.

## B. SYSTEM BLOCKS

CMS-2 programs are "block structured". That is, source statements are grouped into units called blocks. These units may be combined in various ways to form larger blocks. All of the source statements presented to a CMS-2 compiler at one time comprise a unit called a system block. This system block may contain all the elements of a program or may consist of a portion of the whole, to be joined to other such portions later.

The source statements presented together for a compile (the system block) are also referred to as a compilation module. This is the smallest unit acceptable to a CMS-2 compiler.

Within the system block, source language statements are organized into system elements (consisting of system data blocks and system procedures) and header blocks (consisting of declarations, parameters and direction that control the compiling process).

The format of a system block is as follows:

FORMAT

```
system name   SYSTEM   key specification $

          system element(s)

END-SYSTEM  System name $
```

Explanation:

| | |
|---|---|
| system name | - name of the system block being declared |
| SYSTEM | - keyword indicating the beginning of a system block |
| key specification | - (optional) file identifier used in CMS-2 library and loader operations |
| system element(s) | - see following page |
| END-SYSTEM | - keyword indicating the end of a system block |
| system name | - must be the same as the name appearing in the system declaration |

EXAMPLE

```
GENTEST          SYSTEM $

                 END-SYSTEM GENTEST $
```

The name, although appearing in a position usually associated with labels, is not a label. No period is used. The same is true for the names of headers, system elements and loc-dd's. The distinction is made because labels are identifiers for statements to which program control may be transferred by a GOTO, while these declarations are informational only and may not be duplicated.

---

WORK AREA

1. What is meant by the statement --- CMS-2 programs are "block-structured"?

2. What is another name for a system block?

3. What are the keywords for a system block?

CORRECT ANSWERS

1. The source statements are organized into units called blocks

2. Compilation module (the smallest unit acceptable to a CMS-2 compiler).

3. SYSTEM
   END-SYSTEM

---

## C. SYSTEM ELEMENTS

The basic units that are used to form a system block are called system elements. The two types of system elements are system data blocks (also called system data designs) and system procedure blocks.

### 1. SYSTEM DATA BLOCKS

System data blocks contain declarations of data that may be referenced by any subsequent system element in the system. A system data block is the appropriate place to define data that must be used by sub-programs in different system elements.

FORMAT

```
system data block name   SYS-DD    key specification   $
                         data sentence(s)
                         END-SYS-DD   system data block name  $
```

Explanation:

system data block name - name of the system data block being declared

SYS-DD - keyword indicating the beginning of a system data block

data sentence(s) - (optional) declaration of a data unit that can be referenced by a CMS-2 program

END-SYS-DD - keyword indicating the end of a system data block. The name that appears in the system data block declaration must appear in its end declaration.

EXAMPLE

```
GENDD    SYS-DD $
         VRBL (IND1,IND2,IND3) I 32 S $
         END-SYS-DD GENDD $
```

## 2. SYSTEM PROCEDURE BLOCKS

A system procedure block consists of one or more subprograms (functions or procedures) and their associated local data blocks. A typical system procedure may consist of a group of subprograms that together accomplish a specific task. For instance, a system procedure within the compiler might contain sub-programs to examine a line of source input for syntax errors.

FORMAT

```
system procedure    SYS-PROC   key specification $
  block name
                    system procedure sentence(s)
                    END-SYS-PROC  system procedure $
                                    block name
```

Explanation:

| | |
|---|---|
| system procedure block name | - name of the system procedure block being declared |
| SYS-PROC | - keyword indicating the beginning of a system procedure block |
| system procedure sentence | - (optional) definition of a sub-program (function or procedure) or a local data block |
| END-SYS-PROC | - keyword indicating the end of a system procedure block. The name that appears in the system procedure declaration must appear in its end declaration. |

EXAMPLE

```
GENSYSP  SYS-PROC $

         END-SYS-PROC GENSYSP $
```

---

WORK AREA

1. What are the keywords for system data blocks?
2. What are the keywords for system procedure blocks?
3. In what way must the name invented for the initial declaration be repeated?

CORRECT ANSWERS

1. SYS-DD
   END-SYS-DD

2. SYS-PROC
   END-SYS-PROC

3. repeated in the END-declaration

---

## 3. LOCAL DATA BLOCKS

Data that is needed only by the sub-programs of a particular system procedure are defined in local data blocks within the system procedure. One example might be tables or variables to hold intermediated results of sub-program operations.

A local data block must be wholly contained within a system procedure.

FORMAT

```
name     LOC-DD $
         local data sentence(s)
         END-LOC-DD   name $
```

Explanation:

name - (optional) name of the local data block being declared

LOC-DD - keyword indicating the beginning of a local data block

local data sentence - (optional) declaration of a data unit to be referenced by sub-programs of this system element

END-LOC-DD - keyword indicating the end of a local data block. If a name appears in the local data design declaration the same name must appear in its end declaration.

EXAMPLE

```
GENSYSP   SYS-PROC $
LDAT      LOC-DD $
          =
          =
          END-LOC-DD LDAT $
            =
            =
          END-SYS-PROC GENSYSP $
```

## D. HEADER BLOCKS

Header blocks provide information for the compiler. Header blocks may specify:

(1) parameters for the system block being compiled.
(2) the legality or illegality of certain source statements.
(3) processing directives that govern the compiler's interpretation of many CMS-2 operations.
(4) the number and kind of compiler outputs desired.
(5) activation of specialized hardware and software processing procedures.

### 1. MAJOR HEADERS

A system block must begin with and may contain only one major header block. The major header sentences contained in it provide information that applies to compilation of the entire system block. The major header is defined as everything from the system declaration to the first end-head declaration. Use of header declarations within these delimiting declarations is optional. However, after the first end-head, each header declaration must have a matching end-head.

FORMAT

```
major header    HEAD   key specification $
  name
                major header sentence(s)
                END-HEAD    major header $
                              name
```

Explanation:

major header name - (optional) name of the major header block
HEAD - (optional) keyword indicating a header block
major header sentence - (optional) a declaration that will affect compilation of the system block
END-HEAD - keyword indicating the end of a header block. If a name appears in a preceding header declaration, that name must appear in the end declaration.

---

WORK AREA

1. What are the keywords for a:
   (a) local data block?
   (b) major header block?
2. How many major headers may there be in a system block?

CORRECT ANSWERS

1. (a) LOC-DD
   END-LOC-DD
   (b) HEAD
   END-HEAD

2. only one

---

## 2. MINOR HEADERS

Any header sentence encountered after the first end-head declaration is considered part of a minor header. The information contained in minor headers applies only to compilation of the immediately following system element (sys-dd or sys-proc).

FORMAT

```
minor header name        HEAD $
                  minor header sentence(s)
                  END-HEAD  minor header name $
```

Explanation:

minor header name - (optional) name of the minor header block. If a name appears in the header declaration, it must appear in the end declaration.

HEAD - (optional) keyword indicating a header block

minor header sentence - (optional) a declaration that will affect compilation of the following system element (sys-dd or sys-proc) only

END-HEAD - keyword indicating the end of a header block

EXAMPLE

| Declarations | | Remarks |
|---|---|---|
| | HEAD $ | The equals statement will apply to all appearances of NUMWDS in sys-proc TRBL. When the end-sys-proc is encountered, the equals no longer applies. |
| NUMWDS | EQUALS 10 $ | |
| | END-HEAD $ | |
| TRBL | SYS-PROC $ | |
| | = | |
| | = | |
| | END-SYS-PROC TRBL $ | |

## 3. SYSTEM INDEXES

Local indexes were introduced in chapter six along with a brief discussion of machine index registers. Just as local index declarations specified variables whose values were to be held in index registers (if available) during the execution of a sub-program, a system index declaration specifies the names of variables whose values are to be held in index registers during the execution of the entire system block. A system index declaration also specifies the index register number to which the system index is to be assigned.

An example of system index use is a variable that is to be used by a number of system elements as a pointer (item index) to a common table. Having the value in a register may result in more efficient code.

Unlike local indexes, system indexes are limited in number by the number of index registers available for a given target computer. On the AN/UYK-7, the available registers are 1 through 5. Attempts to define additional system indexes will result in error messages and an undefined data unit.

FORMAT

```
SYS-INDEX      register number    system index name  $
```

Explanation:

SYS-INDEX - keyword indicating a system index declaration
register number - a numeric constant value specifying the register to hold the value of the specified system index
system index name - name of the system index being declared

System index declarations must appear in the major header.

EXAMPLE

```
SYS-INDEX 1 CTSX $
```

CTSX is a system index whose value will be held in index register 1 during program execution.

---

WORK AREA

1. What are the system elements?

2. If a system index is used, where must it appear in the system block?

3. What is the meaning of the following declaration?

```
SYS-INDEX 4 DBL S
```

CORRECT ANSWERS

1. system data blocks and system procedures

2. in the major header

3. the value of DBL will be held in index register 4 during execution of the program

---

When deciding whether or not to use system indexes, remember the following:

(1) An index register assigned as a system index is no longer available for use as a local index.

(2) An index register used as a system index may not be used by the compiler for transient values.

The above restrictions mean that heavy use of system indexes (using all 5 on an AN/UYK-7, for instance) may result in less efficient code by depriving the compiler of available registers and causing excessive use of temporary memory locations.

## 4. OPTION SPECIFICATIONS

Options specifications allow the programmer to select outputs from the compiler, such as listings and object code. A wide variety of listings are included, such as source, cross references of all names, and symbol analysis which lists all data units with their descriptions and bit-allocation (when applicable). Outputs, such as loadable object code and library files may be selected. Also certain specific error detection capabilities may be requested (an analysis of source for violations of structured programming rules, for example).

The options specification is also used to indicate the target computer for which code is to be generated.

The options specifications must immediately follow the system declaration, and the first options specification must be the target computer.

FORMAT

| OPTIONS | designation of information $ |
|---|---|

Explanation:

OPTIONS - keyword indicating an option specification

designation of information - the available options (complete with mnemonic designation) are listed in the applicable programmer reference manual

EXAMPLES

| | Declarations | Remarks |
|---|---|---|
| 1. | OPTIONS UYK7 $ | Specifies code generation for the AN/UYK-7 computer. |
| 2. | OPTIONS OBJECT(SM,CR) $ | Specifies source and mnemonic and full address cross reference listing. |

---

WORK AREA

1. What is the keyword for option specification?
2. Where are the options specifications placed in the total system layout?
3. What must always be the first options specification?
4. What is the basic purpose of the options specifications?

CORRECT ANSWERS

1. OPTIONS

2. directly following the system declaration

3. the target computer

4. for the programmer to select his listing and output formats

---

## 5. COMPILE TIME CONSTANTS (NTAGS)

A compile time constant assigns a name to a numeric constant. Simply put, an ntag is a name for a number and may be used anywhere in the program that the number may be used. The advantage of the ntag is that programs may be written using the name, and the value may be changed when necessary by changing only the ntag declaration at compile-time.

FORMAT

| ntag name | EQUALS | ntag expression | $ |
|---|---|---|---|

ntag name - name being declared as an ntag

EQUALS - keyword indicating a compile-time constant

ntag expression - an expression (that can be reduced to an numeric constant) that defines the ntag

EXAMPLE

Assume two tables whose lengths must be adjusted to fit different hardware configurations in which the programs must operate. The second table has three times as many items as the first.

```
TABLE WS2 H MEDIUM PARAMS $
    ==
END-TABLE WS2 $
TABLE FLOWC H MEDIUM FLOWLNTH $
    ==
END-TABLE FLOWC $
VRBL NRRESV I 12 U $
```

The number of items in tables WS2 and FLOWC is defined as the value of PARAMS and FLOWLNTH respectively. The names are defined as ntags in a header or data block preceding the table declarations.

```
PARAMS EQUALS 10 $
FLOWLNTH EQUALS PARAMS*3 $
```

At compile-time the names are replaced by the numeric constant values. Only the PARAMS ntag declaration has to be changed to change the length of both tables.

An ntag must be defined before it may be used in another ntag declaration. For instance, the PARAMS equals declaration must precede the FLOWLNTH declaration, which references it.

The ntag may be used in arithmetic expressions.

```
SET NRRESV TO PARAMS*2+3 $
```

Parenthesized expressions are not allowed in equals statements. All arithmetic is performed strictly left-to-right without operator rank. For example:

```
PRMS EQUALS ALPHA+BETA*3 $
```

will be evaluated by adding the values of ALPHA and BETA and mutiplying by 3.

---

WORK AREA

1. What keyword is used to define an ntag?

2. What is wrong with the following declaration?
   ```
   PRM EQUALS A+B*(C+D) $
   ```

3. What is the value of FLOWLNTH in the example above?

4. What is the value of NRRESV?

CORRECT ANSWERS

1. EQUALS
2. parentheses not allowed in ntag declarations
3. 30
4. 23

---

## E. SCOPE AND ALLOCATION MODIFIERS

### 1. SCOPE

Some type of organization is required to present data to a compiler so that it can do its job of translating a source program to machine code. Some of the information accumulated by the compiler is kept accessible from the point in the source statements at which it is encountered, throughout the remainder of that compilation. Such information is said to be global in scope. Other information is kept accessible only within certain limits of the source statements. Such information is said to be local in scope.

(a) The following information has Global scope:

(1) Names of systems, major headers, system data blocks and system procedure blocks. In programming for the AN/UYK-7 only, the names of local data blocks have global scope.

(2) Major header sentences.

(3) Declarations contained in system data blocks.

(4) Declarations with an (EXTDEF) or (EXTREF) scope modifier (to be discussed in the next section).

(b.) The following information has Local scope:

(1) Minor header sentences.

(2) Declaration contained in local data blocks (unless EXTDEF'ed or EXTREF'ed).

(3) Names of sub-programs within a system procedure block (unless EXTDEF'ed or EXTREF'ed).

(4) Local indexes (restricted to the sub-program in which declared).

Since global data is known throughout a system block, it can be referenced and its value can be changed by any sub-program within the system. Local data (on the other hand) is hidden from any sub-program not in the same system procedure block. It is a good programming practice to use local data wherever global is not required.

## 2. SCOPE AND ALLOCATION MODIFIERS

In CMS-2 an unmodified declaration is both a specification of the attributes of the name being declared and (for machine-addressable names) a specification of allocation. The scope of the name is determined by the position of the declaration in the source statements.

EXAMPLE

```
LOC-DD $
VRBL LINEPNTR I 8 U $

END-LOC DD $
```

The variable declaration specifies the attributes of LINEPNTR as integer, with eight unsigned bits. The compiler will assign the next available appropriate memory allocation site to LINEPNTR. The declaration is within a local data block, so it has local scope.

CMS-2 requires that all names (other than labels within a subprogram) and their attributes be known to the compiler before they may be referenced in source statements. In order to allow element compiles (i.e. compilation of portions of a full program), scope and allocation modifiers have been implemented.

A scope modifier is used to specify global scope for a name whose position would otherwise make it local. An allocation modifier specifies the attributes of a name that is to be allocated elsewhere.

The three modifiers that will be discussed are (EXTDEF) (EXTREF) and (LOCREF).

---

WORK AREA

1. What is the purpose of an allocation modifier?

2. Generally speaking, to what does global scope refer?

3. What is an element compile?

4. What is the basic purpose of scope modifiers?

CORRECT ANSWERS

1. to specify the attributes of a name allocated elsewhere

2. information accessible throughout compilation

3. compilation of portions of a program

4. to change local scope to global scope

---

a. (EXTDEF) - Keyword indicating that the name being declared has global scope. EXTDEF modifies scope only; the declaration remains an allocation declaration.

EXAMPLES

| | Declarations | Remarks |
|---|---|---|
| 1. | LOC-DD $<br>(EXTDEF) VRBL LINEPNTR I 8 U $<br>END-LOC-DD $ | LINEPNTR is declared to have global scope, despite its position in a loc-dd. Allocation will be to the next available appropriate memory site. It can be referenced from any system element that follows it. |

One reason for using an (EXTDEF) modifier on a declaration in a local data design is the discovery during program development that it is necessary to reference that data unit from another system element. The (EXTDEF) makes this possible without having to move the declaration to a system data design.

| | | |
|---|---|---|
| 2. | GENSYSP SYS-PROC $<br>(EXTDEF) PROCEDURE GENEXEC $ | Procedure GENEXEC is declared to have global scope, making it possible to call the procedure from other system elements. Allocation occurs in the usual manner. |

b. (EXTREF) - Keyword specifying the attributes of the name declared and indicating that it will be allocated elsewhere. The declaration modified by (EXTREF) must appear in either a local or system data block. (EXTREF) is always an allocation modifier and (when the declaration is in a loc-dd) is also a scope modifier.

EXAMPLE

```
TDD         SYS-DD $
            VRBL GENFLG B $
            END-SYS-DD TDD $
            SCREIN     SYS-PROC $
                       LOC-DD $
            (EXTREF)   PROCEDURE GENEXEC $
            (EXTREF) VRBL GENFLG B $
                       END-LOC-DD $
                          =
                          =
                       SET GENFLG TO 0 $
                       GENEXEC $
                          =
                          =
                          =
```

Remarks

GENFLG is defined and allocated in the SYS-DD. Within the LOC-DD, GENFLG is specified as a Boolean variable, having global scope, allocated elsewhere. GENEXEC is specified as a procedure with no formal parameters, having global scope, allocated elsewhere.

The two system elements TDD and SCREIN may be compiled together, with or within the element containing GENEXEC, or may be compiled separately without error. The separate compilation capability allows smaller segments to be compiled for checkout and then joined to the whole later.

Rules:

(1) A name may appear in any number of (EXTREF) declarations, but only once in either an (EXTDEF) or a system data block declaration.

(2) The attributes specified in the (EXTREF) and the (EXTDEF) or system data block declaration for a name must be the same.

---

WORK AREA

1. What do the two scope modifiers discussed above accomplish?

2. Which one modifies scope only?

3. Which one modifies allocation primarily and scope secondarily?

4. What is the restriction on the use of a name with these two scope modifiers?

CORRECT ANSWERS

1. change local scope to global scope

2. (EXTDEF)

3. (EXTREF)

4. name may appear only once in (EXTDEF), but in any number of (EXTREF) declarations

---

c. (LOCREF) - Keyword specifying an attribute declaration for a function or procedure that will be allocated later in the same system procedure block. The declaration must appear in a local data block. It is an allocation modifier only. It allows the programmer to organize his sub-programs in some fashion (for example, alphabetical order) without regard to how they are referenced.

EXAMPLE

```
            LOC-DD $
            VRBL L1ID I 15 U $
            VRBL DECID I 15 U $
 (LOCREF)   FUNCTION STATELOC(L1ID) I 15 U $
            END-LOC-DD $

            SET DECID TO STATELOC(@(1273)) $

            FUNCTION STATELOC(L1ID) I 15 U $
```

The (LOCREF) declaration in the LOC-DD makes it possible to call function STATELOC before it is defined and allocated.

Rules:

(1) A (LOCREF) must appear in a local data block and may apply to sub-programs only, not to data units.

(2) The attributes in the (LOCREF) declaration and the sub-program declaration must be the same.

SYSTEM EXAMPLE

```
GENTEST    SYSTEM $
           OPTIONS UYK7 $
           OPTIONS OBJECT(SM,CR) $
           LIBS CCOMN $
           SEL-POOL HEREPOOL $
           SYS-INDEX 1 CTSX $
           END-HEAD $
GENDD      SYS-DD $
PARAMS     EQUALS 15 $
(EXTREF)   PROCEDURE DOUT $
             =
             =
           END-SYS-DD GENDD $
           HEAD $
PARAMS     EQUALS 10 $
           END-HEAD $
GENPROC    SYS-PROC $
           LOC-DD $
           TABLE WS3 H (A 26 S 9) PARAMS $
           END-TABLE WS3 $
(LOCREF)   FUNCTION FIGET(WS3) A 26 S 9 $
             =
             =
           END-LOC-DD $
(EXTDEF)   PROCEDURE DOUT $
             =
             =
           END-PROC DOUT $
           FUNCTION FIGET(WS3) A 26 S 9 $
           END-FUNCTION FIGET $
           END-SYS-PROC GENPROC $
           END-SYSTEM GENTEST $
```

## Chapter 8 Summary

The minimum arrangement required for a successful compile is presented. Emphasis is placed on organizing system elements (SYS-DD,SYS-PROC). Methods of allocating data and procedures are explained.

Use this page for notes or comments.

## CHAPTER 8 QUIZ

1. What is meant by global information?

2. Name the system elements

3. What is the obvious difference between a label and a system name

4. The following keywords refer to what declaration?
   a. SYS-DD
   b. SYS-PROC
   c. LOC-DD
   d. HEAD

5. What is a local data block always contained within?

6. How many major header blocks may there be in one system block?

7. Naming global variables whose values are to be held in index registers is accomplished with what keyword?

8. Where must the OPTIONS declaration be placed?

9. Since HEAD is the same keyword for both major and minors headers, how can one tell the difference?

10. What are the scope modifiers?

11. How would a procedure be made global in scope?

12. What is ntag used for?

13. What keyword is used for ntag declarations?

14. What is the function of the following?

    (a) (LOCREF)
    (b) (EXTDEF)
    (c) (EXTREF)

## CORRECT ANSWERS

1. information that is kept throughout a compilation

2. system data blocks (SYS-DDs) and system procedures (SYS-PROCs)

3. the system name has no period

4. a. system data block
   b. system procedure block
   c. local data block
   d. major or minor headers block

5. within a SYS-PROC

6. only one

7. SYS-INDEX

8. as the first major header sentence in the major header immediately following the system declaration or (major) HEAD

9. the very first END-HEAD declaration delimits the major header. All others refer to minor headers.

10. (EXTDEF), (EXTREF)

11. using (EXTDEF)

12. to assign a name to a numeric constant

13. EQUALS

14. (a) an allocation modifier only
    (b) a scope modifier only
    (c) may modify both allocation and scope

# CHAPTER 9

# SWITCH BLOCKS AND CASE BLOCKS

## A. SWITCH BLOCKS

Switch blocks are a means for conditional transfer of program control to one of several different locations, depending on conditions specified by the programmer. Label switches are used for unconditional transfer and procedure switches are used when transfer with return is desired.

### 1. INDEXED LABEL SWITCHES

An indexed label switch block specifies the name of the indexed label switch and the names of the statements to which control is transferred when a corresponding indexed branch phrase is executed. This is a complete transfer of control to one of the label switch points.

FORMAT

```
SWITCH    label switch name $

label switch point(s) $

END-SWITCH label switch name $
```

Explanation:

| | |
|---|---|
| SWITCH | - keyword indicating a label switch declaration |
| label switch name | - the name of the indexed label switch being declared |
| label switch points | - one or more names of statements to which control may be transferred by execution of an indexed branch phrase |
| END-SWITCH | - keyword indicating the end of a switch block |
| label switch name | - this must be the same as the name in the SWITCH declaration. |

EXAMPLE

```
SWITCH SETMSG $
   NOERRS $
   SRCERR $
   GENERR $
    EDERR $
END-SWITCH SETMSG $
```

RULES

1. An indexed label switch may appear only in a local data block.

2. The switch points (names) declared in the block must be names of statements that follow in the system procedure containing the switch block.

3. There is no limit on the number of switch points in an indexed label switch block.

---

WORK AREA

1. What is the purpose of using switch blocks?

2. What type of switch is used for:

    a. transfer and return?
    b. transfer with no return?

3. What are the statement names in a switch block called?

4. What is the limitation to the number of statement names that may be assigned to a switch?

5. Write a set of declarations for indexed label switch INOP with 3 switch points (called IN1, IN2, and IN3).

6. Where may indexed label switches be declared?

CORRECT ANSWERS

1. To provide a transfer of program control to one of several different locations

2. a. procedure switch
   b. label switch

3. switch points

4. no limits

5.
```
SWITCH INOP $
   IN1 $
   IN2 $
   IN3 $
END-SWITCH INOP $
```

6. local data blocks only

---

## 2. INDEXED BRANCH PHRASE

An indexed branch phrase is used to transfer to one of the switch points in an indexed label switch. It is an unconditional transfer, specifying the next instruction to be executed, depending on the value of an index expression.

FORMAT

| GOTO | label switch name | switch index | INVALID | abnormal branch | $ |
|---|---|---|---|---|---|

Explanation:

GOTO - keyword indicating a branch phrase or an indexed branch phrase

label switch name - name of an indexed label switch that specifies the possible statements to which control may be transferred

switch index - a numeric expression which (used together with the indexed label switch) specifies the statement that will be executed next

INVALID - (optional) keyword indicating that an abnormal branch is being specified

abnormal branch - (optional) name of the statement to be executed next if the value of the index is out of the allowable range

EXAMPLE

```
VRBL ERRCOD I 3 U $

GOTO SETMSG ERRCOD INVALID OUTLIM $
```

Since SETMSG has four switch points (statement names), the valid index values are 0 through 3 inclusive. There are two possible occurences:

(1) When ERRCOD is in range, control is transferred to the corresponding switch point of SETMSG (i.e. ERRCOD equals 0, transfer to NOERRS; ERRCOD equals 1, transfer to SRCERR; ERRCOD equals 2, transfer to GENERR; etc.).

(2) When ERRCOD is out of range, control is transferred to OUTLIM.

When ERRCOD is out of range and INVALID was omitted, execution of the indexed branch phrase has undefined results.

For convenience, the switch index in the example is an integer variable, but any numeric expression may be used. For example:

(A+B)/C or SXFINDER(P*Q) where SXFINDER is a function having a numeric value ---- are valid switch index expressions. If the value of the switch index expression after evaluation includes a fractional portion, the compiler will discard it and use only the integer portion to index the switch.

---

WORK AREA

1. If label switch ERROP has six switch points, what would be the valid index values?

2. An unconditional transfer uses what keyword?

3. What index values would be out of range in question 1 above?

4. What keyword is used to help take care of out of range conditions?

CORRECT ANSWERS

1. 0 through 5

2. GOTO

3. values less than 0 or greater than 5

4. INVALID

---

EXAMPLE

This example contains the previous examples shown for a label switch and indexed branch phrase. The complete procedure and all data units needed to print one of several error messages are presented.

```
SP7         SYS-PROC $

            LOC-DD $

            TABLE PRNTTBL V (H 120) 100 ''100 PRINTLINES
                FOR REPORT'' $

            END-TABLE PRNTTBL $

            VRBL NXTLIN I 7 U ''INDEX FOR PRNTTBL'' $

            COMMENT MESSAGE LINES SPECIFIED IN
                ANOTHER SYSTEM ELEMENT $

(EXTREF)    VRBL (SRCMSG,GENMSG,EDMSG) H 120 $

(EXTREF)    VRBL ERRCOD I 3 U $

            SWITCH SETMSG $
                NOERRS $
                SRCERR $
                GENERR $
                EDERR $
            END-SWITCH SETMSG $

            END-LOC-DD $
```

```
(EXTDEF)    PROCEDURE GETMSG INPUT ERRCOD $
            COMMENT THIS PROCEDURE ADDS ONE PRINT LINE TO
            BUFFER TABLE PRNTTBL $

            GOTO SETMSG ERRCOD INVALID OUTLIM $

NOERRS.     RETURN ''ERRCOD IS Ø, NO ERRORS'' $

SRCERR.     SET PRNTTBL(NXTLIN) TO SRCMSG ''ERRCOD
                IS 1, USE SRCMSG'' $
            GOTO ALLDONE $

GENERR.     SET PRNTTBL(NXTLIN) TO GENMSG ''ERRCOD
                IS 2, USE GENMSG'' $
            GOTO ALLDONE $

EDERR.      SET PRNTTBL(NXTLIN) TO EDMSG $
            GOTO ALLDONE $

OUTLIM.     SET PRNTTBL (NXTLIN) TO H(INVALID ERROR CODE)
                ''BAD INPUT.  PRINT SPECIAL MESSAGE'' $

ALLDONE.    SET NXTLIN TO NXTLIN+1 ''INCREMENT
                LINE COUNTER'' $

            END-PROC GETMSG $
            END-SYS-PROC SP7 $
```

Procedure GETMSG uses input parameter ERRCOD as an index to switch SETMSG to determine which statements will be executed. Should ERRCOD be out of the allowable range (Ø through 3), a special message is output.

---

WORK AREA

Assume that the GOTO statement in procedure GETMSG was written without the INVALID specification:

```
GOTO SETMSG ERRCOD $
```

A call on GETMSG is made with a value of 6 in ERRCOD. What would happen?

The program would transfer to some unspecified location with undefined (and probably disastrous) results.

---

## 3. INDEXED PROCEDURE SWITCH

An indexed procedure switch block specifies the name of an indexed procedure switch, the names of procedures that are called when a corresponding indexed procedure call phrase is executed, and the common formal input and output parameters of those procedures. This is known as a transfer-and-return switch.

Sometimes instead of transferring directly to another program location, it may be desirable to call one of a number of procedures depending upon a programmer specified condition. This is accomplished by using an indexed procedure switch.

FORMAT

```
P-SWITCH        indexed procedure    formal input and output    $
                  switch name              parameters

          indexed procedure switch point(s) $

END-SWITCH      indexed procedure    $
                  switch name
```

Explanation:

P-SWITCH - keyword indicating a procedure switch declaration

indexed procedure switch name - name of the indexed procedure switch being declared

formal parameters - (optional) a declaration of the formal input and output parameters for the procedures named as procedure switch points

index procedure switch point - the name of a procedure to be called by use of an indexed procedure call phrase

RULES

1. The name in the end-switch declaration must match the name in the indexed procedure switch declaration.

2. There is no limit on the number of procedure switch points in an indexed procedure switch.

3. All the procedures named as procedure switch points must have the same formal input and output parameters.

4. Procedures named as procedure switch points may not have formal exit parameters.

5. The appearance of a procedure name as a procedure switch point is an attribute declaration (i.e. an EXTREF or LOCREF). If the name has not been declared previously, it has the same scope (global or local) as the indexed procedure switch name.

6. Indexed procedure switch declarations may appear in either global or local data blocks; however, if declared in a global data block the procedure switch may contain only global procedure names. A procedure switch in a local data block may contain local and/or global procedure names.

EXAMPLE

```
SWDD  SYS-DD $
      VRBL ANGLE A 12 U 4 $
      VRBL SIDE A 9 U 4 $
      VRBL SOLUTION A 12 U 9 $

      P-SWITCH TRIG INPUT ANGLE, SIDE
          OUTPUT SOLUTION $
          SINE $
          COSINE $
          TANGENT $
          COTANG $
      END-SWITCH TRIG $
      END-SYS-DD SWDD $
```

TRIG is an indexed procedure switch with four procedure switch points. Each of the switch points is the name of a procedure (SINE, COSINE, TANGENT and COTANG) which has two formal input parameters (ANGLE, SIDE) and one formal output parameter (SOLUTION).

Since TRIG is declared in a global data block, the names of the procedure switch points must be global as well. This means that when each of the corresponding procedure blocks is declared, it must be EXTDEFed.

## 4. INDEXED PROCEDURE SWITCH CALL

An indexed procedure switch call specifies the execution of one of a set of procedures, depending upon the value of an index expression.

FORMAT

| | | | | | | |
|---|---|---|---|---|---|---|
| indexed procedure switch name | USING | switch index | INVALID | abnormal branch | actual input/ output parameters | $ |

Explanation:

| | |
|---|---|
| indexed procedure switch name | - name of the indexed procedure switch that specifies the possible procedures to be executed |
| USING | - keyword indicating that the switch index expression follows |
| switch index | - a numeric expression whose value used together with the indexed procedure switch specifies the procedure to be executed |
| INVALID | - (optional) keyword indicating that an abnormal branch is being specified |
| abnormal branch | - (optional) name of the statement to be executed next if the value of the index is out of the allowable range |
| actual input/output parameters | - (optional) specification of the actual input and output parameters to be used in the procedure call |

As with indexed label switches, the switch index may be any valid numeric expression.

Omission of an abnormal branch phrase may have undefined results.

EXAMPLE

```
VRBL (ANG,SOL) A 12 U 9 $

VRBL SID A 9 U 4 $

VRBL IX I 2 U $

TRIG USING IX INVALID NOFUNC INPUT ANG,SID
    OUTPUT SOL $
```

1. The procedure specified by the value IX will be executed with ANG and SID as actual input parameters and SOL as the actual output parameter.

2. If the value of IX should not be in the valid range of zero through 3, the statement at label NOFUNC will be executed.

3. Had INVALID been omitted and IX were out of range, execution of the indexed procedure switch call would have undefined results.

---

WORK AREA

1. Write the procedure declaration for procedure SINE. (Refer to example on page 189)

2. What do the EXTREF modifiers on the following variable definitions signify?

```
(EXTREF) VRBL ERRCOD I 3 U $
(EXTREF) VRBL (SRCMSG,GENMSG,EDMSG) H 120 $
```

CORRECT ANSWERS

1. (EXTDEF) PROCEDURE SINE INPUT ANGLE,SIDE
   OUTPUT SOLUTION $

2. That these are attribute declarations only and the variables are global in scope and allocated at another place in the program.

---

## B. CASE BLOCKS

A case block specifies a number of blocks of statements, one of which is to be executed, depending on the value of a selector. Case blocks are a convenient way to accomplish the same thing switches do with usually less programmer effort. The use of case blocks avoids the necessity of programmer defined switches and keep the logic of what happens for each selector value clearly set out in a distinct block of statements. Case blocks also follow the rules of structured programming. For these reasons, use of case blocks instead of switches is encouraged.

FORMAT

```
FOR   case selector   ELSE   alternative $

      BEGIN   case value(s) $

                  value block body

      END $


END $
```

Explanation:

| | |
|---|---|
| FOR | - keyword indicating the beginning of a case block |
| case selector | - simple single-valued data unit |
| ELSE | - (optional) keyword indicating that an alternative statement follows |
| alternative | - a statement to be executed if no block of statements is selected |
| BEGIN | - keyword indicating a begin block |
| case value | - a constant to be compared to the value of the case selector in determining which block of statements is to be executed |
| value block body | - (optional) a block of statements to be executed if the value of the case selector matches one of the case values of the associated BEGIN |

When a case block is executed, the value of its case selector is considered. If it matches one of the case values specified in a begin statement within the block, the value block body associated with that begin is executed.

Unless a statement within the begin block causes execution to transfer without return to a statement outside the case block, the next statement executed will be the one immediately following the case block.

If the case selector does not match one of the specified case values and an else clause is specified, the alternative statement is executed. Otherwise execution continues with the statement immediately after the case block.

## EXAMPLE

The following example shows how system procedure SP7 might be written using a case block instead of an indexed label switch.

```
01  SP7     SYS-PROC $
02          LOC-DD $
03          TABLE PRNTTBL V (H,120) 100 $
04          END-TABLE PRNTTBL $
05          VRBL NXTLIN I 7 U ''INDEX FOR PRNTTBL'' $
06          VRBL (SRCMSG,GENMSG,EDMSG) H 120 $
07          VRBL ERRCOD I 3 U $
08          END-LOC-DD $

09          PROCEDURE GETMSG INPUT ERRCOD $
10          FOR ERRCOD ELSE SET PRNTTBL (NXTLIN) TO
                H(INVALID ERROR CODE)
11              BEGIN 0 $
12                RETURN ''NO ERRORS'' $
13              END $
14              BEGIN 1 $
15                SET PRNTTBL(NXTLIN) TO SRCMSG $
16              END $
17              BEGIN 2 $
18                SET PRNTTBL(NXTLIN)TO GENMSG $
19              END $
20              BEGIN 3 $
21                SET PRNTTBL(NXTLIN) TO EDMSG $
22              END $
23          END $
24          SET NXTLIN TO NXTLIN+1 $
25          END-PROC GETMSG $
26          END-SYS-PROC SP7 $
```

A detailed explanation of the case block portion (lines 10-23) of the previous subprogram is as follows:

10 ERRCOD is used as the case selector in a case (FOR) block statement. The value of ERRCOD will determine which of the value (BEGIN) blocks within the case block is executed. The ELSE clause serves the same purpose as the INVALID phrase in a switch transfer.

11 The value block declaration corresponding to ERRCOD equal to zero.

12 The value block body to be executed. Returns control to calling subprogram.

13 End of value block. Since #12 returned control to the calling sub-program, this statement has no effect on program execution, but is required for correct compilation.

14 The value block declaration corresponding to ERRCOD equal 1.

15 The value block body to be executed. Moves specified message to PRNTTBL.

16 End of value block, transfers control to statement 24.

17 The value block declaration corresponding to ERRCOD equal 2.

18 The value block body to be executed. Moves specified message to PRNTTBL.

19 End of value block, transfers control to statement 24.

20 The value block declaration corresponding to ERRCOD equal 3.

21 The value block body to be executed. Moves specified message to PRNTTBL.

22 End of value block, transfers control to statement 24.

23 End of case block.

The case block statement above (statement 10) contains an else clause. If the else clause were omitted and no value block match was made, execution would "fall through" to the statement following the case block (statement 24). This is a distinct improvement over a switch without an invalid phrase, where an out-of-range reference has undefined results.

It is also possible to specify more than one value for a given value block. For instance:

```
BEGIN 1,4 $

END $
BEGIN 0,5,3 $

END $
```

The first value block would be executed for values of one and four, the second for values of zero, five or three. Notice that no specific provision for a value of two is made. This becomes a "fall through" or else clause condition. The ability to omit provision for a value, bracketed by those really required, does not exist in switches.

Any number of values may be assigned to a given value block, but no value may be used for more than one value block in the same case block.

## Chapter 9 Summary

Declaration and use of indexed label-switch blocks (complete transfer of control) and procedure switch blocks (transfer-and-return) are discussed. Use of case blocks instead of switches is explained and encouraged.

## CHAPTER 9 PROBLEM

**Problem Statement:**

Using the given data declarations (below), write a procedure called PROCSTAT that uses variable STATCODE as the case selector for a case block and does the following:

(1) For status codes of 0,30,31,32,42 and 44 (which indicate good conditions), assign the core address of table LSTBUF to table L2HAN.

(2) For status codes 11,12,17,21 and 22 (which are unrecoverable error conditions), call procedure ABORT.

(3) For status code 13, set the end-of-tape flag (EOTFLG).

(4) For status code 16, set the end-of-file flag (EOFFLG).

(5) If the status code is not one of the above, call procedure ABORT (i.e. an else clause condition).

(6) Following the case block, the procedure should clear the channel busy flag (CHANABSY).

**Given:**

```
            LOC-DD $

(EXTREF)    PROCEDURE ABORT ''FAILURE ROUTINE, PRINTS
                LOCATION WHERE FAILURE OCCURRED AND
                PROGRAM+DATA AREAS'' $

(EXTREF)    TABLE LSTBUF V 1 512 $
            END-TABLE $

(EXTREF)    TABLE L2HAN V 1 INDIRECT $
            END-TABLE $

(EXTDEF)    VRBL STATCODE I 6 U ''STATUS CODE SET BY
                CHANA INTERRUPT PROCESSOR'' $

(EXTDEF)    VRBL (CHANABSY,EOFFLG,EOTFLG) B ''I/O IN
                PROGRESS,END-OF-FILE,END-OF-TAPE FLAGS'' $

            END-LOC-DD $
```

CORRECT ANSWER

```
PROCEDURE PROCSTAT ''PROCESS CHANA STATUS CODES'' $
FOR STATCODE ELSE ABORT ''IF CODE NOT RECOGNIZED,
                          ABORT'' $
        BEGIN 0,30,31,32,42,44 ''GOOD COMPLETIONS'' $
          SET CORAD(L2HAN) TO CORAD (LSTBUF) $
        END ''END GOOD COMPLETIONS'' $
        BEGIN 11,12,17,21,22 ''UNRECOVERABLE ERRORS'' $
          ABORT $
        END $
        BEGIN 13 ''END-OF-TAPE'' $
          SET EOTFLG TO 1 $
        END $
        BEGIN 16 ''END-OF-FILE'' $
          SET EOFFLG TO 1 $
        END $
END ''END OF CASE BLOCK'' $
SET CHANABSY TO 0 ''CLEAR I/O ACTIVE FLAG'' $
END-PROC PROCSTAT $
```

# CHAPTER 10

# INTRINSIC FUNCTIONS

## INTRINSIC FUNCTIONS

Functions were discussed in Chapter 6. A number of special purpose functions have been provided as an integral part of the CMS-2 language. These functions are referenced just as a user declared function is, but during execution they do not affect the value of any data unit in the user's program.

The CORAD intrinsic function was discussed in Chapter 7. Five additional functions (ABS, CONF, REM, SCALF and TDEF) will be discussed in this chapter.

### A. ABSOLUTE VALUE FUNCTION REFERENCE (ABS)

The absolute value function reference (call) returns the absolute value of its actual input parameter.

FORMAT

```
. . . ABS (numeric expression) . . .
```

Explanation:

ABS - keyword indicating an absolute value function reference
numeric expression - an expression whose absolute value is desired

The type of the value returned is the same as that of the actual input parameter except that it is unsigned (i.e., always positive).

EXAMPLES

1.
```
VRBL FLTDEX F $

VRBL FIXDEX A 14 S 6 $

IF ABS(FLTDEX) EQ ABS(FIXDEX) THEN RETURN $
```

Resulting action:
(a) The first absolute value function reference returns a positive floating-point value.
(b) The second absolute value function reference returns an unsigned fixed-point value (A 13 U 6). Only 13 bits are required instead of the 14 in FIXDEX because no sign bit is needed.
(c) The second value is then converted to floating-point for the comparison.

2. 
```
VRBL ABSDIF I 31 U $
VRBL (IND1,IND2) I 32 S $

SET ABSDIF TO ABS(IND1-IND2+1) $
```

Resulting action:
The expression in parentheses will be evaluated first, then its absolute value will be assigned to ABSDIF. Supposing IND1 equals 10 and IND2 equals 14, the value of the expression is -3 and the absolute value is 3.

---

WORK AREA

1. If actual values in example 1 were as follows:

   a. FLTDEX=12.03 FIXDEX=12.00

   b. FLTDEX=-9.00 FIXDEX=9.00

   Where would the program go after evaluation of the IF statement?

2. In example 2 above, if actual value before execution of the statement were as follows:

```
ABSDIF = 8
IND1   = -5
IND2   = 2
```

   What would be the value of ABSDIF after execution of the SET statement?

CORRECT ANSWERS

1. a. to the statement following IF
   b. to RETURN
2. the value of the expression is -5-2+1=6

   The absolute value of -6 is 6, which is the value assigned to ABSDIF. The original value of ABSDIF is irrelevant.

---

## B. CONVERSION FUNCTION REFERENCE (CONF)

A conversion function reference specifies the conversion of a numeric value to another numeric type.

FORMAT

```
. . . CONF(target conversion type,conversion source). . .
```

Explanation:

| | |
|---|---|
| CONF | - keyword indicating a conversion function reference |
| target conversion type | - specification of the numeric type to which the value is to be converted |
| conversion source | - a numeric expression whose value is to be converted to the target type |

EXAMPLE

```
VRBL REL A 11 S 3 $
VRBL VEL A 8 S 5 $

IF CONF(A 10 S 2,REL) EQ CONF(A 10 S 2,VEL)
                    THEN RETURN $
```

The rightmost fractional bit of REL and the rightmost 3 fractional bits of VEL will be truncated and the results will be compared.

Note: Use of a conversion function reference is considered a statement by the programmer that the current value of the source expression can be adequately represented by the target type. No effort is made by the compiler to verify that the representation is correct.

For example:

```
VRBL REL A 11 S 3 $
VRBL NEL A 16 S 3 $

SET REL TO CONF(A 11 S 3,NEL) $
```

It is assumed that the current value of NEL "fits" the A 11 S 3 target specification. The transfer is made without causing any warning message.

CONF may be used with fixed-point and integer operands only. Any portion of the type specification may be changed -- arithmetic type, bit configuration and sign.

Parts of a program may require different degrees of accuracy of the data used. One use of CONF is to discard bits of significance that are not required. For instance, if two variables are defined as A 14 S 3, a portion of the program that needs only the integer values may use CONF to make the values A 11 S 0 (or I 11 S).

---

WORK AREA

1. Using the variables shown below:

```
VRBL MED A 12 S 5 $
VRBL LED A 15 S 4 $
VRBL NED I 14 U $
```

   a. Write a CONF function reference to convert LED to integer type with 10 unsigned bits.

   b. Write a statement to compare MED and NED as I 6 U types and go to DONE if they are equal.

CORRECT ANSWERS

1. a. CONF(I 10 U,LED)
   b. IF CONF(I 6 U,MED) EQ CONF(I 6 U,NED)
      THEN GOTO DONE $

---

## C. REMAINDERING FUNCTION REFERENCE (REM)

A remaindering function reference returns the remainder of a fixed-point division operation.

FORMAT

```
. . . REM(remaindering expression) . . .
```

Explanation:

REM - keyword indicating a remaindering function reference

remaindering expression - a numeric expression containing the fixed-point division operation whose remainder is desired

EXAMPLES

1.
```
VRBL (REMAIN,ROUNDED) A 15 S 6 $
VRBL (FIXDEX,BEL) A 14 S 6 $

SET REMAIN TO REM(FIXDEX/BEL) $
```

The remainder of the division operation is assigned to REMAIN. Assuming FIXDEX equals 13 and BEL equals 5, the division operation 13/5 will give a quotient of 2 and a remainder of 3. REMAIN will be assigned the value 3.

2. SET ROUNDED TO (FIXDEX-BEL)**2/3+
REM((FIXDEX-BEL)**2/3) $

The remainder of the division operation is added to the quotient and the result is assigned to ROUNDED. Using the same values as in the first example, the equation is (13-5)**2/3(which is 64/3), yielding a quotient of 21 and a remainder of 1. ROUNDED will be assigned the value 22, which is 21+1.

Rules:

(1) The remaindering expression must contain only fixed-point operands (constants and integer types are considered fixed-point for this operation).

(2) The division operation must be indicated by the division operator (/) and there must be one (and only one) division operation.

(3) The sign of the remainder is the sign of the numerator and the number of magnitude bits is the number of magnitude bits in the denominator, after any alignment prior to the division operation.

(4) The number of fractional bits in the remainder is the number of fractional bits in the numerator, after alignment prior to the division operation.

---

WORK AREA

Given the definitions:

```
VRBL FLTDEX F $
VRBL LED A 15 S 4 $
VRBL NED I 14 U $
VRBL MED A 12 S 5 $
```

Are the following statements valid?

a. IF REM(FLTDEX/MED) NOT 0 THEN GOTO RNDOFF $
b. SET MED TO REM(LED/NED) $
c. SET FLTDEX TO REM(LED**3/NED) $
d. SET MED TO REM((LED/NED)/2) $

CORRECT ANSWERS

a. NO - the division operation must be fixed-point
b. YES
c. YES
d. NO - may have only one division operation

---

## D. SCALING SPECIFICATION FUNCTION REFERENCE (SCALF)

A scaling specification function reference specifies a fixed point numeric expression and a scale factor to be used in evaluating the expression and aligning its value. More simply, it allows modifying the number of fractional bits.

The SCALF function is used when the programmer desires intermediate results to have either more or fewer bits of scaling than the compiler scaling rules produce.

### FORMAT

```
. . . SCALF(scale factor, controlled expression) . . .
```

Explanation:

| | |
|---|---|
| SCALF | - keyword indicating a scaling specification function reference |
| scale factor | - a numeric constant expression whose value specifies the scaling to be used during evaluation of the controlled expression and the final alignment of its value |
| controlled expression | - a numeric expression whose evaluation is controlled by the scale factor |

### EXAMPLES

```
VRBL NEL A 16 S 3 $
VRBL FIXDEX A 14 S 6 $
VRBL REL A 11 S 3 $
VRBL VEL A 8 S 5 $
```

1. `SET NEL TO SCALF(5,REL+VEL) $`

The value of REL will be aligned to a scaling of 5 and added to the value of VEL.

2. SET NEL TO SCALF(6,(FIXDEX+VEL)/REL) $

The value of VEL will be aligned to a scaling of 6 and added to the value of FIXDEX. The sum will be aligned to a scaling of 9, so that after the division the quotient will have a scaling of 6.

3. IF FIXDEX EQ SCALF(6,(NEL-VEL)*REL) THEN RETURN $

The values of NEL and VEL will be aligned to a scaling of 6 and the subtraction will be performed. Then (without pre-alignment) the result will be multiplied by the value of REL. The product (initially scaled 9) will be aligned to the scale factor 6.

Rules:

(1) The controlled expression must have at least two operands, at least one of which is non-constant.

(2) The controlled expression may contain only fixed-point and constant operands.

(3) Exponentiation is permitted only if the exponent is a constant expression with integer value.

(4) For addition and subtraction, operands are aligned to the scale factor before the operation.

(5) For multiplication (including exponentiation), no pre-alignment occurs. The product is aligned to the scale factor.

(6) For division, the numerator is aligned before the division to a value such that the quotient's scaling equals the scale factor.

(7) The scale factor must be an integer value in the range (-127,127) and the scaling operations involved must be able to be performed using the fixed-point instructions of the target computer.

---

WORK AREA

Using the definitions on the previous page:

Write a statement that will transfer to NOTLS if the quotient of NEL divided by VEL, with a scale factor of 4, is greater than or equal to 7.

CORRECT ANSWER

```
IF SCALF(4,NEL/VEL) GTEQ 7 THEN
    GOTO NOTLS $
```

---

## E. TEMPORARY DEFINITION FUNCTION REFERENCE (TDEF)

A temporary definition function reference causes a bit string representing a value of one type to be treated as if it were representing a value of another type.

Use of TDEF requires detailed knowledge of the internal machine representations and the compiler's method of handling the data.

FORMAT

```
. . . TDEF(target redefinition type, redefinition source) . . .
```

Explanation:

| | |
|---|---|
| TDEF | - keyword indicating a temporary definition function reference |
| target redefinition type | - specification of the type to which the bit string is to be converted |
| redefinition source | - a simple expression whose value is the bit string to be converted |

EXAMPLES

```
VRBL VEL A 8 S 5 $
VRBL FIXDEX A 14 S 6 $
VRBL IND1 I 32 S $
VRBL CHARV H 4 $
```

1. SET VEL TO TDEF(A 8 S 5, FIXDEX) $

The bit string representing FIXDEX is loaded into a register and treated as if it were a bit string of type A 8 S 5. It is stored in VEL without any manipulation. If FIXDEX had simply been assigned to VEL without TDEFing, it would have been aligned to VEL by right shifting one bit.

2. IF IND1 EQ TDEF(I 32 S,CHARV) THEN RETURN $

The bit string representing the value of CHARV is compared to the value of IND1.

Rules:
(1) The target type must be either integer or fixed-point.

(2) The number of bits required by the target type must not be greater than the number of bits of the source expression.

---

WORK AREA

Given the following:

```
VRBL VEL A 8 S 5 $
VRBL FIXDEX A 14 S 6 $
VRBL CHARV H 4 $
```

Are the following TDEF function references valid?

a. TDEF(F,FIXDEX)
b. TDEF(I 13 U, FIXDEX)
c. TDEF(A 32 S 0, CHARV)
d. TDEF(I 9 U, VEL)

CORRECT ANSWERS

a. NO - target type must be either integer or fixed-point
b. YES
c. YES
d. NO - target type has more bits than source

---

## Chapter 10 Summary

Intrinsic functions are sets of operations which have been devised to perform various useful tasks which otherwise would have to be developed by the programmer. These functions (ABS, CONF, REM, SCALF and TDEF) have been introduced and briefly explained in this chapter.

## FINAL PROBLEM

The problem is to update information pertaining to Navy vessels in an imaginary task force situation.

Using the data declarations given below, write a procedure called UPDATE with item-area NEWDATA as a formal input parameter, to do the following:

(1) Calculate the range of the vessel whose data is being updated by the NEWDATA entry. Fuel now on board divided by the rate of fuel consumption at maximum speed times the maximum speed equals range.

(2) When the range has been calculated, write a loop to find the entry in table TASKF which matches the field SHIPDES in NEWDATA and set that table item to NEWDATA, then return.

```
TABLE TASKF V MEDIUM NUMSHPS    ''TASK FORCE SHIP INFO'' $

   FIELD SHIPDES H 8      ''SHIP DESIGNATION'' $

   FIELD MAXSPD I 6 U     ''MAX SPEED IN KNOTS'' $

   FIELD FUELNOW I 15 U   ''CURRENT FUEL IN POUNDS'' $

   FIELD FUELRATE I 6 U   ''FUEL RATE, POUNDS/HOUR'' $

   FIELD RANGE I 12 U     ''MAX RANGE, IN NAUTICAL MILES'' $

   ITEM-AREA NEWDATA      ''WORK AREA'' $

END-TABLE TASKF $
```

CORRECT ANSWERS

```
PROCEDURE UPDATE INPUT NEWDATA $

LOC-INDEX LNX $

SET NEWDATA(RANGE) TO (NEWDATA(FUELNOW)/NEWDATA
    (FUELRATE))*NEWDATA(MAXSPD) ''SET NEW RANGE
    VALUE ON BASIS OF CURRENT FUEL STATUS'' $

VARY LNX WITHIN TASKF ''FIND ITEM TO REPLACE'' $

    IF TASKF(LNX,SHIPDES) EQ NEWDATA(SHIPDES)
        THEN BEGIN ''ITEM FOUND, REPLACE IT'' $

            SET TASKF(LNX) TO NEWDATA $

            RETURN ''ALL DONE,RETURN'' $

            END ''ITEM FOUND'' $

END ''FIND ITEM'' $

END-PROC UPDATE $
```

# APPENDIX A

## COMPOOLS

## Compools

A compool is an ISCM file consisting of system data and major header elements in a modified CMS-2 internal symbol table format. ISCM stands for INTRA-SYSTEM COMMUNICATIONS MEDIUM. Conceptually, ISCM may be thought of as a library containing various kinds of files.

Compools are created by compiling a system consisting of only the major header and system data blocks. The file thus created may later be input as part of other compilations. The effect of using a compool input is as if the source lines that were used in forming it were already present at the beginning of compilation.

Some reasons for using compools are:

(1) The compool is pre-digested into internal format which usually saves processing time later when the rest of the program is compiled using the compool as an input, particularly when the program is compiled many times.

(2) The entries retrieved from the compools do not appear in output listings, saving processing and printing time, resulting in output listings which are a more manageable size.

(3) The compiler's internal data storage limits may be exceeded when large data bases are input. This causes compiling to abort. It is then necessary to use some combination of compools and element or segment compiles, to get a symbol table within limits.

(4) Recompiling large, full systems is inefficient if only a small portion is being changed. Using compools (from which the compiler retrieves only those entries actually needed) and compiling only those elements that change results in very significant savings in time.

### CREATING A COMPOOL

A compool is created by compiling a system containing only a major header and system data blocks and requesting a CMP output. The request is made in an OPTIONS major header sentence. Should the compiler encounter source elements other than a major header, minor header and ssy-dd's, the CMP request will be ignored.

FORMAT

```
OPTIONS  OBJECT(COBJT,CMP(compool name))  $
```

Explanation:

OPTIONS - keyword indicating an option declaration

OBJECT - this specification directs the compiler to proceed through the code generation phase.

COBJT - is a destination specification for certain outputs. Its use is fully described in other publications.

CMP - specifies that a compool file is to be created.

compool name - name given to the compool. If no name is provided for the compool, the name of the last system data block input will be used.

EXAMPLE

This program generates a compool called HEREPOOL:

```
01    HERE82     SYSTEM $
02               OPTIONS UYK7 $
03               OPTIONS OBJECT(COBJT,CMP(HEREPOOL)) $
04               SYS-INDEX 2 SX $
05    NRDONS     EQUALS 10 $
06               END-HEAD $

07    HEREDD     SYS-DD $
08               VRBL POWER I 6 U $
09               VRBL ROOT A 9 S 0 $
10               VRBL RESULT A 32 S 0 $
11               VRBL(CNTR,CNTR1) I 15 U $
12               VRBL(IND1,IND2,IND3) I 32 S $
13               TABLE NULL H 24 0 $
14    (EXTREF)   ITEM-AREA BUFF24,BUFFSQ $
15               END-TABLE NULL $
16               TABLE DONORS V (H 28) NRDONS $
17               END-TABLE DONORS $
18               END-SYS-DD HEREDD $
19               END-SYSTEM HERE82 $
```

## RETRIEVING COMPOOLS

A compool retrieval declaration is a major header sentence that specifies the names of one or more compools and the ISCM libraries on which they can be found. The minimum required in order to introduce compools is all that is shown. Identification and key convention and the librarian functions other than sel-pool are not discussed. Documentation describing the full use of te CMS-2 librarian is available at all user installations.

FORMAT

```
LIBS internal-id (external-id) $
  SEL-POOL compool name      $
```

Explanation:

LIBS - keyword indicating a library select declaration

SEL-POOL - keyword specifying compool retrieval

The compool retrieval declaration must appear in the major header immediately after the options declaration.

Up to 127 compool retrievals may be specified. The compools are retrieved as they are encountered during the library search. This may not be the order in which their names appear in the retrieval declaration. If more than one compool with the same name is encountered, only the first is retrieved.

Since the effect of compool retrieval is as if the source lines of the original input source were present from the beginning of compilation, the rules pertaining to duplicate names in sys-dd's apply. In particular, the same name may appear no more than once as a normal sys-dd or (EXTDEF) entry, but may appear any number of times as (EXTREF) entries.

The compiler will accept duplicate names in the major headers of the compools provided the declarations are identical. Names appearing in compool major headers may not be duplicated in the major header of the source program used to retrieve the compools as input.

When compools are specified for input to a compile, the compiler first scans the source input to determine what names are actually needed. Then the compools are searched and the required entries are selectively retrieved.

## EXAMPLE

The following program retrieves the compool HEREPOOL to compile two system procedures (HEREPOOL was generated two pages back).

```
RETR82      SYSTEM $
            OPTIONS UYK7 $
            OPTIONS (SM,SCR) $
            LIBS COBJT $
            SEL-POOL HEREPOOL $
            END-HEAD $
SP1         SYS-PROC $
            LOC-DD $
(EXTREF)    TABLE NULL H 24 0 $
(EXTDEF)    ITEM-AREA BUFF24,BUFFSQ $
            END-TABLE NULL $
(EXTREF)    FUNCTION RAISE(ROOT,POWER) A 32 S 0 $
            END-LOC-DD $
            PROCEDURE DOBUFF INPUT BUFF24 OUTPUT BUFFSQ $
            VARY SX FROM 23 THRU 0 BY -1 $
            SET BUFFSQ(SX) TO RAISE(BUFF24(SX),5) $
            END $
            END-PROC DOBUFF $
            END-SYS-PROC SP1 $
SP2         SYS-PROC $
(EXTDEF)    FUNCTION RAISE(ROOT,POWER) A 32 S 0 $
            LOC-INDEX RTX $
            SET RESULT TO ROOT $
            VARY RTX FROM POWER -2 THRU 0 BY -1 $
            SET RESULT TO RESULT*ROOT $
```

```
END $

RETURN(RESULT) $

END-FUNCTION RAISE $

END-SYS-PROC SP2 $

END-SYSTEM RETR82 $
```

Before beginning compool retrieval, the compiler scans the source input RETR82 to determine all the names that are referenced. When an item-area or like-table or sub-table is referenced, the complete definition of the parent table is retrieved whether or not the parent is directly referenced. The compool HEREPOOL is then read and the definitions for all these names that can be found in HEREPOOL are stored in the symbol table.

In this example, names SX, POWER, ROOT, RESULT, NULL, BUFF24 and BUFSQ will be retrieved from HEREPOOL. Data units in HEREPOOL which are not used in this system block RETR82 will not be retrieved from the compool (as variables CNTR, CNTR1, IND1, IND2 and IND3 and table DONORS).

More than one compool may be retrieved for a compilation. An example of multiple compool retrieval is shown below.

```
RETMULT   SYSTEM $
          OPTIONS UYK7 $
          OPTIONS OBJECT(COBJT,SM,SCR) $
          LIBS FCTCP $
          SEL-POOL MATHDD $
          SEL-POOL FORMDD $
          LIBS MONTP $
          SEL-POOL LINKDD $
```

After scanning the source program to determine what names need to be retrieved, the compiler begins retrieval with the files on the library indicated as FCTCP. Two compools have been specified as being on this library. The retrieval process begins with the first of the compools encountered in the library, which may or may not be the first one listed in the source program.

When retrieval from FCTCP is complete, retrieval from library MONTP begins. The process continues until all libraries have been searched for the specified compools.

If the compiler detects duplicate definitions in two of the compools, it will output appropriate error messages. When all the compools have been searched for the names referenced in the source program, the compilation of the source program will continue.

# APPENDIX B

# CONDITIONAL COMPILATION

## Conditional Compilation

Conditional compilation provides a capability to specify blocks of source statements to be compiled only if certain conditions are met.

### 1. CSWITCH BLOCKS

Rarely are any two installations identical in their software requirements, the differences, however, may be quite small in comparison to the overall program. Use of cswitches to control compilation where these differences occur can avoid the necessity of maintaining multiple large systems with only minor variations.

Cswitches may also be useful in narrowing down the amount of code one must deal with when a debugging problem arises. By compiling a test vehicle containing only the specific problem area, the programmer may save considerable time and resources.

The blocks of statements that are subject to conditional compilation are bracketed by a cswitch statement and a cswitch terminal statement.

FORMAT

```
CSWITCH        cswitch flag $

END-CSWITCH    cswitch flag $
```

Explanation:

CSWITCH - keyword indicating the beginning of a conditional compilation block

cswitch flag- a name that will be used to determine if this block is to be compiled

END-CSWITCH - keyword indicating the end of a conditional compilation block. The cswitch flag specifications on the cswitch and end-cswitch statements must be the same.

EXAMPLE

```
LOC-DD $
CSWITCH UYK7 $
TABLE SYMBS V MEDIUM 100 $
FIELD SYMBX I 12 U $
FIELD WORDX I 5 U $
FIELD CHARX I 7 U $
FIELD WDPOS I 3 U $
END-TABLE SYMBS $
END-CSWITCH UYK7 $

CSWITCH UYK20 $
TABLE SYMBS V MEDIUM 50 $
FIELD SYMBX I 11 U $
FIELD WORDX I 6 U $
FIELD CHARX I 7 U $
FIELD WDPOS I 2 U $
END-TABLE SYMBS $
END-CSWITCH UYK20 $
END-LOC-DD $
```

The example above shows two blocks of data declarations, each bracketed by cswitch statements with the flags UYK7 and UYK20, respectively. The bracketed tables are identical except for the sizes of the table and three of the fields. Such differences are frequently dictated by different hardware configurations or different target computers. By bracketing the statements that must differ and selecting the appropriate ones at compile times, a source program can be maintained as a unit even though the compiled versions differ in some details.

## 2. CSWITCH SELECTION

Which cswitch blocks will be compiled is determined by cswitch selection declarations. These selections may appear anywhere in the system block after the options declaration and before the appearance of the cswitch block itself.

**FORMAT**

```
CSWITCH-ON    cswitch flag(s) $

CSWITCH-OFF   cswitch flag(s) $
```

Explanation:

CSWITCH-ON - keyword indicating that the named cswitch blocks are to be compiled

CSWITCH-OFF - keyword indicating that the named cswitches are not to be compiled

CSWITCH-OFF is the default condition. A cswitch block must be selected to be compiled.

**EXAMPLE**

```
CSWITCH-ON UYK 7 $
```

The above selection declaration would cause all cswitch blocks with the flag UYK7 to be compiled when encountered.

## 3. CSWITCH DELETE

Cswitches that have not been selected for compilation may be omitted from compiler listings and source output by using a cswitch delete declaration. A cswitch delete may appear only in a header block.

FORMAT

```
CSWITCH-DEL $
```

CSWITCH-DEL - keyword indicating that all cswitches that do not have a CSWITCH-ON declaration, will be deleted from printouts or outputs

EXAMPLE

```
TAPERD    SYSTEM $
          OPTIONS UYK20 $
          CSWITCH-ON UYK20 $
          CSWITCH-DEL $
          END-HEAD $
```

The example above would cause the cswitch block with the flag UYK20 to be compiled. Since cswitch-delete is specified, the non-compiled cswitch UYK7 would not appear in any listings or outputs.

Rules for the use of cswitches:

(1) The same switch flag may be used for any number of cswitch blocks.

(2) A cswitch whose flag is "off" will appear in source listings unless cswitch delete is specified.

(3) Cswitch blocks may be nested.

(4) Cswitch blocks within another cswitch block whose flag is "off" will not be compiled regardless of their flag condition.

Use this page for notes or comments

# APPENDIX C

## NESTED BLOCKS

## Nested Blocks

The programming manual has introduced three operators (vary, begin and for) that indicate blocks requiring an end statement. To simplify examples, an effort was made to avoid more than one level of nesting. (Nesting is the inclusion of one such block within one or more other blocks).

Operational programs frequently require several levels of nesting. This appendix contains several more complex examples with explanations.

### EXAMPLE

Major league baseball has two leagues, each of which has two divisions of six teams each with twenty-five players per team. An array to hold statistics for all the teams might be defined as follows:

```
TABLE MAJORS A MEDIUM 2,2,6,25 $

    FIELD PLAYER H 32 $
    FIELD ATBATS F 4
    FIELD HITS F $
    FIELD WALKS F 4
    FIELD INTENTWK F $
    FIELD BATAVRG F $
    FIELD POSITION H 2 $

    ITEM AREA ONE $
END-TABLE MAJORS $
```

To calculate the batting averages of each player requires four loops, one within another.

```
      LOC-INDEX L1,L2,L3,L4 $
X1.   VARY L1 THRU 1 ''LEAGUE (AMER. OR NATL.) INDEX'' $
X2.     VRY L2 THRU 1 '' DIVISION INDEX'' $
X3.       VARY L3 THRU 5 ''TEAM INDEX'' $
X4.         VARY L4 THRU 24 ''PLAYER INDEX'' $
              SET MAJORS(L1,L2,L3,L4,BATAVRG) TO
              MAJORS(L1,L2,L3,L4,HITS)/
                MAJORS(L1,L2,L3,L4,ATBATS) $
            END X4 $
          END X3 $
        END X2 $
      END X1 $
```

The innermost loop block X4 is executed twenty-five times for each execution of the next level loop X3. X3 in turn is executed six times for each execution of X2, which is executed two times for each execution of X1. X1 is executed two times.

Labels on the vary statements were not required, but are helpful in keeping track of whether or not all necessary end statements have been provided. Whenever the vary statement contains as label, the end statement must contain that name.

The following system procedure illustrates how different kinds of blocks requiring end statements may be nested. Examine the statements as they appear and then examine them gain after reading the comment statements that follow the sys-proc. These statements, inserted just before or after the procedure declaration would provide the documentation that each programmer should provide as an aid for others to understand his program.

```
BATLEADR    SYS-PROC $
            LOC-DD $
(EXTREF)    VRBL MINREQ I 7 $
(EXTREF)    TABLE MAJORS A MEDIUM 2,2,6,25 $
            FIELD PLAYER H 32 $
            FIELD ATBATS F $
            FIELD HITS F $
            FIELD WALKS F $
            FIELD INTENTWK F $
            FIELD BATAVRG F $
            FIELD POSITION H 2 $

(EXTREF)    ITEM-AREA ONE $
            END-TABLE MAJORS $
(EXTREF)    PROCEDURE PRINTIT INPUT ONE $
            END LOC-DD $
            PROCEDURE BATLEADR $
            LOC-INDEX L1,L2,L3,L4 $
LEAGLOOP.   VARY L1 THRU 1 ''0 REPRESENTS NATIONAL, 1 AMERICAN'' $
DIVLOOP.    VARY L2 THRU 1 ''0 FOR EASTERN DIV., 1 FOR WESTERN'' $
            SET ONE(BATAVRG) TO 0 ''INITIALIZE ON EACH PASS'' $
TEAMLOOP.   VARY L3 THRU 5 $
```

```
PLYRLOOP.   VARY L4 THRU 24 $
              FOR MAJORS(L1,L2,L3,L4,POSITION) $
                BEGIN H(CF),H(LF),H(RF)''OUTFIELD POS.  ONLY'' $
                IF MAJORS(L1,L2,L3,L4,BATAVRG) GT ONE(BATAVRG)
                  THEN BEGIN ''CHECK MINIMUM MET'' $
                    IF MAJORS(L1,L2,L3,L4,ATBATS)+
                      MAJORS(L1,L2,L3,L4,INTENTWK) GTEQ
                      MINREQ THEN SET ONE TO MAJORS(L1,L2,L3,L4,) $
                      END ''CHECK MIN MET'' $
                    END ''OUTFIELD ONLY'' $
                  END ''FOR'' $
                END PLYRLOOP $
                END TEAMLOOP $
              PRINTIT INPUT ONE ''FOUND LEADER THIS DIVISION,PRINTIT
               '' $
                END DIVLOOP $
              END LEAGLOOP $
              END-PROC BATLEADR $
              END-SYS-PROC BATLEADR $

COMMENT     THIS PROCEDURE SEARCHES ARRAY MAJORS TO FIND WHICH
            OUTFIELDER IN EACH DIVISION OF EACH LEAGUE HAS THE
            HIGHEST BATTING AVERAGE.  $
COMMENT     ONLY PLAYERS WHOSE COMBINED TOTAL OF ATBATS AND
            INTENTIONAL WALKS RECEIVED MEETS THE REQUIRED MINIMUM
            ARE CONSIDERED $
COMMENT     WHEN EACH DIVISION SEARCH IS COMPLETE PROCEDURE PRINTIT
            IS CALLED TO OUTPUT THE INFORMATION IN ITEM-AREA ONE,
            WHICH IS THE LEADER FOR THAT DIVISION.  $
```

A philosophical note on the difficulty of handling nested blocks:

Some psychologists and anthropologists hold that humanity's natural mode of counting is ''one-two-many''. Experience indicates that effective action by a committee decreases exponentially with each member above the number of three. The ability to follow nested coding of level three and higher seems to agree with the foregoing statements.

The use of notes, comments and any other aid available to document the logic is especially critical in complex blocks of nested coding.

The pressure of deadlines can easily tempt a programmer to push through a programming assignment without "wasting" time on annotation. To avoid giving in to this temptation, the programmer should remember that he or she may not have escaped to greener pastures when someone decides that that program must be modified.

# APPENDIX D

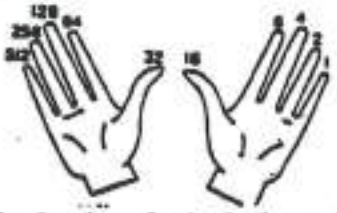
# REVIEW OF SELECTED BASICS

## 1. CONVERTING NUMBERING SYSTEMS

### A. BINARY NUMBERING SYSTEM

Counting on the fingers has a definite bearing on the method of counting for computers. Since the fingers are a basic counting device, every principle that applies to the fingers also applies to more sophisticated counting devices.

It can be mathematically proven that the most efficient code that can be developed for the fingers is one that doubles the value of each finger.



This code is often called the 8-4-2-1 code after the four lowest numbers in the sequence. This code can be used to count from zero to a maximum of 1,023. Including the zero, 1024 values can be represented.

512+256+128+64+32+16+8+4+2+1=1,023

Modern digital computers also use the 8-4-2-1 code. Nearly all computers work in the binary mode. This is a base-2 system utilizing only two digits, zero and one. This is most convenient for computers because an electrical field may be "on" or "off" and a magnetic device may be magnetized or not magnetized. These are also base-2 types of actions. Since computers use binary circuits, the internal arithmetic of computers is binary in nature.

The importance of numbering systems other than decimal is not immediately apparent to most people. We are so accustomed to using the decimal system that it has become almost second nature to us, while other numbering systems seem strange and difficult. They are difficult only because they are strange.

In reality, the two systems to be discussed (binary and octal) are quite important to computer programmers, and, after a bit of study, the student will realize that each system is as simple as the decimal system. The new ground rules must be understood before the systems fall into place in a logical manner.

---

Binary and Octal Numbering Systems courtesy of "Basic Principles of Data Processing". Saxon and Steyer, Prentice-Hall, Inc. 1970.

In the binary system, only two digits are used, zero and one. It requires the invention of a code (using only zero and one) that will cover all possible combinations of numbers to have a workable system. An arbitrary code could easily be devised, but we want a very efficient code and for this reason the 8-4-2-1 code will be utilized. Details of this code and some of the methods for using it will be covered in the following pages.

## Counting in the binary system

The 8-4-2-1 code described on the previous page is, in fact, the binary code. It utilizes just two digits, zero and one. In such a two-state code, the one is arbitrarily chosen as the on condition and the zero as the off condition. Therefore, only the ones will be counted in a sequence of binary numbers.

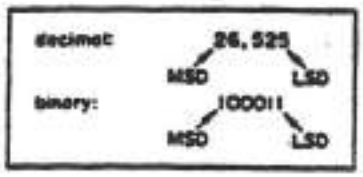
The location of each of the ones (based on the 8-4-2-1 code) is the key to its value, as shown in the table below. It is

| Code value → | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Count only ones |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | = 20 (16 + 4) |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | = 37 (32 + 4 + 1) |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | = 27 (16 + 8 + 2 + 1) |

Code values of the 8-4-2-1 code.

important to note that the values given to each binary position start from the right and progress to the left. (If this sequence were reversed, the results would be entirely different.) The rightmost position, then, is the position of the least-significant digit (LSD) and the leftmost position is that of the most-significant digit (MSD). Notice that this is exactly the same as in the decimal numbering system, in which the leftmost position is the most significant.

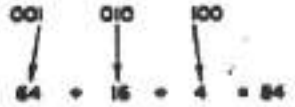
A number of other codes based on the binary system are possible, and many such codes are used for computers. The 8-4-2-1 code described above is usually called the pure binary code.




Position significance of numbers.

Binary compared to decimal

Since binary numbers tend to be extremely long (roughly 3.3 times longer than their equivalent decimal numbers), it is more convenient to group them in threes. This does not change the relative value of each position but simply makes it easier to read.




Counting in the binary system is as follows:

| Decimal | Binary | Decimal | Binary |
|---|---|---|---|
| 0 | 000 | 5 | 101 |
| 1 | 001 | 6 | 110 |
| 2 | 010 | 7 | 111 |
| 3. | 011 | 8 | 001 000 |
| 4 | 100 | 9 | 001 001 |

Since the binary system only contains 0 and 1, it is necessary to take the same "move" at 2 that is taken at 10 in the decimal system. This is to place a "1" to the left and start again (at the right) with "0." Therefore, a decimal 2 is a binary 10, 3 is 11, and then another shift must be made, adding 1 to the left and starting again with 0.

EXAMPLES

Convert the following decimal numbers to binary:

1. In the 8-4-2-1 sequence, on the preceding page, find the number just less than the value of the one to be converted. Start with this number, and continue adding until the required number is reached.

22=010 110

16+4+2=22

2. Add zeros to the left of the MSD to complete this last (leftmost) group of three binary digits.

76=001 001 100

64+8+4=76

## B. OCTAL NUMBERING SYSTEM

We have said that the binary system is a base-2 system. The octal numbering system is a base-8 system and is very convenient to use as a shorthand to binary.

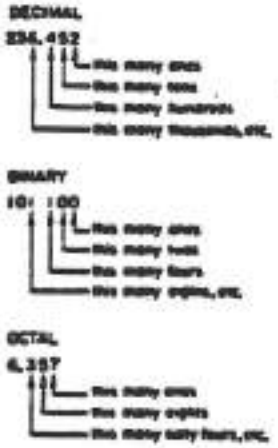
Since it is a base-8 system, it will utilize only the numerals 0 through 7. Counting in this system is as shown below (notice that 8 and 9 are never used):

| DECIMAL | OCTAL | DECIMAL | OCTAL |
|---|---|---|---|
| 0 | 0 | 8 | 10 |
| 1 | 1 | 9 | 11 |
| 2 | 2 | 10 | 12 |
| 3 | 3 | 11 | 13 |
| 4 | 4 | 12 | 14 |
| 5 | 5 | 13 | 15 |
| 6 | 6 | 14 | 16 |
| 7 | 7 | 15 | 17 |

Decimal to octal conversion.

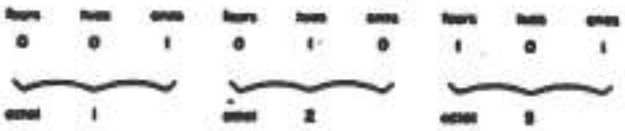
When writing a number in octal, it is usual to designate the system in the following manner: $376_8$. The subscript 8 indicates that 376 is an octal number.

As in the decimal and binary systems, the value of each digit in a sequence of numbers is definitely fixed.



Conversion between octal and binary

The relationship between octal and binary is so simple that conversion may be made instantaneously. Consider every binary number in groups of threes (ØØ1Ø1Ø1Ø1=ØØ1 Ø1Ø 1Ø1). Now, each grouping of three binary digits is identified by ones, twos, and fours positions, and these are used to convert to octal.



Conversely, each octal number is converted to three binary numbers as shown below. The binary representation of the octal numbers is often called binary-coded octal.

Octal to binary conversion.

| OCTAL | BINARY |
|---|---|
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

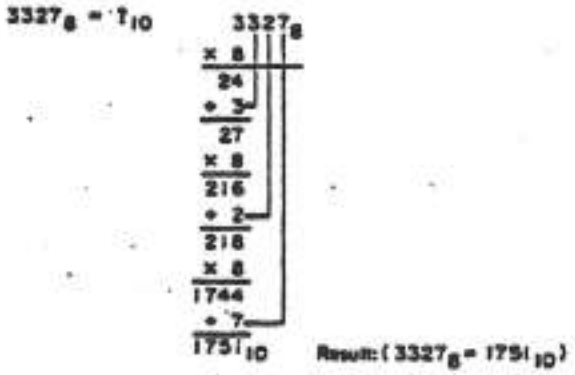
Converting from octal to decimal

This is usually accomplished by looking up the number in a conversion table. It may be accomplished manually in the following manner:

Multiply each octal position in turn by eight, starting with the leftmost position. Then add the next number to the result, multiplying the sum by eight, and continue until the last digit is reached. This one is not to be multiplied.

## EXAMPLES

1.




2.


Converting from decimal to octal

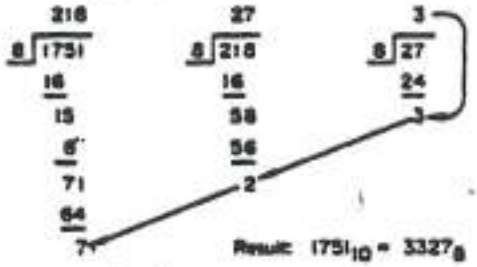
This procedure is also generally accomplished by checking a conversion table, but it may be done manually in the following manner:

Successively divide the decimal figure by eight, until no further division is possible. The octal result will be the last quotient figure, followed by each of the remainders, starting from the last and finishing with the first.

## EXAMPLES

1.

$1751_{10} = ?_8$

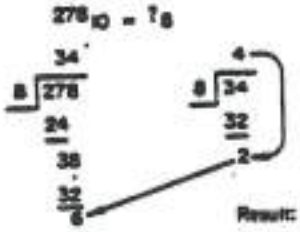



Result: $1751_{10} = 3327_8$

---

2.

$278_{10} = ?_8$

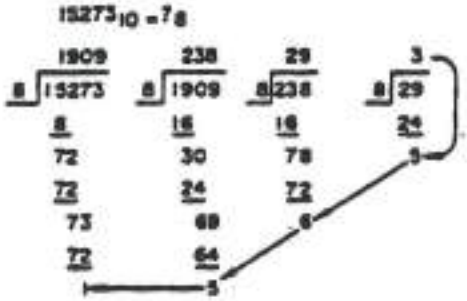



Result: $278_{10} = 426_8$

---

3.

$15273_{10} = ?_8$




Result: $15273_{10} = 35651_8$

## Converting from decimal fractions to octal fractions

To convert decimal fractions to octal fractions, multiply successively by 8. The integer portion of each result (to the left of the decimal point) becomes part of the octal fraction, starting from the first and finishing with the last. Only the resulting fractional portion is multiplied in each successive step.

### EXAMPLES

1. Convert .025 to octal

| Step 1 | Step 2 | Step 3 |
|---|---|---|
| .025 | .200 | .600 |
| x8 | x8 | X8 |
| [0].200 | [1].600 | [4].800 |

Result: $.025_{10} = .014_8$

2. Convert .1036 to octal

| Step 1 | Step 2 | Step 3 | Step 4 |
|---|---|---|---|
| .1036 | .8288 | .6304 | .0432 |
| x8 | x8 | x8 | x8 |
| [0].8288 | [6].6304 | [5].0432 | [0].3456 |

Result: $.1036_{10} = .0650_8$

3. Convert .061 to octal

| Step 1 | Step 2 | Step 3 |
|---|---|---|
| .061 | .488 | .904 |
| x8 | x8 | x8 |
| [0].488 | [3].904 | [7].232 |

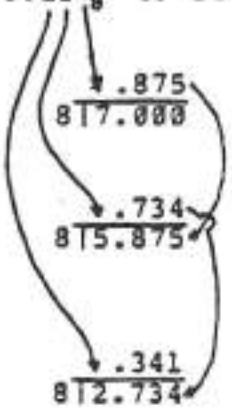
Result: $.061_{10} = .037_8$

## Converting from octal fractions to decimal fractions

To convert octal fractions to decimal fractions, begin working from right to left. Divide the rightmost digit by 8, ignoring the remainder. To this result add the next digit from the fraction placing it into the integer portion. Divide again and proceed in the same manner until all digits in the fraction have been used. The final division result is the desired decimal fraction.

## EXAMPLES

1. Convert $0.257_8$ to decimal



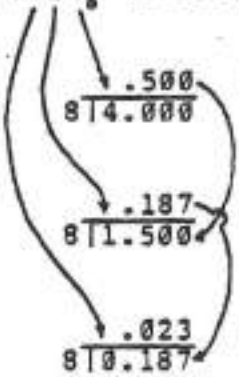

Step 1 - divide rightmost digit by 8

Step 2 - bring down next digit to integer position and result of Step 1 to fraction portion

Step 3 - continue as in Step 2 - this result is the decimal fraction

Result: $0.257_8 = .341_{10}$

2. Convert $.014_8$ to decimal




Step 1

Step 2

Step 3

Result: $.014_8 = .023_{10}$

Note: Most decimal fractions do not have exact equivalents in octal, and vice versa. Therefore, the conversion of fractional data from one base to another is an approximation.

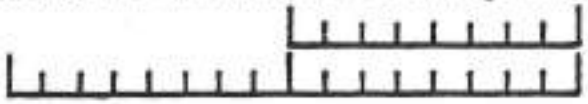
## 2. BITS AND BIT HANDLING

### A. BITS, BYTES AND WORDS

In Chapter 1 it was briefly mentioned that modern digital computers operate in binary mode. This means (among other things) that the instructions that direct computer operations and the data used are represented within the computer as groups of binary digits, called bits.

A bit (binary digit) is the smallest unit of information used in the computer. It is represented by a zero (off) or a one (on).

A byte consists of eight bits in consecutive sequence. Bytes are successive and do not overlap each other in computer memory.




The basic unit that can be directly referenced by the programmer is called a word. A word is usually some multiple of a byte and the size of the word varies with the computer being used. Some computers allow the programmer to reference half-word, double-word, or even byte.

The programmer must know what is the word size and what may be referenced in the particular computer with which he will be working. For example, the AN/UYK-7 computer uses a 32-bit word.

One word = 4 bytes

It allows the referencing of byte (8 bits), half-word (2 bytes), word (4 bytes) and double-word (8 bytes).

It must be remembered that all numbering of bits is from right-to-left beginning with zero; therefore a 32-bit word would be numbered from 0 through 31.

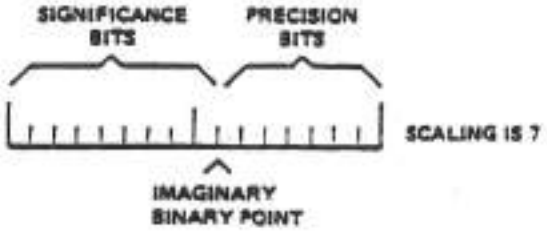
|7|6|5|4|3|2|1|0| bits numbered

### B. COMPUTING THE NUMBER OF BITS IN A DATA UNIT

In the CMS-2Y language, data declarations sometimes call for the programmer to specify the number of bits the data unit will require. The language also provides the capability of specifying the location of an imaginary binary point in certain data units.

This allows the use of fractional data where hardware floating-point operation is not available or is not adequate to the task.

The integer bits (i.e. bits to the left of the imaginary binary point) are sometimes referred to as significance bits. The fractional bits (i.e. bits to the right of the imaginary binary point) are sometimes called precision bits. The number of fractional bits in a data unit are also referred to as scaling of the data unit.




It is desirable that the number of bits specified for a data unit be a true indication of the values the data unit may be expected to assume.

To find the required number of bits:

1. determine the range of values the data unit may have
2. consider the integer and fractional (if any) parts separately
    a. Integer part
        1. using absolute values, determine the maximum value and convert it to octal
        2. determine the number of bits required to represent the value in binary
    b. Fractional part
        1. using absolute values, determine the minimum non-zero value and convert it to octal
        2. determine the number of bits required to represent the value in binary.

Note: Care should be taken in determining the minimum value. For instance, if the range of values is 0.23 through 0.79, the minimum value is 0.23. However, if the range is -0.23 through 0.79, the minimum non-zero value is 0.01.

3. to arrive at a total, combine the two figures, and add a sign-bit if needed
    a. add the number of interger bits and the number of fractional bits. This gives the number of value containing bits (magnitude).
    b. if the data may have negative as well as positive values, add one bit (sign-bit) to contain the sign.

EXAMPLES

| Range of Values | Determination |
|---|---|
| 0 through 1,249 | $1249_{10}$ =$2341_{8}$ =10 011 100 001$_{2}$ =11 bits |
| -350 through 125 | $350_{10}$ =$563_{8}$ =101 110 011$_{2}$ +sign bit=10 bits |
| -3.25 through 17.75 | (1) $17_{10}$ =$21_{8}$ =10 001$_{2}$ =5 bits<br>(2) $.01_{10}$ =$.005_{8}$ =.000 000 101$_{2}$ =9 bits<br>(3) 5 integer bits+9 fractional bits+sign bit=15 bits |

## C. ALIGNING BINARY POINTS AND SCALING

Data transfers, arithmetic operations and evaluation of numeric relational expressions may require data manipulation to align binary points before performing the operation. The method of aligning is to load the data into one of the computer's registers (or two adjacent registers if the bit size exceeds one computer word) and to shift the data left or right as required. During this process bits may be lost (may be shifted out of the register or the result of an operation may overflow).

The detailed algorithms by which the compiler determines scaling may be found in the programmer's reference manuals for each of the computers using the CMS-2Y language. The following is a brief overview.

When data from a sequence of arithmetic operations is to be assigned to some destination data unit, that data unit's scaling determines the final scaling. Intermediate alignment of operands is as follows:

1. Addition and Subtraction

The operands are aligned to the smaller of their respective scalings, unless that is larger than the scaling of the destination. In that case, they are aligned to the scaling of the destination.

Note: When the smaller scaling is referred to, it means the smaller non-zero scaling. If one operand has zero scaling, the scaling of the other prevails.

2. Multiplication

a. If either operand's bit size (including sign bit) exceeds the word size of the target computer, the operation will be performed in floating-point (if available). If floating-point is not available, the compiler will reject the operation and print an error message.

b. If either operand is being held in the computer's registers as the result of a previous multiplication, and its present scaling is larger than the scaling of the destination, then it is aligned to the scaling of the destination before the multiplication occurs. Otherwise no alignment occurs.

c. The scaling of the result equals the sum of the scalings of the operands.

3. Division

a. If the divisor exceeds the word size of the target computer, the operation will be performed in floating-point (if available). If floating-point is not available, the compiler will reject the operation and print an error message.

b. Operands are aligned so that the scaling of the result equals the scaling of the destination.

4. Exponentiation

The scaling of the result equals the original scaling of the operand.

Evaluation of fixed-point expressions appearing as part of a relational expression is somewhat different, since there is no destination scaling to consider.

a. Operations may be performed in floating-point under the same conditions as above. If floating-point is used, the numeric comparison will also be made in floating-point.

b. No alignment occurs for multiplication.

c. Addition, subtraction ad division are aligned to give a result having the same scaling as the smaller scaling of the two operands.

## 3. BASICS OF BOOLEAN ALGEBRA

### A. GENERAL BACKGROUND

A knowledge of Boolean algebra is necessary for the understanding of the relationship between the logic of computers and the circuitry incorporated in most computers.

In 1854, George Boole published a book titled An Investigation of the Laws of Thought On Which are Found the Mathematical Theories of Logic and Probabilities. His intention was to perform a mathematical analysis of logic. This book was the start of the algebra that bears his name.

Boolean algebra differs considerably from the rules of elementary algebra. Confusion between the two forms may be avoided by considering Boolean algebra from the point of view of switching circuits. In 1938, Claude Shannon first used Boolean algebra on switching circuit problems. He developed a mathematical method of depicting circuits that consisted of switches and relays. His methods were almost universally accepted and are still in use today.

The real advantage of Boolean algebra to the computer designer is that the circuitry used can, with this method, be simplified and expressed in mathematical notation instead of in bulky circuit diagrams. A basic knowledge of Boolean algebra is essential in the computer field.

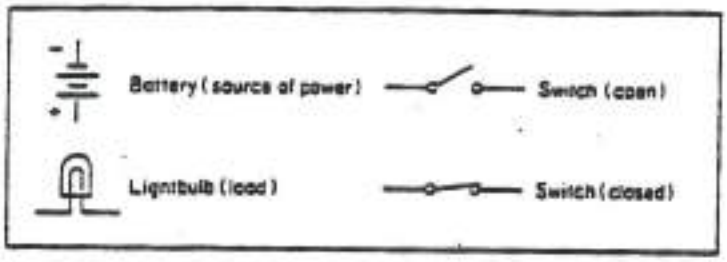
A number of terms and symbols must be learned by the student who has no background in Boolean algebra. He must also be introduced to basic circuits. Additionally, he must understand that the variables used in Boolean equations have the unique characteristic of being able to assume only one of two possible values: zero and one. No matter how many variables there may be in an equation describing a logical circuit, each variable can only have the value of 0 or the value of 1.

### B. BASIC CIRCUITRY

A drawing of an electrical circuit is called a schematic diagram. Symbols are used to represent different parts of the circuit. A few commonly used symbols are shown on the following page.

---

Basics of Boolean Algebra courtesy of "Basic Data Processing Mathematics", James A. Saxon, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

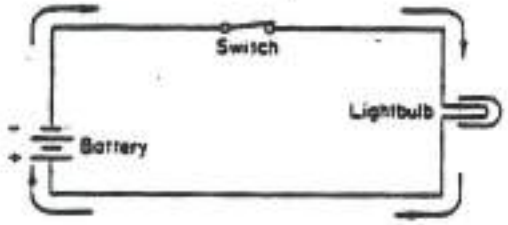



Circuitry symbols.

Most electrical circuits consist of a source (source of power), a switch, and a load. A load is any component that changes electricity into a useful effect. In the example below, the light bulb is the load because the bulb will glow and give off useful light.

EXAMPLE

A complete circuit must have an unbroken path for the current to flow from the negative side of the source through the load and back to the positive side of the source. A switch is used to break the circuit.

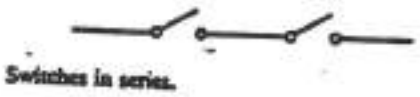



Current will flow from the source (battery), through the closed switch, through the load (light bulb), and back to the source. If the switch is open, current cannot flow. Therefore, we can say that output (the light bulb) will be a 1 if the switch is closed and a 0 if the switch is open.

## C. SERIES AND PARALLEL CIRCUITS

It is possible for more than one switch to be used in a circuit. If two or more switches are used, they may be connected in series or in parallel.

If switches are connected end-to-end, they are connected in series, as shown below:

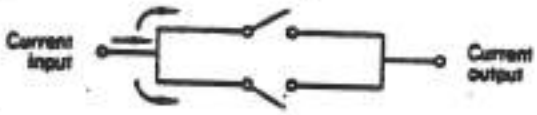


Switches in series.

It is obvious that both switches must be closed for current to flow through the circuit. If only one switch is closed, current will not be able to flow through the other switch, which is open.

If switches are connected side-by-side, they are connected in parallel, as shown in the figure below.

An examination of this example shows that if either switch is closed, current will flow from the input to the output. Both of these concepts are extremely important to the study of Boolean algebra.




Switches in parallel

EXAMPLES

Now we will convert our knowledge of series and parallel circuits to 0 (for no flow of current) and 1 (for flow of current).

How many possible combinations are there for two switches connected in series?

1. Both switches can be open - output 0 (no current).

2. First switch can be open and second one closed - output 0 (no current).

3. Second switch can be open and first one closed - output 0 (no current).

4. Both switches can be closed - output 1 (current flows).

What are the possibilities for two switches connected in parallel?

1. Both switches can be open - output 0.

2. First switch can be open, second one closed - output 1.

3. Second switch can be open, first one closed - output 1.

4. Both switches can be closed - output 1.

For a series circuit, then, the only time that current can flow is if both switches are closed. This leads us to the following truth table (for a series circuit), Table 1, 0 referring to open and 1 referring to closed switches.

In a parallel circuit, the only time that current cannot flow is if both switches are open. This leads us to the following truth table (for a parallel circuit), Table 2.

**Table 1. Truth Table for Switches in Series**

| *Switch 1* | *Switch 2* | *Output* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table 2. Truth Table for Switches in Parallel**

| *Switch 1* | *Switch 2* | *Output* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## D. BOOLEAN SYMBOLOGY

Everyone is familiar with the normal algebraic sign (+) for addition and the dot product sign ( ) for multiplication.

A + B = C  A B = C  or  AB = C
5 + 6 = 11  5 X 4 = 20

In Boolean algebra, these symbols take on entirely different meanings.

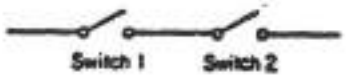
The symbol "+" is defined as "OR." Using switches A and B as examples, we can say that either one or the other or both must be closed for the circuit to conduct current. This is known as the inclusive OR. It is shown in Boolean symbology as: A + B (read as "A or B").

The symbol " " is defined as "AND." This symbol refers to both or all variables connected with the AND sign. If the sign is not there, it is still understood to be AND (e.g., A B may be written AB and still means A AND B).

Note: It is an unfortunate fact that there is considerable variance in the use of symbology throughout the mathematical community. This is particularly true of the symbols used for the logical AND and OR. The symbols used in this text are felt to be the most commonly accepted by a majority of mathematicians.

To understand these new uses for known symbols, let us return to the circuits for examples.

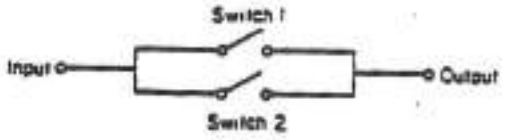
The AND operation can be represented by switches in series:




If the switches are open, the output is "0." If the switches are closed, the output is "1." It makes no difference how many switches there are in such a circuit. For current to flow through the circuit, both switch 1 AND switch 2 must be closed. Under any other condition, the output will be "0."

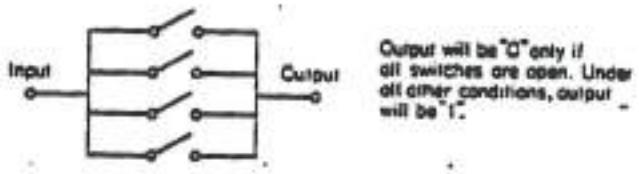
From this we can see that the output of an AND component in a circuit (called a gate) is a 1 only if all inputs are 1's. In this condition, the gate is considered to be enabled. If it is not in this condition (all 1's in; 1 out), the gate is considered to be disabled.

The OR operation can be represented by switches in parallel:




If any switch is closed, the output will be 1. It is called an OR component because when switch 1 OR switch 2 or both switches are closed, the output will be 1. Output will only be 0 if all switches are open.

The number of switches (inputs) has no effect on the result.




Symbolic representations of AND and OR gates are shown below:

# APPENDIX E

## CMS-2 COMPILER RESERVED WORD LIST

## CMS-2 Compiler Reserved Word List

The following list contains all of the reserved words in the CMS-2Y language. These words should not be used as programmer devised identifiers (names). Additionally, identifiers starting with RT should not be used because all available support routines use identifiers starting with RT; thus a duplicate identifier may occur causing compile problems.

| | | | | |
|---|---|---|---|---|
| ABS | DEBUG | FORMAT | OCM | STOP |
| ALG | DECODE | FROM | ODDP | SWAP |
| AND | DEFID | FSUB | OPEN | SWITCH |
| ANDF | DENSE | FUNCTION | OPTIONS | SYSTEM |
| BASE | DEP | GOTO | OR | TABLE |
| BEGIN | DIRECT | GT | ORF | TDEF |
| BIT | DISPLAY | GTEQ | OUTPUT | THEN |
| BY | E | H | OVERFLOW | THRU |
| CAT | ELSE | HEAD | OVERLAY | TO |
| CHAR | ENCODE | IF | PACK | TRACE |
| CHECKID | END | INDIRECT | POS | UNTIL |
| CIRC | ENDFILE | INPUT | PRINT | USING |
| CLOSE | EQ | INTO | PTRACE | VALID |
| CMODE | EQUALS | INVALID | PUNCH | VARY |
| CNT | EVENP | LENGTH | RANGE | VARYING |
| COMMENT | EXCHANGE | LIBS | READ | VRBL |
| COMP | EXEC | LOG | REGS | WHILE |
| COMPF | EXIT | LT | REM | WITH |
| CONF | FADD | LTEQ | RESUME | WITHIN |
| CORAD | FDIV | MEANS | RETURN | XOR |
| CORRECT | FIELD | MEDIUM | SAVING | XORF |
| CSWITCH | FIL | MODE | SCALF | |
| D | FILE | NITEMS | SET | |
| DATA | FIND | NONE | SHIFT | |
| DATAPOOL | FMUL | NOT | SNAP | |
| | FOR | O | SPILL | |

# APPENDIX F

## INTRODUCTION OF KEYWORDS

## Introduction of Keywords

| CHAPTER | KEYWORD | PAGE |
|---|---|---|
| 1 | COMMENT | 10 |
| 2 | VRBL | 15 |
| | SET | 24 |
| | TO | 24 |
| 3 | GOTO | 46 |
| | EQ | 47 |
| | NOT | 47 |
| | LT | 47 |
| | GT | 47 |
| | LTEQ | 47 |
| | GTEQ | 47 |
| | IF | 48 |
| | THEN | 48 |
| | ELSE | 48 |
| | BEGIN | 58 |
| | END | 58 |
| 4 | TABLE | 66 |
| | NONE | 66 |
| | MEDIUM | 66 |
| | DENSE | 66 |
| | FIELD | 68 |
| | END-TABLE | 69 |
| | ITEM-AREA | 74 |
| | LIKE-TABLE | 75 |
| | SUB-TABLE | 79 |
| 5 | VARY | 90 |
| | FROM | 90 |
| | THRU | 90 |
| | BY | 90 |
| | WHILE | 94 |
| | UNTIL | 94 |
| | RESUME | 98 |
| 6 | FUNCTION | 108 |
| | RETURN | 108 |
| | END-FUNCTION | 108 |
| | PROCEDURE | 114 |
| | END-PROC | 114 |
| | INPUT | 115 |
| | OUTPUT | 116 |
| | EXIT | 116 |

# INDEX