Guidelines Enclosed

COBOL Coding Guidelines

Table of Contents

Topic **Page No.**

Naming Conventions

.

Paragraph Naming Conventions

- All paragraph names must start with a number, followed by a descriptive name, example, 4000-READ-MASTER-FILE. These numbers should be incremented appropriately to allow provision for a new paragraph between two consecutive paragraphs

  - the descriptive name should indicate the function of the process
  - the number and descriptive name should be separated by a dash (**-**)
  - the numbers should reflect the top-down, left-to-right relationships of the program's structure i.e., numbers are in an ascending order in a source listing, no paragraph PERFORMs a preceding one
  - the numbers should reflect functional relationships between paragraphs. For example : Paragraph 2000 performs paragraphs 2100 and 2200

Special Prefixes

'WS' - data name occurring in WORKING STORAGE SECTION;
'R' - data name occurring in REPORT SECTION;
'L' - data name occurring in LINKAGE SECTION;
'C' - data name is a condition data name.

Condition Names

- Defined in level 88
- it has two parts
- data name part indicates the data name under which the condition name is defined, for example :

```
01  W-MARITAL-STATUS          PIC 9(02).
    88  C - MS - BACHELOR     VALUE  1.
    88  C- MS - MARRIED       VALUE  2.
    88  C - MS - SEPARATED    VALUE 3.
```

- switches have the same name as the data item they are linked to, except for 'SW' as the prefix.

Procedure Division (Sub-heads)

Paragraph names

NNNN – XXXXX

```
|        |_____     Descriptive name (meaningful name, variable length)
|_____________      Paragraph No.
```

Paragraph names
(exit)

```
NNNN-EXIT
    |
    |_______________    Paragraph No.
```

Identification Division

The IDENTIFICATION DIVISION should contain certain basic information about the program as shown by the following example :

```
PROGRAM-ID.          PROGRAM NAME .
AUTHOR.              SATYAM.
DATE-WRITTEN.        MON  YEAR.
DATE-COMPILED.
************************************************
* MODULE          :
* TITLE           :
* TRANSACTION     :
* MAP/MFS         :

* FUNCTIONALITY:

* LOGIC           :

* PROGRAM FUNCTION KEYS USED  :

* ENTRY FROM                  :

* FILES USED   :
    LOGICAL    PHYSICAL   OPENING    PURPOSE
    NAME       NAME       MODE       SERVED

* FUNCTIONS USED:

* EXIT TO       :

* MESSAGES      :

* DEVELOPMENT / MODIFICATIONS  HISTORY:
    REVISION   MAIN       DATE OF COMMENTS
    NO         AUTHOR     COMPLETION

* REMARKS      :
************************************************
```

The practices governing this division are :

- PROGRAM-ID field should contain the Identification number assigned to the program.

- AUTHOR field should contain the name of the programmer responsible for the current version of the program

- DATE-WRITTEN field should contain the month and year the program was last coded

- DATE-COMPILED field should be left blank, for the compiler to fill up

- FUNCTIONALITY should contain brief description of the program's functioning

- LOGIC should give information of the program logic

- PROGRAM FUNCTION KEYS USED should give a brief on usage of different function keys in the program

- ENTRY FROM gives the name of calling program name(s) or module name(s)

- FILES USED should contain the following information :
  - name of the file used in the program (under Logical name)
  - name of the file on the physical storage (under physical name). Wherever possible ensure that logical and physical names resemble.
  - mode in which the file has been declared in the 'OPEN' statement in the program (under Opening mode); this can be INPUT, OUTPUT, I-O or EXTEND
  - purpose served

- FUNCTIONS USED should contain details of the functions used by the program. These include user-defined functions, library functions, system functions etc.

- EXIT TO gives the possible exits of the program

- MESSAGES give the description of messages or prompts used in the program

- DEVELOPMENT / MODIFICATIONS HISTORY should contain the following information :
  - Date on which the changes are made
  - If the customer has provided a work request number, that should be entered ie. The ISR No. of that work request
  - A brief description of the changes made in that version (COMMENTS).
  - The Tag with which the changes can be uniquely identified

- REMARKS field is meant for any special comment or remarks that the programmer wishes to make about the program, its layout, functioning, or any other aspect of the program. If relevant, it can identify critical or crucial variables used in the program and their brief characteristics. It is left to the

discretion of the programmer to decide which variables are important, and which ones are not. Some suggestions are :

- variables which set as boundary values for the program (for example, loop-counters used in PERFORM statements)
- variable altering the flow of the program
- control flags and variables
- file status variables
- names of data records and important fields
- input variables (especially values accepted on-line).

Environment Division

The ENVIRONMENT DIVISION should contain information about the environment in which the program was developed and the resources used by it. The format of this DIVISION is :

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.   IBM-4381.
OBJECT-COMPUTER.   IBM-4381.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT STOCK-STATUS-FILE
       ASSIGN TO ABCDEFG
       FILE STATUS  IS W-FST-STOCK-STATUS.

SELECT VSAM-SUPPLIER-FILE
      ASSIGN TO  HIJKLMN
      ACCESS MODE   IS  DYNAMIC
      ORGANIZATION  IS  INDEXED
      RECORD KEY   IS   SUPP-NUM   OF   VSAM-SUPP-RECORD
      FILE STATUS  IS VSAM-SUPP-STATUS.
```

The rules governing this DIVISION are :

- A ‘SELECT’ statement should be coded for each file used in the program. This will associate the file name in the program with the DDNAME given in the JCL

- If VSAM files are used, ‘ACCESS’, ‘ORGANIZATION’, and ‘RECORD KEY’, clauses should be used in the ‘SELECT’ statement

- FILE STATUS is required for all files except SORT files and must be checked after every file operation (OPEN, CLOSE, READ, WRITE)

- Entries for FILE STATUS keys in WORKING-STORAGE SECTION, should be done using the standard FILSTKEY structure, and COPY verb. For example:

```
COPY  FILSTKEY  REPLACING  FILENAME  BY STOCK-STATUS.
COPY  FILSTKEY REPLACING  FILENAME  BY VSAM-SUPP-STATUS.
```

Data Division

The DATA DIVISION describes the data to be processed. It consists of three major sections -- **FILE SECTION**, **WORKING-STORAGE SECTION** and **LINKAGE SECTION**. These are explained in the following sub-sections :

File Section

The FILE SECTION of the DATA DIVISION describes the structure and layout of files used by the program. A typical format of the FILE SECTION is :

```
DATA DIVISION.
FILE  SECTION.

FD   M01-STOCK-STATUS-FILE.
     RECORD CONTAINS   80   CHARACTERS
     BLOCK    CONTAINS    0    RECORDS
     LABEL   RECORDS  ARE  STANDARD
     DATA   RECORD   IS   M01-STOCK-REC.
01 M01-STOCK-REC.
     03 M01-X-PART-NUM              PIC X (09).
     03 FILLER                      PIC X (03).
     03 M01-I-REORDER-LEVEL         PIC 9 (05).
     03 FILLER                      PIC X (03).
     03 M01-F-STANDARD-COST         PIC 9(09)V99.
     03 FILLER                      PIC X (49).

SD S02-SORT-FILE.
     RECORD  CONTAINS  24  CHARACTERS
     DATA-RECORD   IS   S02-SORT-REC.
```

```
01 S02-SORT-REC
   03 S02-X-SORT-KEY-1        PIC X (09).
   03 FILLER                  PIC X (03).
   03 S02-X-SORT-KEY-2        PIC X (06).
   03 FILLER                  PIC X (03).
   03 S02-X-SORT-KEY-3        PIC X (03).
```

The rules governing this SECTION are :

- For a Sortfile (using Sort-file descriptions, SD) the 'LABEL RECORDS' clause, 'BLOCK CONTAINS' clause, and 'RECORDING MODE' clause are not supported
- For data sets that have labels (for example, disk files) the 'LABEL RECORDS ARE STANDARD' clause should be used. It should be remembered that most tape files also have labels
- For data sets not having labels, the 'LABEL RECORDS ARE OMITTED' clause should be used
- Since most of the File and Sort-file Descriptions are likely to be stored in a library, they should be included, using the 'COPY' statement
- PICTURE clauses should be aligned, preferably in column 50, to enhance readability. Subordinate data items should be indented to the right of the higher group data items that they are part of
- When coding the 'PIC' clause, enclose in parentheses the number of consecutive occurrences of each format character being used. For example: PIC S9(05), PIC X(01), PIC Z(04)9(02).

Working-Storage Section

The WORKING-STORAGE SECTION of the DATA DIVISION describes fields used by the program to store and manipulate intermediate values. There is no typical format for the WORKING-STORAGE SECTION. The rules governing this SECTION are :

- Level numbers subordinate to a 01 data item should be incremented by 5 to allow provision for future declarations. For example :

```
01 W-PURCH-ORDER-INFO.
   05 W-X-PURCH-ORDER-NUM        PIC X(07).
   05 W-G-PURCH-ORDER-DATE .
      10 W-X-PURCH-CENTURY       PIC X (02).
      10 W-X-PURCH-YEAR          PIC X (02).
      10 W-X-PURCH-MONTH         PIC X (02).
      10  W-X-PURCH-DAY          PIC X (02).
```

- Data names should describe their contents, but should not exceed 30 characters

- Whenever a new level begins it should be there should be an indentation of 2 spaces between the previous level and the newe level number .

- The VALUE , COMP-3 , COMP should be aligned on column 60. If necessary these can be coded on fresh line.

- PICTURE clauses should be aligned, preferably in column 45 , to enhance readability. Subordinate data items should be indented to the right of the higher group data items that they are part of. In addition, the USAGE clause of fields ('COMP-3', 'COMP', 'DISPLAY', etc.) should also be aligned

- PICTURE clause to have minimum 2 digits in bracket. For example :
  PIC 9(01) or PIC X(02).
  instead of PIC 9 or PIC XX. This is to ensure uniformity in coding

- The number of consecutive occurrences of each format character in the 'PIC' clause should be enclosed in parentheses. For example : PIC S9(07)

- Numeric fields being used for arithmetic operations should be defined as signed 'COMP SYNC' or 'COMP-3' fields. For example :

```
01  W-F-UNIT-TOTAL      VALUE  +0.      PIC S9(05)  COMP-3
```

- 77-Levels should not be used because a future compiler will not recognize them

- When a 01-level data item has been copied from a library (using the 'COPY' statement), fields that are part of that 01-level cannot be referred without referring to the 01-level. If this is not acceptable, there are three solutions -- using a 'RENAMES' clause, using a 'REDEFINES' clause, or using the COPY statement with the 'REPLACING...BY...' clause. For example :

```
COPY D012001A     REPLACING OLD-DATA-FORMAT
          BY DATE-RECORD.

01      W-DATE-RECORD.
            03  W-G-DATE-AND-TIME-STRUCTURE.
                05  W-X-CENTURY                PIC X (02).
                05  W-X-YEAR-ANSI-DATE         PIC X (02).
                05  W-X-MONTH                  PIC X (02).
                05  W-X-DAYS                   PIC X (03).
                05  W-X-ALPHA-MONTH            PIC X (02).
                05  W-X-YEAR                   PIC X (02).

        66      W-X-ANSI-DATE               RENAMES
                W-X-CENTURY THRU W-X-DAYS.
        66      W-X-YEAR-MONTH-DAYS         RENAMES
                W-X-YEAR-ANSI-DATE THRU W-X-DAYS.
        66      W-X-EXTERNAL-DATE           RENAMES
                W-X-DAYS THRU W-X-YEAR.

01      W-DATE-FIELDS-REDEF     REDEFINES
                W-DATE-RECORD.
                03  W-I-CENTURY-YEAR-MONTH-DAYS    PIC 9 (08).
```

```
03  FILLER                                  PIC X (05).
```

For levels other than the 01 that have been copied, the programmer is at liberty to choose own field names. An effort should be made to use names that are meaningful to others as well as to oneself. Abbreviations should be used only if they would be understandable to another programmer.

- To the extent possible, using Level-66 (RENAMES clause) should be avoided as it affects program readability

In addition to the above guidelines, there are a few guidelines which may be followed in order to enhance program efficiency, readability and maintainability.

- 'COMP-3' fields should be defined with an odd number of digits in the 'PIC' clause; in other words, the total number of digits before and after the decimal point (excluding the V) should be odd. For example : PIC S9(07)V9(04)

- All unusual or complex data items must be explained by means of comments; this will facilitate maintenance (more so, if the person maintaining it, is not the author of the program)

- When a program calls another program, all WORKING-STORAGE items referenced by the called program should be grouped under a single group level if possible, to avoid the confusion of passing many data names

Linkage Section

The LINKAGE SECTION of the DATA DIVISION describes data fields available to the program from an external source. There are three such sources -- another COBOL program which is calling this program, a JCL statement which is passing PARM (parameters) data, or PCB (Program Communication Block) from an IMS main program. There is no typical format for the LINKAGE SECTION. The Company Guidelines governing this SECTION depends on the mode of usage as outlined above, and are discussed in more detail in the following sub-sections :

Passing Data From One Program to Another

The called program (also known as a 'Sub-program' or a 'Sub-routine') has the LINKAGE SECTION. The rules governing this section are :

- LINKAGE SECTION data items in the sub-program must be 01 levels. Data items in the calling program's 'CALL' statement can declared at any level

- The sub-program should have a 'PROCEDURE DIVISION USING' clause, whose data items are either the same as the data items in the LINKAGE SECTION, or are identical in terms of PIC clause and sequence. For example :

```
LINKAGE SECTION.
   01     L-CALL-DATA.
          03  L-I-VAR-VALUE              PIC   S9(02).
          03  L-X-VAR-NAME                   PIC   X (07).

PROCEDURE   DIVISION   USING   L-CALL-DATA.
```

- The data items in the sub-program's LINKAGE SECTION should match the data items in the calling program's 'CALL' statement, in terms of number and order. This is because COBOL only passes the addresses of the data items in the calling program's 'CALL' statement; hence, when data in the LINKAGE SECTION of a sub-program is being accessed or updated, actually it is the data in the calling program which is edited

- If data in the LINKAGE SECTION data is being accessed repeatedly in a sub-program, it should be moved to corresponding WORKING-STORAGE variables. Accesses or updates should be made only to the WORKING-STORAGE variables. At the end of the sub-program, these variables must be moved back to appropriate LINKAGE SECTION variables.

PARM Data From JCL

All the rules outlined in the previous case apply. The additional guidelines to be followed are :

- The first two bytes of the LINKAGE SECTION should be reserved for a special variable. This will contain the length of the PARM data passed to this program

- The picture clause for this variable can be 'S9 (02) COMP SYNC' or 'X(02)'. The first format should be used if the program needs to access and check the length of the PARM data; otherwise, the second one is good enough. For example :

```
// STEP 01     EXEC PGM=DXL03415, PARM =  '02MAY91'
LINKAGE SECTION.

   01     L-PARM-DATA.
          03  L-I-PARM-LENGTH        PIC   S9(02)  COMP  SYNC.
          03  L-X-PARM-DATE          PIC  X (07).
PROCEDURE   DIVISION   USING   L-PARM-DATA.
```

Procedure Division

The PROCEDURE DIVISION contains the operative part of the program; it consists of COBOL instructions. There is no particular format for the PROCEDURE DIVISION. The next two major sub-sections look at desirable characteristics that the code must possess, and the rules governing major COBOL verbs and constructs.

Desirable Characteristics

There are several requirements that one can desire from the code; the minimum expected of the code is that it must deliver the functionality stated in its specifications, consistently, efficiently and reliably, and be easily maintainable (especially by somebody other than the author). Some of these in turn translate into characteristics the code must possess; for example, in order to be maintainable, the code must be readable, and easily modifiable.

The following sub-sections focus on characteristics like Indentation, Efficiency, Understandability, Maintainability, Comments, etc.

Indentation and Alignment

- Paragraph names should begin in column 8. Executable statements should begin in column 12. If a statement calls for continuation on additional lines, the lines following the first one should be appropriately indented. For example:

```
MOVE W-PART-NUMBER OF INPUT-SCREEN
    TO W-PART-NUMBER OF OUTPUT-SCREEN.
```

- Only one verb should be coded per line

- Only one subject / object pair should be coded per line. For example :

```
IF   (W-STATE-CODE = 'IA')  OR
           (C-MARRIED-STATUS)     OR
          (W-NUMBER-OF-DEPENDENTS NUMERIC)
THEN
            PERFORM 4000-CHANGE-STATUS
                THRU 4000-EXIT
END-IF.
```

- All the IF statements should be paired with END-IF.

- Variable names or verbs should not be broken when continuing a statement

- Clauses which qualify a verb (for example, 'UNTIL', 'THRU', 'TO' etc.) should be indented to the right, on a separate line, under the verb with which they are connected. The exception to this rule is 'VARYING'. For example :

```
PERFORM  0300-CHANGE-CODE-TO-OK-STATUS
            THRU  0300-EXIT
VARYING   W-SUB  FROM  W-I-ONE  BY  W-I-ONE
            UNTIL  (C-SEGMENT-NOT-FOUND  OF  DEALER-PCB)
                  OR  (W-SUB > 25).

COMPUTE    W-I-TOTAL-PERCENT-IMPROVEMENT =
                W-I-TOTAL-PERCENT-IMPROVEMENT +
               W-I-INDIVIDUAL-PERCENT-IMPROVEMENT.
```

Where, W-I-ONE is declared in Working Storage as,

```
01 W-I-ONE              PIC X(01)
                        VALUE +1.
```

- Arithmetic Verbs (ADD, SUBTRACT, DIVIDE, MULTIPLY) and verbs like MOVE handle variables as well as constants. If only constants and simple variables are involved then it is advisable to have the entire statement on one line; on the other hand, if variables having qualifiers (for example, 'OF', 'IN') are involved, the statement should be broken into two to more lines (i.e., in a manner similar to UNTIL, THRU etc.). For example :

```
MOVE   W-STATUS-CODE    OF   RDXCGI10-IO
       TO  W-STATUS-CODE   OF   W-OUTPUT-REC.

ADD    W-I-NUMBER-OF-LINES
       TO   W-I-TOTAL-NUMBER-OF-LINES.

MOVE   HIGH-VALUES  TO   W-SAVE-UNIT.
```

- IF , ELSE should be coded such that :

  - IF should be in alignment with the corresponding ELSE
  - ELSE appear on a line by itself

- Statements constituting ELSE are indented to the right.

An example is shown below :

```
IF  ( W-FACTORY-CODE =  'AX ' )
        MOVE HIGH-VALUES  TO W-SAVE-UNIT
ELSE
        MOVE 88   TO W-SAVE-UNIT
        IF   (C-MATCHING-UNITS)
             ADD W-I-ONE TO W-I-UNIT-COUNT
        ELSE
             SUBTRACT W-I-ONE FROM W-I-UNIT-COUNT
        END-IF
END-IF
```

- In case of coding 'IF' statement, minimum two blanks should be present after 'IF'. For example :

```
IF   STATUS-CODE =
```

- In use of relational operators, at least one blank character should precede and follow the operator.

Efficiency

The following rules enhance program efficiency :

- Accessing files or databases in a way that will generate the least number of I/O's
- Ordering the code so that the most frequently occurring situations are first
- Specifying the field types and formats correctly and optimally for their usage.

Understandability

The following practices improve the understandability of the code :

- Usage of descriptive comments to explain the purpose and function of the paragraph, and complex logic, if any. Usage of embedded comments to further clarify logic
- Wherever practical, splitting a large paragraph into smaller ones. This enhances readability and also provides leeway for inserting additional statements into the paragraphs. Generally, a paragraph should not be more than one or two pages in length
- The contents of a paragraph should be coded considering the level of the paragraph in the program hierarchy. Some useful thumb-rules are :
  - Top-level paragraphs should consist of simple 'PERFORM' and 'IF' statements; they should contain only that amount of detail code needed to implement some control functions (for example, 'MOVE')
  - Medium level paragraphs should consist of more complex 'PERFORM' and 'IF' statements (for example, 'PERFORM UNTIL', 'PERFORM VARYING', nested 'IF' etc.) and slightly more detailed code

- Low level (detailed) paragraphs should consist of action-oriented verbs like 'MOVE', 'CALL', 'COMPUTE', 'READ', 'WRITE', etc.; they should contain little or no control functions

- Indenting and arranging paragraphs uniformly

- Understandability is very poor if a section of the program has several nested 'IF' statements. This gets worse if additional 'IF' statements have to be inserted in the future (as part of maintenance). Also, re-indenting the new code will mean a lot of work. Hence, whenever several levels of 'IF' statements are nested, one should look at ways of eliminating this nesting. Typically, more than three levels of nesting is not recommended

- Simple paragraphs are easy to understand and maintain. Some practices that enhance simplicity are :

  - To the extent possible, the paragraph should perform only one type of function. This makes the program easy to understand and modify (enhancing cohesion of the program)

  - Tricks or clever code that is not readily apparent should be avoided. Similarly, using non-standard features of the language should be avoided

  - Positive logic should be used whenever possible, since most people think in the positive sense and negate any negative logic before they can comprehend it.

- Desk checking and walkthroughs of programs should be conducted to check for understandability, absence of errors or areas for improvement. This should be done without fail for all critical programs. If everyone participating in the walkthrough can understand the code, there is a good chance that future 'Maintainers' will also.

Maintainability

Understandability is a pre-condition for maintainability, i.e., a program can be maintained only if its logic can be read and understood. Hence the practices that make a program easy to understand also make it easy to maintain. Some additional aspects to consider are :

- If a program has the same code in several places, then it is prudent to create a paragraph containing this code, and execute it (through a 'PERFORM') whenever required. This makes the code easy to modify. Incidentally, it also enhances code efficiency and readability

- More often than not, certain values are hard-coded into the program. Later, modifying such values becomes a problem. Some of the situations where one resorts to hard-coding are given below :

  - when constants have to be used in the program (for example, PERFORM XYZ-PARA 20 TIMES)

  - when checking for conditions (for example, IF MARITAL-STATUS = 'S')

- when displaying error or help messages.

Some techniques to avoid hard-coding are :

- replacing all constants with WORKING-STORAGE variables that have VAL = constant

- using level-88 names for checking conditions

- creating an indexed file with each record having a different type of help message, and displaying the appropriate record whenever a help message or prompt has to be provided. This allows the help message to be modified without recompiling the original program, and also facilitates messages in multiple languages. The same strategy can be followed for error messages also.

Comments

- Comments should be included at the beginning of each paragraph explaining the function of the paragraph and any other relevant aspect that might be useful to the person maintaining the code

- Comments should be used within a paragraph to further explain any complex or unusual logic used in the code. However, comments which only repeat what the code says are unnecessary and are discouraged

The section on Comments in Document C605-PRG-000 may be referred for further details.

COBOL Verbs and Constructs

Paragraph Handling

The COBOL paragraphs contain the code for processing all the data, hence they should be coded carefully. Special attention must be paid to the way they are structured and named.

- The program should be organized into a hierarchy of paragraphs. Top-level paragraphs should focus on high-level control, directing program flow. Going down the structure of a program, the emphasis on control should reduce, with increasing focus on detail or specific actions. The lowest paragraphs should be very specific and action-oriented, having little or no program control. Typically they should contain statements such as database calls, reads, calculations, etc.

- Each paragraph name must begin with a number. For example :
  4000-READ-MASTER-FILE

- The paragraph numbers should be incremented appropriately to allow provision for a new paragraph between two consecutive paragraphs. For example, 4000-READ-MASTER-FILE would be followed by 4200-VALIDATE-MASTER-FILE.

- Paragraphs should be numbered such that :

  - the numbers reflect the top-down, left-to-right relationships of the program's structure
  - numbers are in an ascending order in a source listing
  - no paragraph performs a preceding one
  - the numbers reflect functional relationships between paragraphs. For example : Paragraph 4000 performs paragraphs 4100 and 4200

- Paragraphs should be grouped by function. For example, input procedure paragraphs should be coded before output procedure paragraphs

- The paragraph name should be descriptive of the function being performed.

PERFORM

- The 'THRU' option of the 'PERFORM' should always be used. Exiting out of a PERFORM loop should always be through the corresponding exit paragraph.

For example :

```
PERFORM 4000-READ-VOLUME-FILE
        THRU 4000-EXIT.
```

- A switch is not necessary for every condition in the 'UNTIL' clause. Errors should be grouped under one switch. Compound conditions should be used in the 'UNTIL' clause. For example :

```
PERFORM 0400-PROCESS-INPUT
    THRU 0400-EXIT
   UNTIL (END-OF-FILE OF INPUT-STATUS)    OR
    (C-INPUT-AUDIT-ERROR).
```

GO TO

- The 'GO TO' is allowed only if it branches to the EXIT of a paragraph
- Backward branching, or branching between paragraphs is not allowed.

IF

- As explained earlier, nesting several levels (anything more than 3 or 4 levels) of 'IF' statements should be avoided
- Whenever possible, negative logic should be avoided; it makes the program difficult to read, and also can result in errors
- If multiple tests are done in an 'IF', parentheses and new lines should be liberally used to help enhance readability. Parentheses are especially needed if one of the multiple tests has negative logic and especially when mixed type of tests are done -- Relational (>, <, =, Not), Conditional (88-Levels), Class (Numeric, Alphabetic), and Sign (Zero, Positive, Negative) Tests. Parentheses are always helpful and useful to separate 'IF' conditions and are suggested and encouraged. For example :

```
IF  (UNIT- CODE  = 'AX' OR '90' )
      AND
    (BUNDLE-CODE  =  '8080'  OR  '8090')
```

- If the program logic calls for testing multiple conditions, then the most frequently occurring conditions should be tested first. This improves program efficiency. Additionally, it may be possible that after one condition is satisfied, it is unnecessary to check for other conditions. There is an elegant way of accomplishing this; rather than nesting multiple 'IF' statements, a paragraph that contains the 'IF' statements should be created and exited when one condition is satisfied. In case of multiple IF statements evaluating a single variable it is appropriate to use 'EVALUATE' . For example :

```
0200-COUNT-CODES.

    EVALUATE  CODE
       WHEN  'A'
          ADD  W-X-VALUE-ONE  TO  W-X-A-COUNT
       WHEN  'B'
          ADD W-X-VALUE-ONE   TO W-X-B-COUNT

       WHEN  'C'
          ADD W-X-VALUE-ONE  TO W-X-C-COUNT

    END-EVALUATE .
```

Where, W-VALUE-ONE is declared in Working Storage as,

```
01 W-X-VALUE-ONE       PIC X(01)
                       VALUE +1.
```

- A compound 'IF' tests a group of conditions at one time. Use of complicated compound 'IF' and negative compound 'IF' should be avoided. Parentheses should be used to clarify compound 'IF' statements

- Wherever possible, 88-level names should be used in an 'IF' statement to bring about greater readability. In some cases a data item can take only one of two values - 'Y' or 'N'. Such data item is called a **Switch**. These should be implemented with 88-level names. The switch and its conditional name and its associated data item should differ only by the '- SW' suffix and the value of the switch should answer the question posed by the conditional name. For example :

```
01  END-OF-MASTER-SW                PIC X (01)
                                    VALUE 'N'.
    88  C-END-OF-MASTER             VALUE 'Y'.
```

This enhances the readability of the code by simplifying the checking of conditions. For example, with the above switch, one can code :

```
IF  C-END-OF-MASTER
```

instead of :

```
IF  END-OF-MASTER-SW =  'Y'
```

MOVE

- The 'MOVE CORRESPONDING' statement should never be used.

- Moving ZEROES or SPACES to a data item is more efficient (and modifiable) when done through variables as follows :

```
01  W-I-ZERO-VALUE                 PIC 9(01)
                                    VALUE 0.
01  W-X-SPACE-VALUE                PIC X(01)
                                    VALUE '  '.
        MOVE    W-I-ZERO-VALUE
            TO  W-I-FIELD-1.
        MOVE    W-X-SPACE-VALUE
            TO  W-X-FIELD-2.
```

- While moving literal constants, use figurative constants like ZERO, ZEROES, SPACES, ALL "X" etc.. This will avoid conversion problems in case length changes in future.

- Moving one data item to another (by means of the MOVE statement) can be made more efficient in the following manner :

  - ensuring that the source and target data items are of the same length. If the receiving item is larger, the compiler has to generate extra instructions to insert zeros (for numeric fields) or spaces (for non-numeric fields)

  - ensuring that the source and target data items are of the same usage type ('DISPLAY', 'COMP', 'COMP-3', etc.). If they do not have the same format,

the compiler must generate extra instructions to convert one, or sometimes both, data items to a common format

- replacing numeric MOVEs with alphanumeric MOVEs whenever possible. This will prevent the conversion of one or more fields (as mentioned earlier).

COMPUTE

'COMPUTE' generates more efficient code than separate arithmetic statements because the compiler can keep track of internal work areas and does not have to store the results of intermediate calculations. So wherever possible, COMPUTE should be used, keeping mind the following aspects :

- Operations are more efficient when the data items involved have the same number of decimal places. If this is not the case, efficiency is reduced since the compiler has to generate additional instructions to match the input operands, intermediate and final results

- Parentheses should be used to set off the operands and improve readability. For example :

```
COMPUTE    W-I-PERCENT-TOTAL =
                (W-I-PERCENT-TOTAL+ W-I-NEW-PERCENT) * 100.00.
```

DISPLAY

- The 'DISPLAY' verb writes data on an output device. Three forms of the 'DISPLAY' verb are available : CONSOLE, SYSPUNCH, and SYSOUT. The only 'DISPLAY' option recommended is the SYSOUT option

- A 'SYSOUT DD' statement should be included when executing a program that uses the 'DISPLAY' verb

- 'DISPLAY' is not allowed in on-line IMS programs.

File Handling

- The record should be processed in WORKING-STORAGE instead of the 'FD' by using 'READ INTO' or 'WRITE FROM' statements. For example :

```
READ  PROVIDER-CODE-FILE    INTO  W-PROVIDER-CODE-REC.

WRITE  REPORT-REC    FROM    W-REPORT-REC.
```

There are three reasons for this :

- Debugging is much simpler, WORKING-STORAGE areas are easier to find in a dump than buffer areas. When files are blocked, it is also much easier to determine which record in a block was being processed at the time of an abnormal termination (ABEND)

- Trying to access the record in the program after end-of-file (for example : In order to MOVE HIGH-VALUES to a key in the record at end-of-file) can cause problems if the record is only defined in the 'FD'

- 'WRITE FROM' allows multiple writes of the same record. This may not be always possible when using the 'FD' alone because of buffering techniques used by some COBOL compilers

- HIGH-VALUES should be put in as the key value in WORKING-STORAGE to indicate END-OF-FILE when that file is being used in matching logic. For example :

```
READ PROVIDER-FILE INTO W-PROVIDER-CODE-REC.
IF (SUCCESSFUL   OF   PROV-CODE-STATUS)
THEN
    NEXT SENTENCE
ELSE
    IF (END-OF-FILE   OF  PROV-CODE-STATUS)
    THEN
        MOVE HIGH-VALUES
          TO W-PROVIDER-KEY   OF   W-PROVIDER-CODE-REC
    ELSE
        (ABEND LOGIC).
```

- If the last record in the output file shows garbage in it when the program is abnormally terminated, it should be allocated with 'BUFNO=1' in the DCB Parameter. For example :

```
//GCXL45IN DD DSN=AA12345.DRDNN988.GCXL45IN,
  DISP=SHR,
//DCB=BUFNO=1
```

OPEN/CLOSE

- Using one 'OPEN' and one 'CLOSE' statement for multiple files is faster than individual 'OPEN's and 'CLOSE's and should be done whenever possible

- Every file that is opened should be closed. If there is no 'CLOSE' statement for the file, the last record in the output file is repeated twice (this is because the operating system writes out whatever is there in the buffer when closing the file)

- Whenever possible, the CLOSE statement should be used along with the WITH LOCK clause to prevent accidental reopening and misuse of a file whose data has already been processed.

SORT

- Whenever a program uses the 'SORT' verb, the sort routine controls the program for the duration of the sort, including INPUT and OUTPUT PROCEDUREs. If an error is encountered while the program is in an INPUT or OUTPUT PROCEDURE and the program must end abnormally (called ABEND in IBM terminology), then care must to taken to first display a user-code and a message before terminating the program

- A program using 'SORT' should OPEN and CLOSE the files, if necessary, before the SORT and not in the INPUT or OUTPUT PROCEDURE

- 'SORT-RETURN' is a special register which is used to indicate if the SORT operation was successful. 'SORT-RETURN' should always be checked

immediately after the 'SORT' verb and appropriate action taken. A sort-return of '00' means successful completion and '16' means unsuccessful completion

- No values should be moved to the parameters 'SORT-CORE-SIZE', 'SORT-FILE-SIZE', or 'SORT-MORE-SIZE'.

RETURN-CODE

- 'RETURN-CODE' is a special register which is used to pass information to the operating system about the successful or unsuccessful completion of a program. Hence, one should MOVE the appropriate value to RETURN-CODE before terminating the program; a value of Zero (0) indicates successful completion

ABEND

- When a program has to abnormally end (ABEND) for some reason, it should indicate to the user that it is terminating abnormally and the reason for the same. In the case of batch programs, the 'DISPLAY' statement can be used to give the user the above information, whereas in the case of on-line IMS programs, the program should return a screen, with a message, on an EXPRESS=YES IO-PCB before terminating, if possible.

- The text displayed should give the user or analyst complete information on why the program is terminating. It should include RECORD KEY, FILE STATUS values, and other relevant fields. For abnormal termination's relating to 'OPEN', 'CLOSE', 'READ', and 'WRITE' problems, the typical code would be as follows (a check for END-OF-FILE for 'READ' has to be added) :

```
IF  (SUCCESSFUL  OF  REQUEST-FILE-STATUS)
        THEN
            NEXT  SENTENCE
ELSE
            DISPLAY  W-MSG-OPEN-ERROR.
            DISPLAY  W-MSG-FILE-STATUS.
            DISPLAY  W-MSG-ABEND-WRNG.
            MOVE +36XX TO W-USER-CODE
            CALL 'HELPX ' USING  W-I-USER-CODE   W-X-DUMP-CODE.
```

where 'W-I-USER-CODE' and 'W-X-DUMP-CODE' are defined as :

```
01  W-I-USER-CODE          PIC S9(04)  COMP  SYNC
                            VALUE +0.
01  W-X-DUMP-CODE          PIC X(01)
                            VALUE 'D'.
```

and Display messages are defined as :

```
01 W-MSG-OPEN-ERROR      PIC X(31)
            VALUE 'OPEN ERROR-REQUEST  FILE-GDXPRITD'.
01 W-MSG-FILE-STATUS     PIC X(13)
                         VALUE 'FILE STATUS=' .
01 W-MSG-ABEND-WRNG      PIC X(27)
            VALUE 'PROGRAM WILL END  ABNORMALLY'.
```

- The 'DUMP' option of 'HELPX' should be used when terminating due to I/O errors

- Termination's arising out of database problems in batch IMS programs should display the type of call being attempted, the SSA's involved in the call, and the

PCB fields. Code to display the PCB Fields can be copied in. A typical batch IMS code to handle such termination's would be :

```
IF  (SUCCESSFUL  OF  DEALER-PCB)          OR
      (SEGMENT-NOT-FOUND  OF  DEALER-PCB)
   THEN
      NEXT SENTENCE
ELSE
      DISPLAY   'UNEXPECTED STATUS CODE FOR GU CALL '
      DISPLAY   'PROGRAM  WILL  END  ABNORMALLY '
      DISPLAY   'RDXDLX10-SSA = ' RDXDLX10-SSA
      DISPLAY   'RDXDLR14-SSA = ' RDXDLR14-SSA
      PERFORM   9900--DISPLAY-DLR-PCB
                        THRU    9900-EXIT
      MOVE  +36XX  TO  W-I-USER-CODE
      CALL   'HELPX ' USING   W-I-USER-CODE   W-X-DUMP-CODE.

   9900-DISPLAY-DLR-PCB.
      COPY DISPPCB REPLACING PCB- NAME BY DLR-PCB.
   9900-EXIT.
         EXIT.
```

STOP RUN/ GOBACK

- Only one such statement should be coded per program
- 'GOBACK' should be used in IMS programs and in all sub-programs
- 'STOP RUN' should be used in MAIN and non-IMS programs.

Tables

Indexes referred to in this section imply the 'INDEXED BY' option of the 'OCCURS' clause and not the 'INDEX' usage type of a 'PIC' clause

- Subscripts should be used instead of Indexes, even though indexing is more efficient than subscripting. This is because subscripts are normal WORKING-STORAGE fields that can easily be found in a dump, while indexes are compiler-generated fields that are not so easily found

- Multiple levels of subscripting and indexing should be avoided, since they cause extra calculations to find the occurrence

- When actually accessing a table, its subscript or index should never have a value less than one or greater than the number of occurrences of the table. If the subscript or index is not used to access the table, at a given point in time, it can have any value

- Wherever possible, literals should be used as subscripts (for example : (10)). The location of the data can then be determined at compile time. Subscripting with data items (for example : (sub)) causes the location of the data to be determined at execution time, and is inefficient

- Since indexes are actual displacements rather than occurrence numbers, an index should only be used to access the table it is defined in. It can be used to

access other tables only if those other tables are identical (same structure of 'OCCURS', same length of each elementary item at each level, etc.).

- To initialize an entire table, a 'PERFORM UNTIL' can sometimes be replaced by single 'MOVE' by setting up a duplicate area in WORKING-STORAGE with the necessary initial values

- It is more efficient to use a 'SEARCH' statement than a 'PERFORM...UNTIL' to search a table. 'SEARCH' or 'SEARCH ALL' should be used to find data in a table that has been indexed

- Subscripts cannot have arithmetic expressions in them; indexes can. For example :

```
W-TABLE-ENTRY  (W-SUB + 1)
```

- In a 'SEARCH' statement, the 'WHEN' condition can be a class condition (Numeric, Alphabetic), a condition name (88-Level), a relational condition (<, >, =, Not), or a sign condition (Zero, Positive, Negative). In the 'SEARCH ALL' statement, the 'WHEN' condition can only be a condition name (88-Level) or a relational condition of '='. In the 'WHEN' statement, the first data item must be in the table being searched. The 'WHEN' condition cannot be in parentheses in a 'SEARCH ALL'. The statement under the 'AT END' of a 'SEARCH' cannot be 'NEXT SENTENCE'. It must be an imperative statement such as 'PERFORM', 'GO TO', 'MOVE', etc. For example :

```
SET   W-SUB   TO  W-I-VALUE-ONE.
        SEARCH   W-MAJOR-TABLE
              AT  END
                 MOVE   SPACES
                     TO  W-X-MAJOR-KEY
          WHEN  W-MAJOR-TABLE-ENTRY (W-SUB) = W-X-OTHER-VALUE
          MOVE W-MAJOR-FIELD OF W-MAJOR-TABLE-ENTRY (W-SUB)
                     TO   W-X-MAJOR-KEY.
```

Where, W-I-VALUE-ONE is declared in Working Storage as,

```
01 W-I-VALUE-ONE        PIC X(01)
                        VALUE  +1.
```

- An index cannot be set to zero directly. This has to be done in the manner shown below :

```
SET   W-SUB   TO   W-I-VALUE-ONE.
SET   W-SUB   DOWN   BY  W-I-VALUE-ONE.
```

Where, W-I-VALUE-ONE is declared in Working Storage as,

```
01 W-X-VALUE-ONE        PIC X(01)
                        VALUE  +1.
```

If an index is set to zero and then a 'PIC S9' type field is 'SET' to that index, the 'PIC S9' type field will not have a value of zero.

- In an 'IF' or 'PERFORM UNTIL' with an 'OR' in it, the check for exceeding the table limit should be made before all other checks. For example :

```
(W-MAJOR-TABLE  HAS  14  ENTRIES)
        PERFORM  4000-CALC-SUM
             THRU 4000-EXIT
   VARYING W-TABLE-SUB  FROM  W-I-VALUE-ONE
                              BY   W-I-VALUE-ONE
                              UNTIL  ( W-TABLE-SUB  >  +14 )   OR
                              ( W-MAJOR-TABLE (W-TABLE-SUB)  =  0 )
.
```

Where, W-I-VALUE-ONE is declared in Working Storage as,

```
01 W-I-VALUE-ONE        PIC X(01)
                        VALUE   +1.
```

This is done because in an 'OR', once a true condition is found, the rest of the conditions are not checked.

This will prevent W-MAJOR-TABLE (15) from being checked and possible errors occurring.

- Comparisons and arithmetic operations can be made more efficient in the following manner :

  - ensuring that the source and target data items are of the same usage type
  - replacing numeric operations with alphanumeric operations whenever possible.

First Time Logic

- The use of first time switches should be avoided. For example : 'ON 1' should never be used

- Initial routines should be used to accomplish the first time logic. Initial READs and IMS GET calls are suggested.

Page Overflow

Page overflow is generally handled by performing a new page routine whenever a line counter reaches a specified value. The line counter is initialized to a large value to force an initial page break. For example :

```
0500-WRITE-DETAIL-LINE.
    IF ( W-I-LINE-COUNT  >  60 )
    THEN
       PERFORM 0600-PAGE-HEADINGS
            THRU 0600-EXIT.
    MOVE   W-I-FIELD-1
       TO   W-I-ITEM1  OF  W-DETAIL-LINE.
    (Etc.)
```

EXAMINE

The 'EXAMINE' statement should never be used; instead use 'INSPECT'.

Tally

The 'TALLY' special register will not be supported in future COBOL compilers, and should hence be replaced with an appropriate WORKING-STORAGE counter.

ENDJOB

The 'ENDJOB' PARM option of the compiler should be always used, especially when compiling IMS on-line programs. This is because some COBOL verbs and compiler options do 'GETMAIN's to get additional storage and modules and the 'ENDJOB' option will free this storage and delete the modules when it executes a 'STOP RUN' or a 'GOBACK' in a main program.

COBOL verbs that do GETMAIN's include 'ACCEPT', 'SEARCH', 'INSPECT', 'STRING', 'UNSTRING', 'MOVE' (to edit fields), 'DISPLAY', 'TRACE', 'EXHIBIT', 'SORT', 'OPEN', 'CLOSE', 'READ', AND 'WRITE'.

COBOL tasks that do GETMAIN's include 'INVALID KEY' and 'END-OF-FILE'.

Compiler options that cause GETMAIN's include 'FLOW', 'STATE', 'RES', and 'SYMDMP'.

COPY

- To make a PATH IO AREA out of two IO areas which have been copied, the following method should be used :

```
01 RDXDLX10-14-PATH-IO-AREA.
    COPY   RDXDLX10   REPLACING = = 01 RDXDLX10 = =.
              BY = 02 RDXDLX10-IO = =.
    COPY   RDXDLR14   REPLACING = = 01 RDXDLR14 = =.
              BY = 02 RDXDLR14-IO = =.
```

- Suppose source code has to be copied into the PROCEDURE DIVISION while replacing AREA 'A' entries (for example, a paragraph name). The COPY ... REPLACING statement can be used, making sure that the new values are coded in AREA 'A'. For example, if member 'STUVWXYZ' of a COPYLIB looks like this :

```
Col. 8
   |
   2000-PERFORM-SALARY-CALCULATIONS.
   ************************************************************
      CODE TO PERFORM SALARY CALCULATIONS
   ************************************************************
   2000-END-SALARY-CALC.
        EXIT.
```

The 'COPY' statement needed to copy this source code into the program and change the paragraph names would look like this :

```
COL. 8
   |
        COPY STUVWXYZ      REPLACING = = PERFORM-SALARY-
CALCULATIONS= = BY= =
   9000-CALC-SALARY= =
              REPLACING = = END-SALARY-CALC= = BY = =
   9000-CALC-SALARY-EXIT = =.
```

This results in the code shown below :

```
9000-CALC-SALARY.
************************************************************
       CODE TO PERFORM SALARY CALCULATIONS
************************************************************
9000-CALC-SALARY-EXIT.
        EXIT.
```

- A character string can be changed while copying source code (through 'COPY' statement) even if the new character string is too long to fit on a line. For example, suppose COPYLIB member 'ABCDEFGH' contains the code shown below :

```
07  W-NEW-EMPLOYEE-CODE-NUMBER    OCCURS 130.
       09  W-X-CODE1                PIC X(02).
       09  W-X-CODE2                PIC X(02).
```

The goal is to change 'OCCURS 130.' to 'OCCURS 130 INDEXED BY EMP-NUMB.', which would not fit on the same line with the rest of the text. The following commands can be used :

```
Col. 8
   |
   COPY   ABCDEFGH  REPLACING
      = = OCCURS 130. = =  BY  = =
          OCCURS  130  INDEXED  BY  EMP-NUMB. = =.
```

This results in the code shown below :

```
07 NEW-EMPLOYEE-CODE-NUMBER
       OCCURS 130 INDEXED BY W-X-EMP-NUMB.
       09 W-X-CODE1          PIC X(02).
       09 W-X-CODE2          PIC X(02).
```

Miscellaneous

- If an on-line program or sub-program uses 'DCLOAD', a 'DCFREE' call should be coded just before the 'GOBACK' statement. However, 'DCLOAD' should be avoided in favor of dynamic COBOL 'CALL' statements. For example :

```
01 DXCG2134              PIC X(08)
                         VALUE 'DXCG2134'.

    CALL DXCG2134 USING PARM-1.
```

- COBOL programs and sub-programs should be compiled with PARM, source = 'CLIST,OPT, ENDJOB'. Programming guideline, 'Standard JCL', may be referred for more information

- Sub-programs and on-line IMS programs should re-initialize WORKING STORAGE items for each new execution (Sub-programs) or input message (on-line IMS programs). This is done because WORKING-STORAGE is not automatically initialized for each new execution of a sub-program once it is loaded or for every new input message of an IMS program

- There is no predictability about the values taken by WORKING-STORAGE data items that do not have 'VALUE' clauses. If it is important that a variable should have a specific value, then care must be taken to set the value, either through VALUE clause in WORKING-STORAGE or through a MOVE statement

- Dynamic calls in COBOL code which have the form 'CALL DXCG2134 USING ...' (No quotes around the sub-program name) force the COBOL compiler to use the 'Res' compile option. When dynamic sub-program calls are present, the sub-programs are loaded at execution time. If the STEPLIB/TASKLIB DD name concatenation at execution time does not include load libraries containing the required sub-programs or if these load libraries have wrong versions of the sub-program, abnormal termination's (such as an S806) or unpredictable results may occur. If this situation arises, the load libraries containing the required sub-programs should be entered in the STEPLIB/TASKLIB DDname concatenation. If the program also contains DB2 SQL calls, 'Sys1.DB239, SSPGM' should be entered as a load library, so that it is before 'Sys1.Subrtns'. Failure to do this will result in a SOC4 when the DB2 SQL call is done, because the wrong version of 'DSNHADDR' or 'DSNHLI' is loaded from 'SYS1. SUBRTNS'

- Suppose an alphanumeric field that contains a number which is left-justified and padded with trailing blanks needs to be converted to a true numeric field (i.e. the value is right-justified and padded with leading zeroes). There are two ways of achieving this :

- the 'R JUSTIFY' sub-routine can be used, as long as the field is 256 bytes long or less; or,

- the 'UNSTRING' statement can be used as follows (assuming that W-X-AMOUNT is the source field and W-I-AMOUNT-NUMERIC is the destination or target field) :

```
01  W-X-AMOUNT                 PIC X (15).
01  W-I-AMOUNT-NUMERIC         PIC 9 (15).

        UNSTRING W-X-AMOUNT DELIMITED BY ' '
            INTO W-I-AMOUNT-NUMERIC.
```

Assuming W-X-AMOUNT contains '6300', the above statement will result in W-I-AMOUNT-NUMERIC containing '000000000006300'.

- If an alphanumeric field contains a number which has to be extracted by removing all leading spaces and left justifying the number, the following method may be used :

```
01  W-X-PIECE-ID-RIGHT-JUST     PIC X(06).
01  W-X-PIECE-ID-LEFT-JUST      PIC X (06).
01  W-I-CNT                     PIC 9 (02).

        MOVE  0  TO  W-I-CNT.
        INSPECT  W-X-PIECE-ID-RIGHT-JUST
          TALLYING  W-I-CNT  FOR  LEADING SPACES.

    IF (W-I-CNT = 0  OR  6)
*       NO LEADING SPACES OR ALL SPACES
        MOVE  W-X-PIECE-ID-RIGHT-JUST
                TO W-X-PIECE-ID-LEFT-JUST
    ELSE
*       MOVE POINTER TO FIRST NON-SPACE CHARACTER
        ADD W-I-VALUE-ONE TO W-I-CNT
        UNSTRING  W-X-PIECE-ID-RIGHT-JUST
        INTO  W-X-PIECE-ID-LEFT-JUST  WITH  POINTER  W-I-CNT.
```

Where, W-I-VALUE-ONE is declared in Working Storage as,

```
01 W-I-VALUE-ONE        PIC X(01)
                        VALUE +1.
```

Annexure - I

VS COBOL II Considerations

The following aspects are to be kept in mind when dealing with VS COBOL II programs :

- The 'BLOCK CONTAINS' clause should be always shown with '0 RECORDS'. This will allow the JCL to supply the block size

- The compiler does not permit a data item to have the same name as the program name; error message 'IGYDS1266-E' is flashed when the compiler detects such a data item

- VS COBOL II can issue run-time user ABEND (abnormal end) codes in the range U1000-U1999. The last three digits of the user ABEND code correspond to the 'NNN' in the associated run-time message IGZNNNI. The section on 'Run-Time Message' in the VS COBOL II debugging guide should be referred for more details

- When 'NEXT SENTENCE' is used in an IF statement and gets activated, control passes to the statement after the closest following period, not to the statement following the 'END-IF'. This can produce apparently puzzling results. An example illustrates this point :

```
IF X = Y
        NEXT SENTENCE
ELSE
        PERFORM 8000-PARA THRU 8000-EXIT
        END-IF
        IF   SEVERE-ERROR
            GO TO 1000-EXIT .
        MOVE   X  TO  Y.
```

If 'NEXT SENTENCE' is executed, control passes to the 'MOVE' statement and not to the 'IF SEVERE-ERROR' statement. To avoid this, one should code 'CONTINUE' instead of 'NEXT SENTENCE', or put a period after the 'END-IF', or use both 'CONTINUE' and a period

- To execute VS COBOL II programs that contain a 'SORT', one must allocate a 'SYSOUT' file for sort messages and an 'IGZSRTCD' file for sort control statements. The files can be allocated like this :

```
//  SYSOUT DD  SYSOUT = * ALLOC   FI(SYSOUT) DA
    (*) SHR REUSE
//  IGZRTICD   DD   DUMMY ALLOC   FI(IGZSRTCD) DUMMY
    REUSE
```

The 'SYSOUT' file can be 'DUMMY' but then this would result in the loss of messages in the event the SORT failed. The 'IGZSRTCD' file can be a dataset or 'DD', if DFSORT has to be supplied with some control statements. The VS COBOL II application programming guide manual should be consulted, under the

heading 'SORT', for more information. Since no control statements are needed in most cases, IGZSRTCD is usually 'DUMMY'

- An in-line 'PERFORM' can be used if the code to be performed is less than one page in length and the code is used only once in the program. If the code is used more than once, it should be placed in a paragraph and that paragraph should be executed (through a 'PERFORM' statement) wherever required

- When dealing with 88-level names, the 'SET ...TO TRUE' statement should be used instead of a 'MOVE' statement and may be easier to understand. For example :

```
01  DEALER-TYPE                  PIC  X (01).
    88  C-AG-DEALER                  VALUE   'A'  .
    88  C-INDUSTRIAL - DEALER        VALUE   ' I '  .
    88  C-CP- DEALER                 VALUE   ' L ' .

SET  C-AG-DEALER  OF  DEALER-TYPE  TO  TRUE .
```

which is definitely more obvious than :

```
MOVE   'A'   TO   DEALER-TYPE .
```

- When initializing fields in fixed-length tables the 'INITIALIZE' statement should be used. This is more efficient than using 'PERFORM' and 'MOVE'. For example :

```
01  W-UNIT-TABLE
    03  W-UNIT-ENTRY            OCCURS 100 TIMES
              INDEXED BY W-I-UNIT-INDEX
        05  W-X-UNIT-NAME             PIC  X (37).
        05  W-F-UNIT-TOTAL            PIC  S9(07)   COMP-3
```

To store SPACES in UNIT-NAME and ZEROES in UNIT-TOTAL, the code would be as follows :

```
MOVE SPACES TO W-UNIT-TABLE .
    PERFORM
    VARYING   W-I-UNIT-INDEX  FROM   W-I-VALUE-ONE
                                BY   W-VALUE-ONE
                                   UNTIL (W-I-UNIT-INDEX > 100 )
          MOVE + 0  TO   W-F-UNIT-TOTAL (W-I-UNIT-INDEX)
    END-PERFORM.
```

Where, W-I-VALUE-ONE is declared in Working Storage as,

```
01 W-I-VALUE-ONE          PIC X(01)
                          VALUE +1.
```

With the 'INITIALIZE' statement, the code would be as follows:

```
MOVE   SPACES   TO   W-UNIT-TABLE .
INITIALIZE   W-UNIT-TABLE .
```

It should be remembered that the tables which use 'OCCURS ...DEPENDING ON' cannot be initialized through a 'INITIALIZE' statement. Also, any filler fields in a group will not get initialized; that's the reason why the above example had a MOVE SPACES statement first

- VS COBOL II programs and sub-programs should be compiled with PARM, Source = 'OFFSET, SSRANGE, OPT'

- The 'PROCESS', 'CBL', 'CONTROL', or 'CBL' statements should not be used

- VS COBOL II permits an additional file-status field on the 'FILE STATUS' clause of the 'SELECT' statement for VSAM files. For example :

```
SELECT REQUEST-FILE         ASSIGN TO S-GDXPCT1R
       FILE STATUS IS REQUEST-STATUS REQUEST-VSAM-
       STATUS.

01  REQUEST-VSAM-STATUS .
       03  VSAM-RETURN-CODE              PIC 9 (02) COMP.
       03  VSAM-FUNCTION-CODE            PIC 9 (01) COMP.
       03  VSAM-FEEDBACK-CODE            PIC 9 (03) COMP.
```

These 'VSAM -' fields can then be displayed in case of problems during an 'OPEN', 'CLOSE', 'READ', or 'WRITE' operation of the VSAM file.

Annexure - II

Proposed Checklist

The following is a check list for a programmer to ensure adherence to the COBOL Coding Conventions.

- PIC clause has to be coded from column no. 50
- VALUE clause (if necessary) has to start on a fresh line -from column number 60
- All working storage data names to have prefix WS
  For example : WS- PURCH-ORDER-NUM
- Avoid all hardcoding in Procedure - Division
  Make the necessary declarations in Working-Storage
- Only one Verb is to be coded per line
- Try to minimise use of several nested 'IF' statements
- All the IF statements should be paired with END-IF statements
- All Help-Messages are to be pre-defined in the Working-Storage.
  All Error-Messages and Display-Messages may also be described in the Working-Storage or in a Copybook
- Use comments which improve Code understandability
- Each paragraph name must begin with a number
  For example : 4000-READ-MASTER-FILE

  These numbers to be in ascending order of source listing and should reflect the functional relationships between paragraphs. For example :

  2000-PARA PERFORMS 2200-PARA.

  Common paragraphs should start from numbers between 9000 to 9999
- Try to avoid use of negative logic in the 'IF' statements. This is to improve Code understandability
- Try to avoid use of 'GOTO'. It is allowed only if it branches to the EXIT of a paragraph
- Use 88 level for conditional names
- Use minimum 2 digits in PIC.
  For example PIC 9(01) or PIC X(02)
  not PIC 9 or PIC XX.