

Table Handling

SUBSCRIPTING	1
A 2-D Table	2
A 3-D Table	2
Loading a Table	4
The INITIALIZE Statement	5
The Replacing Option	6
Initialising fixed length Tables	6
Restrictions on the use of INITIALIZE	6
Variable length Tables	7
Using Subscripts	8
The Compiler option SSRANGE	9
Indexing	10
Indexing v's Subscribing	12
Looking up data in Tables	13
Solving Dumps involving Tables	14
Exercise 1	16



Table Handling

SUBSCRIPTING

If a data item is repeated several times, e.g. 5 item codes, the following description can be used:

```

01      ITEM RECORD.
      03      ITEM-CODE-1  PIC S9 (7)    COMP-3.
      03      ITEM-CODE-2  PIC S9 (7)    COMP-3.
      03      ITEM-CODE-3  PIC S9 (7)    COMP-3.
      03      ITEM-CODE-4  PIC S9 (7)    COMP-3.
      03      ITEM-CODE-5  PIC S9 (7)    COMP-3.

```

If there were 100 item codes this would clearly be very inefficient.

We can use subscripting to overcome this problem.

```

01      ITEM-RECORD.
      03      ITEM-CODE     PIC S9 (7)    COMP-3    OCCURS 100 TIMES.

```

N.B. The OCCURS clause cannot be used on 01 or 77 level fields.

Within the COBOL program you need to be able to specify which occurrence of ITEM-CODE is to be referenced. To do this a SUBSCRIPT is used.

The SUBSCRIPT is a binary field, usually 4 bytes long, which will contain the number of the occurrence to be used.

It is defined in Working-Storage as:

```

03      ITEM-SUBSCRIPT          PIC S9 (9)    COMP    VALUE +1.

```

You should ensure that the subscript does not contain a value outside the number of occurrences. In our example not 0 or > 100.



A 2-D TABLE

If the repeated information is a group of fields the description of the table may look like:

```
01      ITEM-RECORD.  
      03      ITEM-INFO OCCURS 100 TIMES.  
              05      ITEM-CODE          PIC S9(7)      COMP-3.  
              05      ITEM-DESCRIPTION   PIC X(24) .  
              05      ITEM-PRICE        PIC S9(3) V99 COMP-3.
```

A 3-D TABLE

```
01      STORE-RECORD.  
      03      STORE-INFO OCCURS 1000 TIMES.  
              05      STORE-NO   PIC S9 (5)      COMP-3.  
              05      STORE-STUFF      PIC X.  
              05      STORE-SALES     PIC S9 (5) V99 COMP-3 OCCURS 12 TIMES.
```

This definition provides for a store number and suffix followed by 12 monthly sales figures. The whole group repeated 1000 times.

In this case two subscripts need to be defined in Working-Storage,

```
01      PROGRAM-SUBSCRIPTS.  
      03      ST-SUBSCRIPT  PIC S9 (9)      COMP  VALUE +1.  
      03      ST-SALES-SUB PIC S9 (9)      COMP  VALUE +1.
```

To COPY an item from a table to, say, a report, you code

```
MOVE ITEM-CODE(integer) TO REP-ITEM-CODE.
```

or

```
MOVE ITEM-CODE(subscript) TO REP-ITEM-CODE
```

For example:

to copy the third item code

```
MOVE ITEM-CODE(3) TO REP-ITEM-CODE.
```

or

```
MOVE ITEM-CODE(ITEM-SUBSCRIPT) TO REP-ITEM-CODE.
```

where ITEM-SUBSCRIPT is defined in Working-Storage and has been assigned the value 3 in the program.

The second of these methods is usually coded as this allows flexibility in which occurrence is required.

LOADING A TABLE

VALUE clauses can not be used in table definitions so a table must be loaded with information during the run of the program. This will generally be done by using the

PERFORM VARYING statement.

For Example:

The Store Table described on page 2 may be loaded from the records on the Master Store File. The file would be read, one record at a time, and entries made in the table for each store:

```
. . . . .
. . . . .
READ MASTER-STORE-FILE
  AT END
    MOVE 'Y' TO ST-EOF.
PERFORM LOAD-STORE-TABLE VARYING ST-SUBSCRIPT
  FROM 1 BY 1
  UNTIL END-OF-STORES
    OR ST-SUBSCRIPT > 1000.

. . . . .
. . . . .
LOAD-STORE-TABLE.
  MOVE MSF-ST-NO      TO STORE-NO (ST-SUBSCRIPT).
  MOVE MSF-ST-SUFF    TO STORE-SUFF (ST-SUBSCRIPT).
  PERFORM LOAD-SALES VARYING ST-SALES-SUB
    FROM 1 BY1
    UNTIL ST-SALES-SUB > 12

  READ MASTER-STORE-FILE
    AT END
      MOVE 'Y' TO ST-EOF

LOAD SALES.
  MOVE ST-SALES-SUB TO MSF-SALES-SUB.
  MOVE MSF-SALES (MSF-SALES-SUB)
    TO STORE-SALES (ST-SUBSCRIPT, ST-SALES-SUB).
```

You can see from this example that the Store Table has 2 subscripts, which must be defined in Working-Storage: one to reference the different stores (ST-SUBSCRIPT) and one to access the monthly sales for the store (ST-SALES-SUB).

The Master Store File record description would also contain a table definition for the 12 occurrences of the monthly sales figure. These sales figures are referenced via the subscript called MSF-SALES-SUB.

The INITIALIZE Statement

General Format:

INITIALIZE identifier-1 (REPLACING data-type BY { identifier-2
literal-1 })

The INITIALIZE statement offers an alternative to the MOVE statement in initialising data items to fixed values. The following examples illustrates the simple form of INITIALIZE:

```
05      RECORD-COUNTER          PIC 9 (5)

INITIALIZE RECORD-COUNTER.
```

In this example, the data item, RECORD-COUNTER, is defined as numeric. The INITIALIZE statement causes it to be set to zero.

The subject of an INITIALIZE statement can also be data items defined as alphanumeric, alphabetic, alphanumeric edited or numeric edited. The following example illustrates this point:

```
01      GROUP-ITEM.
03 FIELD-1          PIC X (4).
03 FIELD-2          PIC A (3).
03 FIELD-3          PIC BBBXX.
03 FIELD-4          PIC Z9.99.
03 FIELD-5          PIC S9 (5) V99 COMP-3.
03 FIELD-6          PIC S9 (4) COMP.

INITIALIZE          FIELD-1 FIELD-2 FIELD-3 FIELD-4 FIELD-5
FIELD-6.
```

In this example FIELD-1 FIELD-2 and FIELD-3 are initialised to SPACES, while FIELD-4 FIELD-5 and FIELD-6 are set to zero.

The INITIALIZE statement also works to initialise a group item. For example, the following INITIALIZE statement will have the same effect as the one on the previous page:

```
INITIALIZE GROUP-ITEM.
```

Unless you specify otherwise, alphanumeric, alphabetic and alphanumeric edited items are initialised to SPACES. Numeric and numeric edited fields are initialised to zero.

The REPLACING option

The REPLACING option for the INITIALIZE statement lets you initialise data items to values other than SPACES or zeros. For example, the following statement initialises a counter to 1:

```
05 PAGE-COUNT PIC 99.  
INITIALIZE PAGE-COUNT REPLACING NUMERIC DATA BY 1.
```

This option is also available on group items and tables.

Initialising fixed length tables

For example,

```
01 A-TABLE.  
03 TABLE-INFO OCCURS 5 TIMES.  
05 TABLE-NAME PIC X (20).  
05 TABLE-NUMBER PIC 9 (5).  
...  
INITIALIZE A-TABLE.
```

Restrictions on the use of INITIALIZE

- 1) INITIALIZE cannot be used to change the value of an index data item (see page 10).
- 2) You cannot use INITIALIZE to initialise a variable-length table (see page 7).
- 3) If you initialise a group item, subordinate FILLER items will not be initialised. Therefore, INITIALIZE is not appropriate for initialising a print line to SPACES.

VARIABLE LENGTH TABLES

Not all tables have a set number of occurrences. Consider a file which contains a weekly sales total for a store. At the beginning of the financial year there are only a few sales totals while towards the end of the year there may be up to 52. Rather than waste space on a disk or tape the file could be designed in such a way that the record would "grow" each week, i.e.,

The record description would contain the following -

```
.
.
02  ST-WEEK-COUNT          PIC S9(4)    COMP.

.
.

02  ST-WEEKLY-SALES       PIC S9(5)V99  COMP-3
                              OCCURS 52 TIMES
                              DEPENDING ON ST-WEEK-COUNT.

.
.
```

When the record is output to the new file only the number of occurrences specified in the count of weeks (ST-WEEK-COUNT) will be written to disk/tape. It is the programmer's responsibility to ensure that the number of weeks is increased by one during the program so that the length of the record "grows".

USING SUBSCRIPTS

What actually happens when a subscript is used in COBOL program?

- 1) The value in the subscript field is multiplied by the length of the recurring group in the table.
- 2) The length of one group is subtracted from the total to give the displacement along the table of the start of the recurring group to be accessed.

For example:

The recurring group in the Store File sales table described on page 2 contains the following:

Store Number	3 bytes
Store Suffix	1 byte
12 sales figures 12 x 4 bytes	
Total length	52 bytes

To refer to the 6th store in the table the number 6 is in the subscript field.

$$\begin{aligned} 6 \times 52 &= 312 \\ 312 - 52 &= 260 \end{aligned}$$

260 bytes along the table is the start of the 6th occurrence.

Note that the subscript is used while it contains 0 the calculation will result in a value of -52 which will cause the program (in this case) to access the data held in a position in the Working-Storage 52 bytes before the start of the table.

Similarly, if the value in the subscript is greater than the number of occurrences in the table then the program will obtain its information from an area of computer memory which may be Working-Storage, machine code of the program or the program Save Area where the register contents are stored. This may result in an 0C4 abend.

The compiler option SSRANGE

If you specify the SSRANGE compiler option, the VS COBOL II compiler generates additional code that makes certain a table subscript or index does not address an area outside the boundaries of the associated table. This compiler option also generates code to ensure that OCCURS DEPENDING ON values, which are set dynamically by VS COBOL II source code statements, do not go beyond the maximum boundaries initially defined for the associated variable-length table.

To invoke this option you need to code the parameter

```
COPT=SSRANGE
```

on the compile step of your job.

If you have compiled your program with the SSRANGE compiler option, subscript range checking will occur during program execution for all tables as they are referenced. If an "out of bounds" condition occurs, a diagnostic message is generated and the program is abnormally terminated. Subscript range checking occurs when a program is being executed because most table elements are referenced using computed subscripts or indexes rather than numeric literals.

The default VS COBOL II compiler option is NOSSRANGE.

If your program is compiled with the SSRANGE option you have the option to cancel the checking performed at execution time. This is accomplished by specifying the NOSSRANGE execution time option as follows:

```
//RUN EXEC PGM=MYPROG,PARM='/NOSSRANGE'
```

You can tell NOSSRANGE is a run-time option rather than a compiler option because of the slash.

It is recommended that you use the SSRANGE compiler option when you are testing programs. However because script range checking involves the execution of additional code, your program executes a little slower than it might. Therefore when programs are ready to be handed over for production running they should have the default compiler option of NOSSRANGE.

INDEXING

Indexing can be used instead of subscripting to deal with table handling. It is faster than subscripting if used correctly as it saves time by not having to recalculate the displacement of a group occurrence each time it is used.

To assign an index to a table use the

OCCURS.....INDEXED BY.....clause.

For example:

```
02      INV-STOCK-NUMBER          PIC S9(7)  COMP-3 OCCURS 100 TIMES
                                           INDEXED BY INV-INDEX.
```

INV-INDEX is the name given to the index field. The field does not have to be defined in Working-Storage as it will be set up by the COBOL Compiler in the Memory Map TGT. The indexes are stored, in the order that they are defined in the program, under the name INDEX CELLS. Each index field is a 4 byte binary field.

Within the COBOL program only the following instructions can alter the contents of an index field - SET, PERFORM, SEARCH.

The IF statement can be used to interrogate the contents of the index only.

Examples:

To alter the occurrence number use the SET instruction -

```
SET INV-INDEX UP BY 1          (Move along the table 1 occurrence)
SET INV-INDEX DOWN BY 1       (Move back along the table entry)
SET INV-INDEX TO 5            (Access the 5th entry in the table)
```

To process a table in a loop -

```
PERFORM TABLE-PROCESS VARYING INV-INDEX FROM 1 BY 1
                                           UNTIL INV-INDEX > 100
```

Subscripts can be used to reference a number of different tables as they only hold the occurrence number and the program calculates the displacement along the table each time a subscript is used.

Indices, however, may only be used for the table with which they are identified in the Data Division or for a table which has recurring groups of exactly the same length.

The reason for this is the way that indices are used by the program. The value held in the index is not an occurrence number but the displacement along the table of the start of the required occurrence.

For example:

SET INV-INDEX TO 5.

Using the ITEM-RECORD table on page 1, this instruction will result in the following calculation:

$$5 \times 4 = 20, 20 - 4 = 16$$

the value 16 is stored in the index.

SET INV-INDEX UP BY 1.

Causes the value of one occurrence's length to be added to the contents of the index field, thus:

$$16 + 4 = 20, \text{ the value } 20 \text{ is stored in the index.}$$

NB: A value of 0 in the index field points to the first entry in the table as this has a zero displacement from the beginning of the table.

INDEXING v SUBSCRIPTING

INDEXING

1. INDICES contain a displacement equivalent to (the number of the occurrence -1) x length of data items in the occurrence.
2. Displacement is calculated only when the index value is changed - this can save processing time if the same index value is needed in a number of instructions, eg., a number of MOVES to copy data from a table entry to a print line or new record.
3. Created by the INDEXED BY clause at compile time.

SUBSCRIPTING

- SUBSCRIPTS contain an occurrence number.
- Displacement is calculated every time the subscript is used.
- Specified in Working-Storage and treated as an ordinary data time.

Note:

You cannot use a mixture of indexing and subscripting on a data item at the same time. If this is done the compiler will discard the subscript and issue an error message after the instruction.

LOOKING UP DATA IN TABLES

The real power of tables (arrays) is that they can be searched very quickly for values, and COBOL provides a special command to do it. If a table has been defined with an index variable, the SEARCH command can be used to search the table for a specific piece of information.

The SEARCH command starts from the current value of the index variable, so it is important to remember to set the index to 1 before the search begins.

The search syntax includes an AT END clause that is optional. The COBOL program knows the size of the table, and it knows whether the index has reached the limit of the table. If this occurs, the command following AT END is executed. Otherwise, the SEARCH command starts from the current index variable value and performs the test described by the WHEN condition. If the condition is true, the command associated with the WHEN is executed and the search is ended.

The following is the syntax:-

```
SEARCH table name
      [AT END
        do something ]
      WHEN condition
        do something
      END-SEARCH.
```

Here is an example:-

```
SET INDEX TO 1
SEARCH TABLE-RECORD
      AT END
        PERFORM SEARCH-FAILED
      WHEN ITEM = ITEM-CODE(INDEX)
        PERFORM SEARCH-SUCCEEDED
      END-SEARCH.
```

SOLVING DUMPS INVOLVING TABLES

In order to find the current entry table in a dump the following procedures should be used:

Subscripted Fields

The subscript itself is defined in the program and represents an occurrence number. Thus to get the current address of the field, find the base address and displacement of the field (from the compiler listing) and also look in the dump for the value of the subscript.

Then calculate:

$$\begin{array}{r r r} \text{BASE ADDRESS} & + & \\ \text{FIELD DISPLACEMENT VALUE} & + & \\ \text{(SUBSCRIPT - 1)} & + & \\ \text{OCCURRENCE LENGTH} & + & \\ \hline & & \text{FIELD ADDRESS} = \\ \hline \end{array}$$

Indexed Fields

The index itself is not to be defined by a PIC clause so therefore does not appear in the compiler listing with a base address and displacement.

Also, unlike a subscript, it does not contain an occurrence number but is set to the displacement of the current entry within the occurrence.

Therefore the first step is to find the value in the index itself.

Finding the Index value

The values of the various indexes in a module are stored in the INDEX CELLS and situated in the SYSABOUT file of the output when the program abends. The nth index cell corresponds to the nth index defined in the module.

For example:

Contents of indexes are :

1 - 00000060 2 - 0000000A 3 - 0000004C

The value of an index is a displacement within the OCCURS, so to find the current location of a field, that is indexed by the nth index in the module, simply calculate

$$\begin{array}{ccccccc} \text{FIELD} = & \text{BASE} & + & \text{DISPLACEMENT} & + & \text{VALUE IN NTH} & \\ \text{ADDRESS} & \text{ADDRESS} & & & & \text{INDEX CELL} & \end{array}$$

The occurrence number corresponding to the index value can also be determined by the formula

$$\text{OCCURRENCE} = 1 + (\text{VALUE IN INDEX CELL} / \text{OCCURRENCE LENGTH})$$

NUMBER

Exercise 1

```

11.43.27 JOB 8399 IEA995I SYMPTOM DUMP OUPUT
          ABEND CODE SYSTEM=0C7 TIME=11.43.26 SEQ=006607 CPU=0000 ASID=0045
          PSW AT TIME OF ERROR 078D0000 00006E90 ILC 6 INTC07
          ACTIVE LOAD MODULE=TRP2 ADDRESS=00006270 OFFSET=00000C20
          DATA AT PSW 00006E8A - FA22619D 619A940F 619D5810
          GPR 0-3 00006F18 00006A0C 00006610 008ED654
          GPR 4-7 000065D8 50007010 00006310 000066F0
          GPR 8-11 0000AE70 00006FB2 00006270 00006AAC
          GPR 12-15 00006A20 00006778 40006E8A 00007A4A
          END OF SYMPTOM DUMP
  
```

Extract of the Data Division Map

01	UPDATE-REPORT-1	BLW=0000	0B0	DS 0CL132	GROUP	
02	FILLER.	BLW=0000	0B0	DS 4C	DISP	
02	UR1-AREA-CODE	BLW=0000	0B4	DS 2C	DISP	
02	FILLER.	BLW=0000	0B6	DS 2C	DISP	
	UR-INDEX				INDEX-NM	
02	UR1-SALES-PAIR.	BLW=0000	0B8	DS 0CL12	GROUP	O
03	UR1-F	BLW=0000	0B8	DS 1C	DISP	
03	UR1-ITEM-CODE	BLW=0000	0B9	DS 6C	DISP	
03	UR1-SALES-QUANTITY. . .	BLW=0000	0BF	DS 5C	NM-EDIT	
03	UR1-S-Q	BLW=0000	0BF	DS 5C	DISP	R
02	FILLER.	BLW=000	130	DS 4C	DISP	

Extract from the SYSABOUT file

Contents of base locators for working storage are
0 - 00006310

Contents of indexes are
1 - 00000024 2 - 00000000 3 - 0000008C 4 - 0000005A
5 - 00000060 6 - 00000000

If UR-INDEX is the 5th index defined in the module, find the address in the dump of

- (a) UR1-ITEM-CODE
- (b) UR1-AREA-CODE
- (c) UR1-SALES-QUANTITY

Also, find the occurrence number on which the program has abended.