Global 16-bit Development System
Global Cobol Language Manual
Version 8.1

TABLE OF CONTENTS

APPENDICES

Appendix Description Pa

1. Foreward

This reference manual describes the Global Cobol Language used to develop programs to run under Global System Manager. There is a short bibliography of other relevant Global Software documentation following this introduction, then the Foreword concludes with a section explaining the notation used in describing Global Cobol.

The remainder of this Language Manual is divided into chapters and appendices, each of which focuses on a particular area. Their content is described briefly below.

Chapter 2 introduces the fundamentals of the language - character set, symbol construction, character and numeric strings and statement format. It explains how the compiler's COPY, PAGE, OPT, PROGRAM and ENDPROG directives are used and how a program is structured as a data division and procedure division.

Chapter 3 describes all the statements of the data division with the exception of those concerned with file and map definitions.

Chapter 4 defines the procedure division statements, apart from those used in file processing, screen handling, map processing, and sorting. As well as the conventional COBOL statements there are special extensions for structured programming, the management of chained programs and overlays, and multi-user working.

Chapter 5 describes the use of pointers, pointer arithmetic, based areas and the GLOBAL statement.

Chapter 6 covers Intermediate Code, allowing you to interface with the interpreter at a more basic level than Global Cobol itself.

Appendix A defines the standard ASCII character set used by Global Cobol.

Appendix B describes the various compiler and cross-reference options available through either a command keyed at run-time or an OPT statement coded as part of your program. For example there are compiler options to determine whether or not source lines are listed, whether statements included from copy books are to be printed, and whether a symbol table is to be produced at the end of the listing.

Appendix C summarises in one place the various restrictions that you should bear in mind when producing Global Cobol programs to run on a wide variety of configurations. It is recommended that you read this carefully before starting program design.

Appendix D is a collection of Global Cobol programming hints derived from practical experience.

Appendix E describes suggested symbol and file naming conventions for Global Cobol which have been proved in practice on a number of projects.

Appendix F describes the format of a Global Cobol copy library.

Appendix G contains a list, and full explanation, of all the error messages that can be produced by the Global Cobol compiler.

1.1 Bibliography

The Global Cobol Language Manual is one of a set of inter-related documents listed below:

Global System Manager Manual
Global Operating Manual
Global Utilities Manual
Global Cobol User Manual
Global Cobol Language Manual
Global Cobol Screen Presentation Manual
Global Cobol Screen Support Manual
Global Cobol Data Management Manual
Global Development File Management Manual
Global Development Toolkit Manual
Global Development System Subroutines Manual
Global Development Job Management Manual
Global Speedbase Development System User Manual
Global Speedbase Development System Language Manual
Global File Converters User Manual
Global Configurator User Manual
Global Assembler Interface Manual

The **Global System Manager Manual** contains descriptions of the most commonly used Global System Manager commands. It is recommended preliminary reading if you are new to Global System Manager since it provides a short, end-user oriented description of the environment established by Global System Manager. The manual also explains how to customize your Global System Manager.

The **Global Operating Manual (o/s)** explains how to initiate Global System Manager on a particular operating system. For example, the "Global Operating Manual (Unix)" manual explains how to install, run and configure Global System Manager under Unix. The various operating manuals also describe any additional utilities, and additional functionality in standard utilities, which are dependent on the host operating system.

The **Global Utilities Manual** extends the Global System Manager Manual with descriptions of the more sophisticated utilities and detailed information required by system supervisors and computer specialists. Programmers developing Global Cobol software will find the account of program checks and the lists of STOP and EXIT codes invaluable.

The **Global Cobol User Manual** describes the commands used for program preparation, debugging and maintenance. These include a text editor and linkage editor as well as the Global Cobol compiler itself. There is also a source program cross reference utility. A powerful symbolic debugging system aids program testing: with it you are able to set traps in programs and on variables and then examine, and optionally modify, any named field before resuming execution.

The **Global Cobol Language Manual** describes the Global Cobol language used to develop programs to run under Global System Manager.

The **Global Cobol Screen Presentation Manual** contains the Global Cobol commands used for screen displays and accepts together with the special Global Cobol verbs (MAPIN, MAPOUT and MAPCLEAR) that are used with the $FORM screen formatting command. $FORM allows you to create a display screen format table known as a map. Then, instead of having to DISPLAY and ACCEPT individual fields you can simply code the MAPOUT statement to output the entire map, and the MAPIN statement to obtain all the input fields. Coding effort and program size is, as a result, greatly reduced in certain applications. The map is declared in a map definition (MD) coded in the data division. You will find that, in the interest of accuracy, there are a number of references to map definition and map processing statements in the language description which follows, even though the statements involved are explained in the Screen Presentation Manual. This manual also contains some advanced screen handling techniques and describes the AutoGuide facility.

The **Global Cobol Screen Support Manual** describes how to create a Terminal Attribute Program (TAP) in order to support a new type of screen under Global System Manager. This involves preparing a special format source program containing details of the keyboard and sequences used by the display. The $TAP command is used to create the TAP from the source. This manual also describes the $T command in complete detail.

The **Global Cobol Data Management Manual** is a complete guide to the Global Data Management system, including the DMAM access method, various associated system subroutines and the utility programs that form the Data Management System.

The **Global Development Cobol File Management Manual** contains a detailed description of the file organizations and access methods available in Global System Manager, together with the associated system subroutines.

The **Global Development Toolkit Manual** describes a collection of command programs designed to aid the production of large systems written in Global Cobol. Separate sections also deal with Product Translator, Intermediate Code generated by $COBOL, the Metajob Management and the Macro Preprocessor languages.

The **Global Development System Subroutines Manual** really forms a logical extension of the Language Manual. It contains the system subroutines used for Date and Time Conversion, Arithmetic and Data Conversion, Program Management and System Management, together with special coding techniques, storage management facilities and scientific calculation facilities.

The **Global Development Job Management Manual** explains how you can create catalogued procedures, known as job files, to run Global command programs, or your own applications, in a non-interactive mode in which, after initial parameterisation, operator input is supplied from the job file rather than from the console. The system might typically be used in conjunction with the $V command program to initialize a series of diskettes or hard disk units. You can optionally suppress console dialogue from programs run under job management.

The **Global Speedbase Development System User Manual** describes the commands used for the preparation of programs developed using the Global Speedbase 4th Generation Language (4GL). These include a text editor as well as the Speedbase compiler itself.

The **Global Speedbase Development System Language Manual** describes the Global Speedbase language (4GL) used to develop programs to run under Global Speedbase Presentation Manager.

The **Global File Converters User Manual** describes the Unix and MS-DOS file converters. This manual also specifies how you can interface in a low-level way to the host operating system and Global System Manager.

The **Global Configurator User Manual** explains how you can update the configuration file distributed with your Global System Manager to change parameters such as the maximum number of files that can be open at any one time, the maximum number of users, buffer sizes, hardware devices and addresses, and so on.

The **Global Assembler Interface Manual** explains how you can create and run assembler code programs under Global System Manager. This manual also describes the interface available to the Serial Port Driver (SPD) software.

1.2 Notation Used in Describing Global Cobol

Global Cobol statements are described using a simple notation which will already be familiar to most programmers.

All numbers and UPPER CASE words, with the exception of the single letters A, B, C ..., are to be reproduced unchanged whenever a statement containing them is coded. Lower case words in italics (such as item, date, condition), together with the single letters A, B, C ..., are generic terms to be replaced by an actual value when the statement is coded.

When a portion of a statement description is enclosed in square brackets, [ ], that part is optional. You may choose whether or not you include it when coding the statement.

A vertical bar, |, is used to separate alternative items. Only one of the alternatives may appear in a given instance. For example the OPTION statement may be coded as either OPTION ERROR or OPTION RESET. We show this as:

```
     OPTION ERROR|RESET
```

Sometimes a choice must be made from a number of independent Global Cobol statements. In this case each option appears on a separate line, enclosed by vertical bars:

```
     | IF condition |
     | ON EXCEPTION |
     | ON OVERFLOW |
     | ACCEPT...NULL|
```

The ellipsis (...) is used in a statement description to indicate that a repetitive part has been omitted in the interests of clarity. The context will always make the meaning of the ellipsis clear.

Consider, as an example, the CALL statement. It is used to pass control to the entry name of a subroutine. The optional USING clause can be employed to supply up to seven parameters. A possible statement description for CALL would be:

```
     CALL entry-name [USING A B ...]
```

Some valid CALL statements that you might code include:

```
     CALL SUBRTN
     CALL DELRT USING FILEA
     CALL QUADR USING ALPHA BETA GAMMA
```

When it is necessary to refer to hexadecimal values in the text, we prefix the hexadecimal digits (0-9, A-F, inclusive) with a # character. Thus instead of having to write, for example, "the quantity 13 (hexadecimal)", we can simply refer to "the quantity #13".

2. Fundamentals

2.1 Language Elements

2.1.1 Character Set

The character set is an ASCII 8-bit code with the senior (parity) bit set to zero, as defined in Appendix A:

- The **digits** are ASCII 0 to 9
- The **letters** are upper and lower-case ASCII A to Z and $.
- **Alphanumerics** are characters which are either digits or letters, or the hyphen.
- **Blank** is represented by b in this document.

Other **special characters** will be introduced as the language description unfolds.

2.1.2 Symbols

A symbol must start with a letter and be followed by any number of alphanumeric characters. Normally the first six characters of each symbol must be unique throughout the program, the compiler ignoring the seventh and subsequent characters, apart from listing them. However, if the "long names" option is in force, the compiler treats the first 31 characters of each symbol as significant. (Compiler options are explained in 2.3.1 and Appendix B.)

Symbols are used for data names, section names, paragraph names, entry names, program names and file names. Entry names and program names are **global symbols** which are processed by the linker as well as the compiler. Because the linker only treats the first six characters of each global as significant, the first six characters of each global participating in a linkage edit must be unique irrespective of whether the "long names" option is in force or not.

The letter $ should not be used in a symbol created by the application program since it is employed in symbols used by Global software. In addition there are twelve **reserved words** in Global Cobol and these must not be used as symbols. They are:

DEPENDING	FILLER	HIGH-VALUES	LOW-VALUES
NEXT	SPACE	SPACES	USING
PRIOR	FIRST	LAST	INTO

Other language words such as IF, PIC and NOT can be used as symbols, although it is recommended that they be avoided in the interests of clarity.

2.1.3 Character Strings

A character string may be made up of any combination of the graphic ASCII characters (those with a numeric equivalent in the range decimal 32 to 126 - see Appendix A for details) with the exception of double quotes (" decimal 34), which are used as string delimiters. When coded, the string appears as:

```
     "character string"
```

For example:

```
     "HELLO WORLD"
```

is a character string contains the 11 characters:

```
     H E L L O b W O R L D.
```

The compiler assumes that programs have been created on machines with industry-standard tab settings at character positions 9, 17, 25... and so on, and if it finds a tab character within the string replaces it with the appropriate number of blanks.

2.1.4 Integers

Integers must be in the range -32768 to +32767. The plus sign is optional when an integer is coded.

2.1.5 Numeric Strings

Numeric strings are character strings consisting of:

- optional leading blanks, followed by...
- an optional + or - sign, followed by...
- 1 to 15 digits, which may be omitted if the decimal point is present. These in turn are followed by...
- an optional decimal point which, if present, must be followed by between 1 and 7 digits...
- and optional trailing blanks.

The total number of digits must not exceed 18. Examples of valid numeric strings are:

```
     -3   3.14159   +1246    -0.120        .7
```

The strings:

```
     3.   3.1b4159  + -12         .7-       +b9
```

are **not** valid numeric strings. Programmers familiar with COBOL should note that in Global Cobol a string of ASCII blanks is **not** a numeric string, and such a string cannot therefore be used as a display numeric zero.

2.1.6 Standard Numeric Strings

When a number is converted to character form, either for output or for storage in a display numeric variable, it assumes the format of a **standard** numeric string. In such a string:

- leading zeros will always be replaced by blanks (except in the units position);
- the sign will be omitted if positive;
- at least 1 digit will always precede the decimal point;
- there will be no trailing blanks;
- if the number is defined as fractional a decimal point and the number of decimal places specified will be printed, even if the value is a whole number.

Decimal point customisation can be applied using the $CUS command program to cause the decimal point appearing in standard numeric strings to be represented by a comma rather than a period (full stop). This means that decimal fractions which are displayed, printed, or keyed by the operator assume the format familiar in many European countries. Programmers however must still continue to use the period as a decimal point whenever they code a numeric string in Global Cobol, i.e. in VALUE statements (3.4) and computational literals (4.2.3).

Here are some typical conversions from numeric string to standard format assuming the fields involved are signed with 2 digits before the point and two after, and decimal point customization is **not** applied:

```
3          becomes   bb3.00
+02.13     becomes   bb2.13
.1         becomes   bb0.10
-.2        becomes   b-0.20
-21.43          becomes   -21.43
```

2.1.7 Hexadecimal Strings

A hexadecimal string is coded as a # sign followed by pairs of ASCII "digits" in the ranges 0-9, A-F inclusive. The number of digits in the string must be even, since each digit pair establishes a single byte. For example:

```
#07
```

is a single byte string, representing the ASCII bell character, and:

```
#FFFF
```

is a two-byte string, with each bit set to 1.

2.2 Code Layout

2.2.1 Comments

Comments must be preceded by an asterisk. They may appear either on a line on their own or following a statement and can appear anywhere in a program.

2.2.2 Statement Format

A Global Cobol statement, including comments, consists of a single line of up to 72 characters. Statements may not be continued onto the next line, nor may more than one statement appear on a line. The individual constituents of a statement (e.g. language words, variables, strings and comments) must be separated from each other by one or more blanks or tabs but, apart from this consideration, the spacing within a line is unimportant. However, the following conventions, if adopted, result in a tidy, readily readable listing:

- Paragraph names and the statements listed below should start in column 1:

  ```
  PROGRAM      COMMON SECTION
  DATA DIVISION          EXTERNAL SECTION
  01                LINKAGE SECTION
  77                PROCEDURE DIVISION
  FD                ENTRY
  MD           SECTION
                    ENDPROG
  ```

- Generally, other statements should begin in column 9. The exceptions are:

  - The level numbers (02 to 49) used in group data definitions. These should be suitably indented to make the data structure clear. There is an example in the section of Chapter 3 which deals with data definitions;

  - The VALUE statement, which should be coded underneath the preceding PIC statement. The PIC statement itself should begin in column 9;

  - Each TO statement of a GO TO DEPENDING ON construct, which should be coded so that it is aligned with the TO of GO TO DEPENDING ON;

  - Statements within a conditional or iterative structure. These should be indented an additional four spaces for each level of nesting, to highlight the program structure.

- Comments should start in column 41 in the procedure division, and 49 in the data division, except for 'across the page' comments which should start in column 1.

2.3 Compiler Directives

2.3.1 The OPT Statement

The OPT statement is used to select certain compilation or cross-reference options to apply to the program in which it appears. One or more OPT statements may be coded at the very beginning of the compilation, and may be preceded only by comments or the PAGE instruction.

The statement is coded:

```
     OPT compiler-option
```

where compiler-option is a sequence of non-blank characters. For example, the statement:

```
     OPT NCX
```

(no copy book expansion) stops the compiler from listing any statement included in the program from a COPY book, unless that statement is flagged in error. Those options which control the listing, such as NCX, may appear anywhere within the program, and not just at the start. They remain in force until a contradictory option is found.

Each option has a standard default associated with it, which applies when no details concerning the option are supplied. For example, unless NCX is specified, the default is to list all statements included by a COPY, unless COPY...SUPPRESS was coded.

If a number of OPT statements supply conflicting compiler options then the option in force will be the one coded last. You may also key options at the console when you run the compiler, and these run-time options override those supplied in OPT statements. The various compiler options which are available are described in detail in Appendix B.

2.3.2 The PAGE Statement

A new page in the program listing is generated by the statement:

```
     PAGE ["character string"]
```

The optional character string may be supplied to provide a title up to 30 characters in length, to be printed on the listing. This will appear in all subsequent page headings until a PAGE statement is coded which specifies a new title. If the table of contents compilation option is in force, an extra page will be produced at the start of the listing containing the character strings specified in page statements within the program, together with their line numbers.

The initial 30 characters of the first title found in a program are used to provide a title for the resulting compilation file. This title will be used to identify the module when it is linkage-edited or stored on a library.

2.3.3 The COPY Statement

The COPY statement is used to include a group of one or more Global Cobol statements from a copy library. It is coded:

```
     COPY cc [SUBSTITUTING "character string"][SUPPRESS]
```

where cc is a one or two character book name which serves to distinguish different groups of statements (books) within the library. (Appendix F defines the format of a copy library.)

The COPY statement may be coded anywhere within the Global Cobol source providing that it does not appear before the last OPT statement or after the ENDPROG statement that terminates the source. The copied statements may themselves contain COPY statements, and the books thus retrieved may contain further COPY statements, but that is as far as the nesting may go. Thus just two levels of internal nesting are supported.

The optional SUBSTITUTING clause allows a parameter string coded in the copy book to be replaced by the specified character string. A parameter within a copy book is a string of one or more & characters. Although it may appear anywhere within a statement or label, a parameter should not be coded within a comment, since the copy process does not change comments. If the SUBSTITUTING clause is omitted, but the copy book contains parameter strings, substitution will take place as though the copy book name itself had been specified. That is, the following two statements are treated identically:

```
COPY cc [SUPPRESS]

COPY cc SUBSTITUTING "cc" [SUPPRESS]
```

If, however, the COPY statement is nested and the SUBSTITUTING clause is omitted, the substitution string for the containing copy book is used. During the copy process each & character from a parameter string is replaced by the corresponding character from the substitution string. If there are more &s than substitution characters, the rightmost ones are replaced by hyphens. For example, if a copy book AB contains the line:

```
77 &&area PIC X(20)
```

then substitutions can be made as shown below:

```
COPY AB SUBSTITUTING "XY"         generates 77  XYAREA     PIC
X(20)
COPY AB SUBSTITUTING "X"     generates 77  X-AREA     PIC X(20)
COPY AB SUBSTITUTING "XYZ"        generates 77  XYAREA     PIC
X(20)
COPY AB                      generates 77  ABAREA     PIC X(20)
```

The optional SUPPRESS phrase, if present in a source program COPY statement (rather than one which is itself embedded in a copy book), prevents the compiler from listing the statements it copies apart from those that are found to be in error. If the SUPPRESS phrase is omitted from the source program COPY statement, then every statement it introduces will be listed unless an overriding compiler option is in force. A SUPPRESS phrase coded on an embedded COPY statement is ignored: suppression is controlled at the source program COPY level.

When more than one copy library is specified a book is always copied from the first copy library that contains that book.

2.4 Language Skeleton

Every Global Cobol program is constructed as follows:

```
     [OPT statements]

     PROGRAM program-name

     DATA DIVISION

     [file, map and data definitions]

     [COMMON SECTION section-name        |repeated for
                                         |
     file, map and data definition]            |each section

     [EXTERNAL SECTION section-name          |repeated for
                                         |
     file, map and data definitions]           |each section

     [LINKAGE SECTION

     file, map and data definitions]

     [PROCEDURE DIVISION]

     [ENTRY [entry-name] [USING A B ...] ]

     [SECTION section name]                  |repeated for
                                         |
     other language statements                 |each section

     ENDPROG
```

PAGE, COPY, and across-the-page comment statements can appear anywhere before the ENDPROG statement which terminates the source.

2.4.1 The PROGRAM Statement

The PROGRAM statement is coded, following any OPT statements, at the start of the compilation. The format is:

```
     PROGRAM program-name
```

The program-name is a global symbol used to identify the Global Cobol compilation. This is the name which the linkage editor uses to refer to the compilation when creating a program file. It is not used as the identifier of the resulting program file since a number of compilations may be combined to produce the file.

The program name is printed on the map listing output by the linkage editor together with the starting location and length of the storage area that the associated compilation will occupy when the program file containing it is run. The program name must be a symbol unique within its own compilation. In addition, since it is a global symbol, its first six characters must not be the same as any other global symbol participating in linkage edits involving the compilation.

2.4.2 The Data Division
The data division begins with the header DATA DIVISION and is delimited by the header PROCEDURE DIVISION. It is discussed in Chapter 3.

2.4.3 The Procedure Division
The procedure division begins with the header PROCEDURE DIVISION and is delimited by the trailer ENDPROG. The language statements it contains are discussed in Chapter 4.

2.4.4 The ENDPROG Statement
The ENDPROG statement must be the very last statement of the program. Any statements following it will be flagged in error.

3. The Data Division

3.1 Structure

The data division describes the data used by a Global Cobol program. It begins with the header:

```
     DATA DIVISION
```

which must be coded on a new line. It is terminated by the header:

```
     PROCEDURE DIVISION
```

also on a new line.

The data division may contain local data, for use by the current compilation only, then zero or more common sections, zero or more external sections, and finally an optional linkage section. When these extra sections are present they are introduced by the headers:

```
     COMMON SECTION section-name
     EXTERNAL SECTION section-name
     LINKAGE SECTION
```

The name used to identify a common or external section is a global symbol and must be unique within the compilation. In addition, since it is a global symbol, its first six characters must not be the same as any other global symbol participating in linkage edits involving the program.

The part of the data division consisting of local data and common sections is known as **working storage**. There is an important difference between working storage data definitions and those appearing in external sections or the linkage section because the former occupy space in the program itself whereas the latter simply describe data located elsewhere. Sometimes this difference is reflected in the Global Cobol language, but in most cases data division statements appearing in working storage, external sections, or the linkage section are treated identically.

Data division statements (apart from the headers) are used to set up file definitions, map definitions and data definitions. Except when OPT ED (even data) is specified, each file, map and data definition is aligned to begin on an 8-bit byte boundary, and occupy an integral number of bytes. However, if OPT ED is in force, binary zero padding bytes are inserted as necessary between file definitions, map definitions, and level 01 data definitions, so that they begin on a 16-bit, even-byte boundary, and occupy an integral number of 16-bit "words". (This option can improve performance on certain machines, as explained in Appendix D.)

3.1.1 Common and External Sections

Common and external sections provide Global Cobol with a facility equivalent to FORTRAN labelled COMMON, or PL/1 external static storage. Data defined within a common section can be referenced directly by other programs using an external section of the same name. The defining program containing the common section and the referencing programs accessing it through external sections must be linkage edited together, so the referencing programs would typically be subroutines or overlays used by the definer. The linker changes statements referencing external sections so that they access the corresponding common sections. No storage is reserved for external sections in the resulting program file.

Common and external sections enable large quantities of data to be passed between different programs in an efficient manner. Typically a common section can be used for the single occurrence data of a system that virtually every program requires, for example master file definitions. Alternatively a common section can provide a work area for use by a set of related subroutines.

Obviously the data division statements within an external section must correspond exactly to the definitions used within the matching common section since both define the same storage area. The simplest way to achieve this is to include the statements in a copy book. For example, suppose book MF contains master file definitions used throughout a Sales Ledger system. The definitions could be included in a main program by the statements:

```
     COMMON SECTION SAMF
     COPY MF
```

and referenced from a subroutine by:

```
     EXTERNAL SECTION SAMF
     COPY MF
```

3.1.2 The Linkage Section

The linkage section must be used to define data whose address is not known until run-time. The most obvious example is parameters supplied by a calling program by means of the CALL... USING statement (4.5.4). A subroutine invoked by such a statement must define the parameters it receives in the linkage section. In addition, based areas (see chapter 7), whose run-time address can be varied by means of the BASE statement or pointer manipulation, must be defined in the linkage section.

It is important to appreciate the difference between parameters passed via the common/external section mechanism and via the linkage section. The former can be used when only one copy of the information exists at a location which, although unknown when the program is compiled, is fixed by the time it is linked. Thus common/external parameter passing is employed to pass information, such as a set of master file definitions. It would not, for example, be used by a subroutine whose function was to convert a date from one form to another. Clearly such a routine might be called to process any number of dates scattered throughout different data divisions of different callers. The correct strategy is of course to pass the date to be converted as a parameter of the CALL...USING statement.

3.2 Data Definitions

Global Cobol provides the usual COBOL data definition facilities, extended with enhancements for systems programming. The language supports level 77 elementary items, as well as level 01 group items which can themselves be subdivided into elementary items, or as many as 19 levels of subgroup.

3.2.1 Defining Level 77 Elementary Items

Level 77 elementary items, which are not subdivided, may be defined in the data division by coding:

```
     77 data-name        [REDEFINES name-1] [OCCURS n] picture clause
                    [BASED name-2]
```

The data-name must be a symbol (you **cannot** use the reserved word FILLER in its place). If an OCCURS clause is present the quantity n must be an unsigned positive integer.

The optional REDEFINES clause allows you to redefine a previously declared item whose data-name you specify as name-1. An item with a REDEFINES clause is known as a **redefinition**. The particular rules that apply when coding a redefinition are summarised in section 3.5.

The optional BASED clause enables you to declare a special type of item known as a **based area** whose location is determined from the contents of the pointer whose data-name is name-2. It can only be used in the linkage section. Based area working is described in section 5 of this manual.

The statement establishes one or more elementary items whose attributes are determined by the picture clause (explained later). When the OCCURS clause is omitted just a single item is set up: when the clause is present space is allocated for a table of n such items.

3.2.2 Defining Group Items

Group items, which are subdivided, are introduced by a level 01 data item, followed by any number of subordinate (level 02 - level 49) items. Groups may be defined anywhere within the data division.

The level 01 item is defined by coding:

```
     01 data-name        [REDEFINES name-1] [OCCURS n]
                    [BASED name-2]
```

The quantities data-name, name-1 and name-2 should be supplied as symbols, as necessary. If it is not required to refer to the group explicitly the reserved word FILLER can be coded for the data-name. If an OCCURS clause is present the quantity n must be an unsigned positive integer.

The optional OCCURS clause allows you to set up a table, each of whose entries has the format of the data area described by the group. If the clause is omitted just a single occurrence of the group will be established. When the clause is present space is allocated for a table of n such groups.

Following the level 01 definition, the remaining subordinate items are declared by statements of the form:

```
     level-number   data-name [OCCURS n]     [picture clause]
```

The level-number must be two digits in the range 02 to 49 inclusive. The data-name should normally be a symbol, although if it is not required to refer to the item explicitly you may supply the reserved word FILLER instead. If the OCCURS clause is present n must be an unsigned positive integer.

If the picture clause is omitted then the item forms a **subgroup**, containing all the following items up to, but not including, the next item with an equal or lower level number. If the definition does not contain an OCCURS clause just a single occurrence of the subgroup will be established: where the clause is present space is allocated for a table of n such subgroups.

If the picture clause is coded then the item is an **elementary item**, treated in exactly the same way as a level 77 elementary item.

If the definition of a group or subgroup contains an OCCURS clause then it is termed a **repeating group**. It is a Global Cobol restriction that no subordinate definition within a repeating group can itself contain an OCCURS clause. This means that any tables defined by repeating groups are one-dimensional only.

3.2.3 Example

Figure 3.2.3 shows a level 01 group named AREA, with the level numbers of subordinate data items suitably indented so that the structure of the data is readily apparent. Data names such as C-2-1 have been chosen to highlight structural characteristics for the purposes of this example, and do not conform to recommended coding practice.

The lines with a picture clause (PIC X, explained later) all define elementary items, or in the case of B and C-2-2, a single entry of a table of such items. Note how a table, C-2-2, can itself be part of a subgroup. It could not of course be part of a repeating group, because no item within such a group can itself contain an OCCURS clause.

The example shows how subgroups and repeating groups can themselves contain subgroups. Subgroup C contains subgroup C-2, and repeating group D contains subgroup D-2. A subgroup can contain repeating groups (e.g. E contains E-2). The only combination which is not possible is for a repeating group to contain another repeating group, because of the OCCURS clause limitation.

```
01          AREA
  03        A                  PIC X       * ELEMENTARY ITEM
  03        B     OCCURS 20    PIC X       * TABLE OF 20
                                           * ELEMENTARY ITEMS
  03        C                              * SUBGROUP OF AREA
    05      C-1                PIC X       * ELEMENTARY ITEM IN C
    05      C-2                            * SUBGROUP IN C
      07    C-2-1              PIC X       * ELEMENTARY ITEM IN C-2
      07    C-2-2 OCCURS 5     PIC X       * TABLE OF 5
                                           * ELEMENTARY ITEMS IN C-2
  03        D     OCCURS 10                * REPEATING GROUP OF AREA
    05      D-1                PIC X       * ELEMENTARY ITEM IN D
    05      D-2                            * SUBGROUP IN D
      07    D-2-1              PIC X       * ELEMENTARY ITEM IN D-2
      07    D-2-2              PIC X       * ELEMENTARY ITEM IN D-2
  03        E                              * SUBGROUP OF AREA
    05      E-1                PIC X       * ELEMENTARY ITEM IN E
    05      E-2   OCCURS 8                 * REPEATING GROUP IN E
      07    E-2-1              PIC X       * ELEMENTARY ITEM IN E-2
```

```
        07    E-2-2            PIC X       * ELEMENTARY ITEM IN E-2
```

Figure 3.2.3 - Group Data Definition Example

3.3 Picture Clauses

The Global Cobol picture clause has the general format:

```
     PIC type[(qualifier)] [COMP]
```

where type indicates the type of item being declared, qualifier its precision or length and the COMP phrase applies to computational items only.

Programmers familiar with COBOL should note that the type [(qualifier)] construct can appear only once in each Global Cobol picture clause, and picture clauses such as PIC S9999 are **not** allowed.

3.3.1 Character Pictures

The picture clause for a character item is written:

```
     PIC X(length)
```

where length is the number of characters required and bytes occupied. If the length is 1, (1) may be omitted.

You can also use the special form:

```
     PIC X(?)
```

in which case the item must be initialised by subsequent VALUE statements so that the compiler can count the characters involved to determine the length to use. (See also 3.4.1.)

3.3.2 Display Numeric Pictures

For both computational and display numeric items the number of digits before the decimal point (p) and the number of digits after the decimal point (q) may be specified: p must be in the range 1 to 15 inclusive and q must be in the range 1 to 7 inclusive, and the sum of p and q must be no greater than 18.

A display numeric variable picture clause is written in one of the following formats:

(a) unsigned integer, p digits (size p bytes):

```
          PIC 9(p)
```

If p is 1, (1) may be omitted

(b) signed integer, p digits (size p + 1 bytes):

```
          PIC S9(p)
```

If p is 1, (1) may be omitted

(c) unsigned decimal, p digits before the point and q following the point (size p + q + 1 bytes):

```
PIC 9(p,q)
```

(d) as (c), but signed (size p + q + 2 bytes):

```
PIC S9(p,q)
```

Number of digits (p + q)	**Size in bytes**	**Approximate capacity**
1	1	±1.27 x 10^{2-q}
2	1	±1.27 x 10^{2-q}
3	2	±3.277 x 10^{4-q}
4	2	±3.277 x 10^{4-q}
5	3	±8.389 x 10^{6-q}
6	3	±8.389 x 10^{6-q}
7	4	±2.147 x 10^{9-q}
8	4	±2.147 x 10^{9-q}
9	4	±2.147 x 10^{9-q}
10	5	±5.497 x 10^{11-q}
11	5	±5.497 x 10^{11-q}
12	6	±1.407 x 10^{14-q}
13	6	±1.407 x 10^{14-q}
14	6	±1.407 x 10^{14-q}
15	7	±3.602 x 10^{16-q}
16	7	±3.602 x 10^{16-q}
17	8	±9.223 x 10^{18-q}
18	8	±9.223 x 10^{18-q}

Table 3.3.3 - Byte Length and Capacity of Computational Variables

3.3.3 Computational Pictures

The picture clause for a computational item is coded in the same way as that for a display numeric except that it is terminated by the phrase COMP. (As before, p must be in the range 1 to 15 inclusive, and q must be in the range 1 to 7 inclusive, and the sum of p and q must be no greater than 18):

(a) unsigned integer, p digits:

```
PIC 9(p) COMP
```

If p is 1, (1) may be omitted;

(b) signed integer, p digits:

```
PIC S9(p) COMP
```

If p is 1, (1) may be omitted;

(c) unsigned decimal, p digits before the point and q following:

```
PIC 9(p,q) COMP
```

(d) as (c), but signed:

```
PIC S9(p,q) COMP
```

A computational item is between 1 and 8 bytes in size; the number of bytes allocated depends on the number of digits it contains (p + q) and is given by column two of Table 3.3.3. Positive numbers are represented in true binary notation with the low address byte the most significant. The senior bit of this byte is interpreted as a sign bit: it is zero for positive numbers and one for negative numbers, which are held in two's-complement form.

Because arithmetic working is in binary, computational items are themselves binary and are capable of containing numbers greater than the size implied by p. Also, they can always hold negative numbers even if their picture clause states that they are unsigned. The actual range of values that a computational variable can assume is termed its **capacity**. It is capacity, rather than the format specified in the picture clause, which determines when overflow occurs during arithmetic operations and moves to computational fields.

Note, however, that if you attempt to DISPLAY a computational item containing a value which does not agree with its picture clause, overflow will occur. Similarly the picture information is used in validating computational input obtained using ACCEPT so it is impossible to input a computational value which does not agree with the receiving field's picture clause.

3.3.4 Pointer Pictures

A pointer is a data item, two bytes long, in which the location of a data item or program statement can be stored. The value of a pointer must be between 0 and 65535 (64K-1). It is represented by true binary notation. The low address byte of a pointer is the most significant and its senior bit is not interpreted as a sign bit but is considered to represent the 32K unit position.

To indicate that a data item is a pointer you code the picture clause:

```
     PIC PTR
```

A full description of how pointers are used appears in chapter 6.

3.3.5 Date Pictures

The picture clause for an item which will contain a date is coded as follows:

```
     PIC DATE
```

This causes the compiler to generate a PIC 9(6) COMP item whose value may be set up as specified in section 3.4.5.

3.3.6 Floating Point Pictures

The picture clause for a floating point variable (used with the scientific calculation routines covered in chapter 9 of the Global Development System Subroutines manual) is written:

```
     PIC FLT
```

This causes the compiler to generate a PIC X(6) variable to hold the floating point number, whose value may be set up as specified in section 3.4.6.

3.4 VALUE Clauses

A VALUE clause can be used to initialise an elementary item defined in working storage, providing that item is not part of a repeating group. Table 3.4 shows the 8 different formats that the VALUE clause can assume.

FORMAT	VALUE CLAUSE SYNTAX
(a)	VALUE "character string"
(b)	VALUE #hexadecimal string
(c)	VALUE numeric string
(d)	VALUE ZERO
(e)	VALUE LOW-VALUES
(f)	VALUE HIGH-VALUES
(g)	VALUE SPACE or VALUE SPACES
(h)	VALUE symbol

Table 3.4 - Value Clause Formats

Format (a) is used to set up character and display numeric items. Character strings are described in 2.1.3.

Format (b) can be used to set up any type of item. Hexadecimal strings are described in 2.1.7.

Format (c) is used to set up computational items. The numeric string must obey the conventions described in 2.1.5.

In format (e) LOW-VALUES always sets each byte of the item to binary zeros. HIGH-VALUES in format (f) sets each bit of each byte to 1. The interpretation of the ZERO phrase in format (d) depends on the type of item being initialised.

Format (h) is used to set up pointer items.

3.4.1 VALUE Clauses for Character Items

Elementary character items may be initialised using formats (a), (b) and (d) to (g). Formats (a) and (b) only initialise the number of bytes specified by the string, whereas formats (d) to (g) initialise the whole of the item to ASCII zeros, low values, high values, or ASCII blanks respectively. Several VALUE clauses may be specified following a data definition, in which case the values are concatenated together. For example:

```
77    A     PIC X(10)
            VALUE "ABC"
            VALUE "XYZ"
            VALUE SPACES
```

causes the first 6 bytes of A to be set to ABCXYZ and the remaining 4 bytes to be set to blanks.

Note that VALUE SPACES is not necessary in this example since uninitialised rightmost bytes of character items are set to blanks by default as explained in 3.4.5.

If the character variable being initialised contains an OCCURS clause it is treated as a single long character string during VALUE clause processing.

If you wish to set up a constant text string as a character item then you may code the picture clause as:

```
PIC X(?)
```

This means that the length of the item is to be the sum of the lengths of the VALUE clauses which follow it, and eliminates the necessity to count up the length of such strings. For example:

```
77    A      PIC X(?)
             VALUE "INSUFFICIENT STOCK REMAINING"
             VALUE " - CANNOT PROCESS ORDER"
```

means that A will be 51 bytes long.

3.4.2 VALUE Clauses for Display Numeric Items

For elementary display numeric items formats (a), (b) and (d) to (f) are valid.

Format (a) will convert the string specified to a standard numeric string, as described in 2.1.6, and initialise the item to this value.

Format (b) initialises the item to the value specified, left justified and padded with LOW-VALUES if necessary.

Formats (d) to (f) initialise every byte of the item to ASCII zero, LOW-VALUES, or HIGH-VALUES respectively.

Note that if the data definition contains an OCCURS clause a separate VALUE clause must be coded to initialise each occurrence: the first VALUE clause sets up occurrence 1, the second occurrence 2, and so on.

3.4.3 VALUE Clauses for Computational Items

For elementary computational items formats (b) to (f) are valid.

If format (b) is used the hexadecimal string specified must establish every byte of the item and no more.

Format (d) initialises every byte of the item to binary zeros. The other formats are described in the introduction to this section.

Note that if the data definition contains an OCCURS clause a separate VALUE clause must be coded to initialise each occurrence: the first VALUE clause sets up occurrence 1, the second occurrence 2, and so on.

3.4.4 VALUE Clauses for Pointer Items

A format (b) VALUE clause can be used to give a pointer an absolute value, and format (h) to initialise a pointer to address a specified symbol. When a pointer is declared in working storage, and does not belong to a repeating group, you may initialise it by coding a value clause of the form:

```
     VALUE symbol
or:
     VALUE #hexadecimal string
```

The symbol may be a data name, filename, mapname, paragraph name, section name or entry name appearing in the current compilation. It may also be a symbol defined in another compilation which appears as the operand of a GLOBAL statement within the current compilation.

Data names, filenames and mapnames used in a pointer value clause must be defined in working storage, **not** the linkage section.

If the symbol is a data name the pointer is initialised to address the first byte that the associated data item occupies. For a filename or mapname the pointer is set up to address the first byte of the corresponding file or map definition.

If the symbol is a paragraph name, section or entry name, the pointer is initialised to address the first executable instruction within the paragraph, section or entry point it identifies.

When a VALUE clause is not coded for a pointer declaration appearing in the initialised part of working storage (following the first value clause, MD or FD), the pointer is set to #FFFF.

3.4.5 VALUE Clauses for Date Items

A format (b) or (c) VALUE clause can be used to give a date field a specified numeric or hexadecimal value. A format (a) VALUE clause, where the character string is a valid long or short date in dd/mm/yyyy format, will cause the date field to be initialized to the internal format representation of the date specified.

3.4.6 VALUE Clauses for Floating Point Items

A format (b) VALUE clause can be used to set a floating point field to a particular hexadecimal value. A format (a) VALUE clause, where the character string is a valid floating point number (as defined in chapter 9 of the Global Development System Subroutines manual) will cause the floating point field to be initialised to the internal format representation of the number specified.

3.4.7 Bytes Not Initialised by Value Clauses

Bytes of working storage which are not initialised by VALUE clauses are treated as follows:

Any even number of bytes preceding the first location initialised, either by a VALUE clause, or by the first map definition or file definition, remain uninitialised and are left undisturbed by the loader when the Global Cobol program is brought into memory prior to execution. If the first initialized byte is at an odd address, the preceding byte will be initialized to binary zero.

All bytes following the first VALUE clause, map, or file definition, which are not themselves set up by VALUE clauses, are initialised to binary zeros **if** the elementary item containing the bytes belongs to a repeating group. If the item does not, then the bytes are set up according to the data type established by its picture clause:

- Uninitialised bytes within character items are set to ASCII blanks;
- Uninitialised display numeric items are set to ASCII zeros. (Note that this is an invalid value unless the item is an unsigned integer);
- Uninitialised computational items are set to binary zeros;
- Uninitialised pointer items are set to HIGH-VALUES;
- Uninitialised date items are set to zero, which is not a valid internal format date;
- Uninitialised floating point items are set to SPACES. Note that this is a completely illegal floating point value, so you should either always initialise such items explicitly or ensure that a sensible value is placed in them before they are used.

Data items appearing within a redefinition do not cause any initialisation to take place.

3.4.8 VALUE Clause Restrictions

A VALUE clause is ignored if it is coded by mistake in an attempt to initialise an elementary item appearing in the linkage section, an external section, a repeating group, or a redefinition. The compiler will output a warning message except for items which are within copy books, when no warning is generated. (This allows copy books to contain VALUE statements yet still be used, if required, in the linkage section, an external section, or a redefinition.)

3.5 Redefinitions

A redefinition is a level 01 group or level 77 elementary item (A, say) which redefines the storage occupied by another data item (B, say). A redefinition is introduced using the REDEFINES clause in the data definition:

```
01    A     REDEFINES B [OCCURS n]
  03 etc.
  03 etc.
```

or:

```
77    A     REDEFINES B [OCCURS n] PIC etc
```

3.5.1 Redefinition Rules

B, the data item being redefined, may itself have been declared as level 77, level 01, or indeed as any of the subordinate levels from 02 to 49. It may also be the filename or mapname labelling a file or map definition. However, it must have been defined **previously** in the data division, or be one of the in-built system variables described in the Global Development System Subroutines manual.

A and B must either both be in working storage, or belong to the same external section, or both be defined in the linkage section. The redefinition of a system variable must reside in the linkage section.

The size of A in bytes should be no greater than that of B. If this rule is broken, the compiler will flag the first subordinate item of A occupying storage outside that allocated B with a warning message. If either A or B is a data definition with an OCCURS clause, then the size of the item for the purpose of this comparison is considered to be the length in bytes of the total table defined by the OCCURS clause.

3.5.2 Redefinition Implications

VALUE clauses belonging to subordinate items within a redefinition will be ignored. A warning message will be output during compilation unless the item is within a copy book.

If a redefinition in the linkage section redefines a subordinate item (i.e. one with a level number between 02 and 49 inclusive), or a system variable, then the data name of the redefinition may not appear in the USING clause of an ENTRY statement.

4. The Procedure Division

4.1 Structure

The procedure division contains the executable statements of a Global Cobol program. It begins with the header:

```
PROCEDURE DIVISION
```

which must be coded on a new line. It is terminated by the trailer:

```
ENDPROG
```

also on a new line.

When a main program is executed control initially passes to its entry point, the first statement of the procedure division. When a subroutine is invoked, linkage edited with the main program, control is passed via the CALL statement to an entry point defined by an ENTRY statement.

The procedure division is made up of sections and paragraphs. The first statement following the PROCEDURE DIVISION statement should be a SECTION statement or an ENTRY statement.

4.1.1 Sections and Paragraphs

A section is introduced by the statement:

```
SECTION section-name
```

where section-name is a symbol as defined in 2.1.2.

A paragraph name is written as:

```
symbol.
```

It may either appear on a line by itself or begin the statement it labels.

4.1.2 Sections Entered by a CALL Statement

If a section is to be entered by a CALL statement an ENTRY statement must be coded as the first statement of the section, immediately preceding the SECTION statement if present. The format of the ENTRY statement is:

```
ENTRY entry-name [USING A B ...]
```

Where entry-name is a global symbol as defined in 2.1.2. As well as being unique within the containing program, it must also be different from any other global symbols participating in the same linkage edits. Note that the entry name cannot be the same as the name of the section which it precedes. When an ENTRY statement is used the SECTION statement is optional is usually omitted.

Each operand A, B, etc. in the USING clause, if it appears, must be a filename, mapname, level 01 group item, level 77 elementary item, or a redefinition of one of these, coded in the **linkage section.** It may not be a subordinate (level 02-49) item, nor the redefinition of such an item.

Up to 7 operands may appear in the USING clause. The details of parameter passing are explained as part of the description of the CALL statement, which is used to transfer control to an entry point defined by an ENTRY statement.

4.1.3 Entry and Section Identifiers

Normally the first five characters of each entry name or section name are generated as part of the code expanded by the ENTRY or SECTION statement involved. These section and entry identifiers are available to Global at run-time and are output if the program is terminated in error, to identify the routines and sections in the control path at the time of failure.

To ensure that the diagnostics provided by Global are not ambiguous it is good practice to make the first **five** characters of each entry name and section name unique within a program.

4.2 Operands

Operands in procedure division statements are usually represented by the capital letters A, B, C ... in the language description. In general, an operand can be a simple variable, an indexed variable, or a literal, although there may be restrictions in particular cases. Sometimes an operand must be the symbol used to label a file definition or map definition. This type of operand is normally represented by the word filename or mapname rather than a capital letter. You should only code a filename or mapname where the language description explicitly states it to be allowed or required.

A picture clause is associated with each operand (apart from a filename or mapname) so that the compiler can determine its data type, ie whether it is character, display numeric, computational or pointer. The picture clause also contributes additional, type-dependent, information such as the length of a character operand or the number of decimal places in a computational operand.

An "understood" picture clause can be considered to be associated with groups, subgroups, literals and figurative constants, as explained below.

4.2.1 Simple Variables

A simple variable is the data-name of a non-repeating group or subgroup, or an elementary item without an OCCURS clause. A group or subgroup is treated as though it had a picture clause:

```
     PIC X(n)
```

where n is its length in bytes.

If the data definition possesses an OCCURS clause, or belongs to a repeating group, you can only reference individual entries from the table thus defined. This is achieved by the use of indexed variables.

4.2.2 Indexed Variables

Indexed variables are coded:

```
     A(B)
```

A must be the data name of a repeating group, a subordinate item within such a group, or an elementary item with an OCCURS clause. If A is an elementary item the picture clause associated with A(B) is simply the picture clause of A itself. If A is a group or subgroup the picture clause is taken as:

```
     PIC X(n)
```

where n is the length in bytes of a single occurrence of A.

B must be a computational simple variable or a positive integer literal. At run-time if B contains a value whose integral part, i, is in the range 1 to 32,767 then A(B) references the i'th occurrence of A within a table established by an OCCURS clause.

If (B) is omitted then a reference to A will be treated by the compiler as though A(1) had been coded. However, a warning message will be output whenever such a reference occurs.

Note that Global Cobol performs **lower** bound checking, inasmuch as it checks that i is in the range 1 to 32,767 and terminates the job with an error if this is not the case. Upper bound checking does not take place. If i is greater than the value specified in the OCCURS clause the result will be unpredictable.

4.2.3 Computational Literals

A computational literal can be supplied in the place of a read-only computational variable. It is coded like a numeric string, as defined in 2.1.5. Therefore valid computational literals are:

```
     -3   3.14159   +1246    -0.120        .7
```

A special type of computational literal is the **integer literal** which is coded as an integer in the range -32768 to +32767, the plus sign being optional. It is treated as though it had a picture clause:

```
     PIC S9(4) COMP
```

irrespective of its actual size. The picture clause associated with non-integer computational literals is determined from the presence or absence of a minus sign, and the number of digits before and after the decimal point, if any. Leading zeros are ignored, except in the units position, as are trailing zeros to the right of the decimal point, except when such a zero appears in the tenths position. For example:

```
     32768          PIC 9(5) COMP
     3.14159   PIC 9(1,5) COMP
     -0.120         PIC S9(1,2) COMP
     3.0       PIC 9(1,1) COMP
```

4.2.4 Character Literals

A character literal can be supplied in the place of a read-only character variable. It may be coded as either a character string or a hexadecimal string:

```
     "character string"
or:
     #hexadecimal string
```

It is treated as though it had a picture clause:

```
     PIC X(n)
```

where n is its length in bytes. For example:

```
     "ABC"
and:
      #414243
```

are 3-byte character literals with identical values.

4.2.5 Figurative Constants

The reserved words:

```
     HIGH-VALUES    SPACE
     LOW-VALUES     SPACES
```

are figurative constants, and can be used in MOVE statements and conditions wherever a character literal would be valid. They represent character strings containing bytes which are:

- all #FF, i.e. with every bit set to 1 (HIGH-VALUES)
- all #00, i.e. with every bit set to 0 (LOW-VALUES)
- all #20, i.e. with every byte an ASCII blank (SPACE or SPACES)

The length of a figurative constant is set up to match that of the corresponding variable in the MOVE statement or condition in which it appears. Thus it can be considered to possess a picture clause of the form:

```
     PIC X(n)
```

where n is that length, in bytes.

4.2.6 Variables

In the descriptions which follow, the term "variable" is used as a shorthand for "simple or indexed variable".

4.3 Arithmetic Statements

The permissible arithmetic statements are summarised in Table 4.3:

STATEMENT	**ACTION**
ADD A TO B [ROUNDED]	B + A ---> B
ADD A TO B GIVING C [ROUNDED]	B + A ---> C
SUBTRACT A FROM B [ROUNDED]	B - A ---> B

SUBTRACT A FROM B GIVING C [ROUNDED]	B - A ---> C
MULTIPLY A BY B [ROUNDED]	B x A ---> B
MULTIPLY A BY B GIVING C [ROUNDED]	B x A ---> C
DIVIDE A INTO B [ROUNDED]	B / A ---> B
DIVIDE A INTO B GIVING C [ROUNDED]	B / A ---> C

Table 4.3 - Arithmetic Statements

In arithmetic statements A, B and C may be computational variables. Alternatively A may be a computational literal. B may be a computational literal only if the GIVING clause is present. C may be a display numeric variable. No other combinations are valid.

4.3.1 Arithmetic Truncation and Rounding

If the result of an arithmetic operation contains more digits following the decimal point than are contained in the receiving variable then the extra digits will be truncated. However, if the ROUNDED phrase was coded and the most significant digit thus truncated was 5, 6, 7, 8 or 9 then the least significant digit of the receiving variable will be augmented by one, otherwise it will remain unchanged.

4.3.2 Overflow

Any arithmetic statement will suffer overflow if the result exceeds the capacity of a receiving computational variable, or does not satisfy the picture of a receiving display numeric variable. Overflow will also take place if the capacity of internal Global fields used to hold intermediate results is exceeded. More specifically, let the number of decimal places in the two operands A and B be a and b respectively, and the number of places in the receiving field (B or C) be r. Then overflow will occur in the following circumstances:

- if the magnitude of the result of an ADD or SUBTRACT statement exceeds $9.2 \times 10^{18-p}$ approximately, where p is the greater of a and b;

- if the magnitude of the result of a MULTIPLY statement exceeds $9.2 \times 10^{18-a-b}$ approximately;

- if the dividend (B) of a DIVIDE statement exceeds $9.2 \times 10^{18-a-r}$ approximately, or if the magnitude of the divisor (A) is greater than $2.1 \times 10^{9-a}$ approximately, or if the magnitude of the quotient is greater than $2.1 \times 10^{9-r}$ or $2.1 \times 10^{9-b}$ approximately.

You may check for overflow by coding the ON OVERFLOW statement (4.6) as the statement **immediately following** the arithmetic statement. If no such ON OVERFLOW statement is coded and overflow occurs the program will be terminated with an error.

If an arithmetic statement suffers overflow it is suppressed and the receiving variable remains unchanged.

4.3.3 Examples

Suppose A and B are declared as PIC 9(2,1) COMP and PIC 9(2,2) COMP respectively. Then:

```
     ADD 1 TO B GIVING A ROUNDED
```

has the effect, for B = 3.42, of:

```
     1 + 3.42 (=4.42) yielding A = 4.4
```

Alternatively, for B = -4.75, we have:

```
     1 + -4.75 (= -3.75) yielding A = -3.8
```

Finally, consider the division of B by A, where A = 1.1 and B = 0.28. The Global Cobol:

```
     DIVIDE A INTO B GIVING A ROUNDED
```

results in:

```
     0.28/1.1 (=0.254545...) yielding A = 0.3.
```

To summarise, a useful rule to remember is that truncation is towards zero and that rounding, when the digit involved is 5 or more, is away from zero.

4.3.4 The COMPUTE Statement

The COMPUTE statement, which is used to perform floating point calculations, is covered in detail in section 9 of the Global Development System Subroutines Manual.

4.4 The MOVE Statement

The MOVE statement is coded:

```
     MOVE A TO B [C D...]
```

where A is a literal, figurative constant or variable and B, C, D..., etc are variables. A maximum of seven variables may follow the word TO. Where more than one variable follows TO the content of A is moved, in turn, to each of these variables. of simple moves had been coded.

In the simple move, MOVE A TO B, execution depends on the data type of each operand. Table 4.4 shows that, of the 16 types of move theoretically possible, 8 are supported in Global Cobol.

TO B **MOVE A**	**CHARACTER**	**COMPUTATIONAL**	**DISPLAY NUMERIC**	**POINTER**
CHARACTER	YES	NO	YES	NO
COMPUTATIONAL	NO	YES	YES	NO
DISPLAY NUMERIC	YES	YES	YES	NO
POINTER	NO	NO	NO	YES

Table 4.4 - Valid Data Types for MOVE A TO B

4.4.1 Character to Character Move

The contents of A are moved to B, byte by byte. The transfer takes place one byte at a time, from left to right. The leftmost (low location) byte of A is copied to the leftmost byte of B, then the next byte of A is copied to the next byte of B, and so on. If A is shorter than B the move operation pads B with rightmost blanks. If B is shorter than A movement stops once B has been filled: in this case **character truncation** (as distinct from the arithmetic truncation described in 4.3.1) is said to have occurred.

In addition, A may be a figurative constant, and you may code:

```
     MOVE HIGH-VALUES TO B          * EACH BYTE SET TO #FF
     MOVE LOW-VALUES TO B                * EACH BYTE SET TO #00
     MOVE SPACE TO B                     * EACH BYTE SET TO
     MOVE SPACES TO B               * ASCII BLANK (#20)
```

The A and B fields involved in a character to character move may overlap. Indeed, the following example shows how overlapping fields may be used to set every byte of a long field to a particular value, in this case ASCII E:

```
     01   A
       03 A1   PIC X
       03 B    PIC X(999)
     .
     .
     .
          MOVE "E" TO A1
          MOVE A TO B
```

This particular operation is called a **ripple move** and is useful for initialising large data items.

4.4.2 Character to Display Numeric Move

This is treated like a character to character move as described in 4.4.1. You must be careful that the number of bytes in the B field is sufficient since the move can result in the loss of digits if character truncation takes place.

4.4.3 Computational to Computational Move

A is transferred to B taking account of the precision of the two operands. If B is of lower precision than A the difference in precisions will result in arithmetic truncation. If B is of insufficient capacity to contain A, overflow will occur.

4.4.4 Computational to Display Numeric Move

A is converted to standard display numeric format according to the picture of B. If A is too large, or is negative when B is unsigned, overflow will occur.

4.4.5 Display Numeric to Character Move

This is treated like a character to character move as described in 4.4.1. You must be careful that the number of bytes in B is sufficient or the move can result in the loss of digits if character truncation takes place.

4.4.6 Display Numeric to Display Numeric Move

The numeric string at A is converted to standard numeric string format in B. If A is too large, or does not contain a valid numeric string conforming to the picture clause of A, or is negative when B is unsigned, overflow will occur.

4.4.7 Display Numeric to Computational Move

A is converted to binary according to its picture clause and the result is stored in B. Arithmetic truncation will take place if the precision of A is greater than that of B. If A is too large, or does not contain a valid numeric string conforming to the picture clause of A, overflow will occur. It will also take place if A is valid but B is of insufficient capacity to contain the result.

4.4.8 Pointer to Pointer Move

The contents of the two-byte pointer at A are transferred, unchanged, to the two-byte pointer at B.

4.4.9 Overflow

Overflow can occur, for the reasons described above, in the following types of move operation:

- computational to computational;
- computational to display numeric;
- display numeric to display numeric;
- display numeric to computational

You may check for overflow during a simple move, or the last operation of a compound move, by coding the ON OVERFLOW statement (4.6) immediately following the MOVE statement. If no such ON OVERFLOW statement is coded and overflow occurs the program will be terminated in error. This will also occur if ON OVERFLOW follows a compound move but the operation suffering overflow was not the last.

If a move operation suffers overflow it is suppressed and the receiving variable remains unchanged.

4.5 Transfer of Control Statements

4.5.1 The GO TO Statement

GO TO unconditionally transfers control to a paragraph or section. It is coded:

```
GO TO A
```

where A is the name of a paragraph or section, or the name or a pointer set to address the first executable instruction of a paragraph or section.

4.5.2 The GO TO DEPENDING ON Statement

GO TO DEPENDING ON provides a switch capability. It is coded:

```
     GO TO DEPENDING ON data-name
          TO label-1
          TO label-2
          " "
          " "
          TO label-n
```

where label-1, label-2, label-n are paragraph or section names.

The data-name must be the name of a computational variable whose integral part, i, is in the range 1 to n when the statement is executed. In this case control is passed to label-i as if a:

```
     GO TO label-i
```

statement had been executed. If i is greater than n the results will be unpredictable since there is no upper bound run-time range checking.

4.5.3 The PERFORM Statement

The PERFORM statement passes control to a paragraph or section. It is coded:

```
     PERFORM A
```

where A is the name of a paragraph or section, or the name of a pointer set to address the first executable instruction of a paragraph or section.

The statements beginning at the indicated section or paragraph are executed until control is returned to the statement following the PERFORM by an EXIT statement (see 4.5.5).

4.5.4 The CALL Statement

The CALL statement passes control to an entry point identified by the entry-name appearing in an ENTRY statement. The ENTRY statement may reside either in the current compilation, or in a compilation to be linkage edited with it. CALL is coded:

```
     CALL A [USING B C ...]
```

where A is an entry name, or the name of a pointer set to address the first executable instruction at the entry point. A maximum of 7 parameters may be passed in the optional USING clause. If the CALL statement does not possess a USING clause then neither must the ENTRY statement. Otherwise the parameters in the two USING clauses involved must correspond one for one. (There is a special technique available for creating subroutines which can accept a variable number of parameters, but in the interests of simplicity its description is contained in the Global Development System Subroutines Manual.)

Each operand of a CALL statement's USING clause may be a variable, literal, filename, mapname, paragraph name, section name, or entry name. However, the figurative constants HIGH-VALUES, LOW-VALUES, SPACE and SPACES must **never** appear.

Each operand in the target ENTRY statement must be of the same type as the corresponding operand of the CALL statement, and must be defined in the linkage section. An ENTRY operand may, as appropriate, be a filename, mapname, or the redefinition of one of these. It may **not** be a subordinate (level 02-49) item, nor the redefinition of such an item.

When a **variable** is passed as a parameter the corresponding ENTRY operand is a level 77 item or level 01 group describing the storage area the variable occupies.

If an **integer literal** is passed the corresponding ENTRY operand should be a level 77 item or level 01 group describing a single PIC S9(4) COMP field. This will overlay the integer literal, and must therefore be read-only.

If a **character literal** is passed the corresponding ENTRY operand is a level 77 item or level 01 group describing the character string. This will overlay the character literal and must therefore be read-only.

When a **filename** or **mapname** is passed as a parameter the corresponding ENTRY operand is a file definition or map definition. The appropriate file or map processing statements can then be used to manipulate the passed file or map in the normal way within the called routine.

When a **paragraph name, section name** or **entry name** is passed, the corresponding ENTRY operand should be a level 77 item or level 01 group describing a single PIC PTR field. This pointer will address the paragraph, section or entry point in question, and must remain read-only. The pointer form of the GO TO, PERFORM or CALL statement can then be used to invoke the passed paragraph, section or entry point from the called routine.

4.5.5 The EXIT Statement

The EXIT statement causes control to be returned to the statement following the last **outstanding** PERFORM or CALL statement executed.

When a PERFORM or CALL is executed Global Cobol remembers the address of the following statement in an internal stack. The previous contents of the stack are "pushed down" so that a large number of outstanding PERFORMs or CALLs can be nested.

When an EXIT is executed the top item in the stack is used to determine the statement to which control is to be passed: this stack item is then made available for re-use and the stack contents are "popped up".

The EXIT statement is therefore the **dynamic** end of a sequence of code entered by a PERFORM or CALL. This provides additional flexibility compared with COBOL. In addition an explicit EXIT statement, rather than an implied exit at the end of a paragraph, section or group of sections, makes code easier to follow and facilitates structured programming.

Note that an EXIT statement issued from the highest level of a program, when there is no outstanding PERFORM or CALL, is equivalent to a STOP RUN.

4.5.6 The STOP RUN Statement

The STOP RUN statement can be coded anywhere within a main program or subprogram. It returns control to Global System Manager indicating that the program has completed normally.

4.5.7 The FINISH Statement

The FINISH statement can be coded anywhere within a DO loop (as described in 4.6.2). It has the effect of transferring control to the statement following the next ENDDO, thereby exiting from the loop in a clear and structured manner.

4.5.8 Prefixed Transfer of Control Statements

The transfer of control statements:

```
     GO TO label
     PERFORM label
     EXIT
     FINISH
     STOP RUN
```

may be prefixed by 'IF condition', 'ON OVERFLOW', 'ON EXCEPTION', and by 'ACCEPT...NULL'. (These clauses are described in 4.6 and 4.7.) The result is to make execution of the transfer of control statement dependent on the condition defined by the prefixing clause. For example:

```
     IF A ZERO GO TO LAB2

     MOVE A TO B
     ON OVERFLOW PERFORM AA223

     READ A INTO B
     ON EXCEPTION EXIT

     ACCEPT A NULL STOP RUN
```

4.6 Conditional and Iterative Statements

4.6.1 Format of Conditional Structures

There are two basic formats for conditionals. Format 1:

```
     |IF condition |
     |ON OVERFLOW |
     |ON EXCEPTION |
     |ACCEPT...NULL |
     [OR statement(s) | AND statement(s) ]
     ............                  |statements to be executed if
     ............                  |condition true
     ............                  |(group A)
     [ELSE
     ............
     ............                  |statements to be executed if
     ............                  |condition false
     ............ ]                     |(group B)
     END
```

and format 2:

```
     |IF condition |
     |ON OVERFLOW | transfer of control statement
     |ON EXCEPTION |
     |ACCEPT...NULL|
```

The possible transfer of control statements in format 2 are defined in 4.5.8.

If, in format 1, the statements in group A are not terminated by a GO TO, GO TO DEPENDING ON, EXIT or STOP RUN then, when the ELSE statement is encountered control will be passed to the statement following END. If the ELSE statement is missing there are no group B statements. In this case if the group A statements are not terminated by an unconditional transfer of control, when the END statement is met it will be ignored and execution will continue with the next statement.

Similarly, if the group B statements are not terminated by a GO TO, GO TO DEPENDING ON, EXIT or STOP RUN then, when their last statement has executed, control will drop through the END statement and continue with the next statement.

The statements ELSE and END must be coded on new lines and cannot be combined with other statements.

Format 1 conditionals may be nested up to 32 times and may contain iterative structures (explained below). An END statement terminates the most recent conditional statement which has not yet been matched by an END statement. An ELSE statement refers to the most recent conditional statement which has not yet been matched by an END statement.

Format 2 conditional statements may appear within format 1 conditionals. Format 2 statements always generate less code than the equivalent logic coded using Format 1.

4.6.2 Format of Iterative Structures

There are four formats for iterative structures (or DO loops).

Format 1:

```
     DO
     ........
     ........                |statements to be executed
     ........
     ENDDO
```

Format 2:

```
     DO WHILE condition
     [OR statement(s) | AND statement(s)]
     .........
     .........               |statements to be executed while
     .........                    |the condition remains true
     ENDDO
```

Format 3:

```
     DO UNTIL condition
     [OR statement(s) | AND statement(s)]
     ........
     ........                |statements to be executed until
     ........                |the condition becomes true
     ENDDO
```

Format 4:

```
     DO FOR X = A [TO B] [STEP C]
     ........                |statements to be executed while X
     ........                |stays  in  range  and  condition  remain
     true
     ........
     ENDDO
```

In format 1 the enclosed statements are executed over and over until some transfer of control (such as GO TO or FINISH) causes an exit from the loop.

In format 2 the enclosed statements are executed zero or more times, while the condition remains true. The condition is tested before the first iteration, and then before each subsequent iteration. As soon as it is not satisfied, the statement immediately following the ENDDO receives control; therefore it is possible that the statements between DO and ENDDO may not be executed at all.

Format 3 is similar to format 2 except that the enclosed statements are executed only as long as the condition remains false.

In format 4 the processing is more complex. Here X (the loop count) must be a computational variable, A (the initial value) must be a computational variable or literal, B (the limit) must be a computational variable or literal if the TO clause is coded and C (the step length) must be a numeric literal if the STEP clause is coded. The variable X is first set to the value in A. This is compared with the limit B, if specified, and provided all is well the enclosed statements are executed. When the ENDDO is encountered the value in X is increased by the step length C, or 1 if no STEP clause was coded, and then processing resumes with a comparison with the limit.

If the step length C is not negative, or is omitted, then execution will continue provided that the value of X is not greater than the value of the limit B. If the step length is negative then execution will continue provided that the value of X is not less than the value of the limit B.

If the STEP clause is omitted then a step length of 1 is assumed. If the TO clause is omitted then no limit checking takes place, and execution will continue until a transfer of control statement causes execution to leave the DO loop, or the variable X overflows.

DO loops may be nested up to 16 times and may contain conditional structures. An ENDDO statement terminates the most recent DO statement which has not yet been matched by an ENDDO statement.

CONDITION CLAUSE	**IDENTICAL EQUIVALENT**	**RESTRICTIONS**
A EQUAL B A NOT EQUAL B A LESS B A NOT LESS B A GREATER B A NOT GREATER B	A = B A NOT EQUAL B A < B A NOT < B A > B A NOT > B	A and B must both be either computational, character, or pointer items. Alternatively, A may be display numeric if B is computational. One, but not both of the operands may be a literal, or in the case of a character comparison, a figurative constant.
A SPACES A NOT SPACES A HIGH-VALUES A NOT HIGH-VALUES A LOW-VALUES A NOT LOW-VALUES	A SPACE A NOT SPACE	A must be a character variable
A ZERO A NOT ZERO A POSITIVE A NOT POSITIVE A NEGATIVE A NOT NEGATIVE		A must be a computational or display numeric variable
A NUMERIC A NOT NUMERIC		A must be a display numeric variable

Table 4.6.3 - The Condition Clause

4.6.3 The Condition Clause

The condition clause that appears in the IF and DO statements may assume any of the formats summarised in the left-hand column of Table 4.6.3. The mathematical symbols =, > and < can be coded in the place of the words EQUAL, GREATER and LESS, respectively, and the figurative constants SPACE and SPACES are synonymous. This is reflected in the middle column of the table. The conditions are divided into four groups, depending on the restrictions which apply to the operand or operands.

Comparison of display numeric and computational items obeys the normal rules of arithmetic. The comparison of character variables and pointers takes place, byte by byte, from the leftmost byte at the low address to the rightmost byte at the high address. The bytes being compared are treated, for the purposes of the comparison, as 8-bit unsigned numbers (See Appendix A for the numeric values corresponding to ASCII characters.) If two strings of unequal length are compared, the shorter will be considered to be extended to the right with ASCII blanks. When a figurative constant is involved it is treated as a character string of exactly the same length as the variable with which it is being compared.

The condition:

```
A NUMERIC
```

is true only if the variable A contains a valid numeric string, as defined in 2.1.5, which is compatible with the picture clause of A.

The following three examples show the use of condition clauses in a format 1 conditional, format 2 conditional and iterative structure:

```
     IF COUNT > 100                * FORMAT 1 CONDITIONAL
          MOVE 0 TO COUNT
     END

     IF NAME SPACES GO TO ASKRTN   * FORMAT 2 CONDITIONAL

     DO WHILE COUNT POSITIVE       * ITERATIVE STRUCTURE
          ADD -1 TO COUNT
          PERFORM CALC
     ENDDO
```

4.6.4 Compound Conditions

Compound conditions may be established by coding groups of one or more OR statements or AND statements immediately following one of these nine initial conditional statements:

```
     IF condition             ACCEPT... NULL      DO FOR condition
     ON OVERFLOW         ON EXCEPTION        DO WHILE condition
     ON NO OVERFLOW ON NO EXCEPTION     DO UNTIL condition
```

The format of a compound condition is either:

```
     initial conditional statement
     OR condition-1
     .
     .
     OR condition-n
```

or:

```
     initial conditional statement
     AND condition-1
     .
     .
     AND condition-n
```

The first compound condition is true if **any** of the condition clauses it contains is satisfied, but the second form is only true if **all** of the constituent conditions hold. Once sufficient conditions have been evaluated to establish the result of the compound condition, no further conditions are evaluated. It is not allowable to mix AND and OR statements in the group following the initial conditional statement, and if you attempt to do so the compiler will flag any out-of-place statement in error.

Here are two examples using compound conditions:

```
     IF COUNT > 100
     OR COUNT NEGATIVE             * COUNT ZEROISED IF NOT
          MOVE 0 TO COUNT          * BETWEEN 0 AND 100
     END

     DO WHILE COUNT POSITIVE
     AND NAME NOT SPACES
     AND ERRFLAG ZERO
          ADD -1 TO COUNT
          PERFORM CALC
     ENDDO
```

4.6.4 The ON OVERFLOW Statement

The ON OVERFLOW statement is used for checking for the overflow condition which may result following an arithmetic statement, or a MOVE, DISPLAY or EDIT statement.

ON OVERFLOW must be coded as the statement immediately following the one generating the condition to be tested. If this is not done and an overflow condition arises the program will be terminated in error.

Note that the second of the two examples which follow shows the coding required to simply ignore an overflow condition:

```
     ADD COUNT TO ACCUM            * SET ACCUM NEGATIVE
     ON OVERFLOW                   * IF ANY COUNT NOT
     OR COUNT NEGATIVE             * POSITIVE AND IN RANGE
          MOVE -1 TO ACCUM
     END
     ADD COUNT TO ACCUM            * IGNORE ANY
     ON OVERFLOW                   * OVERFLOW CONDITION
     END
```

4.6.5 The ON EXCEPTION Statement

STATEMENT	**EXCEPTION NUMBER RETURNED IN $$COND**	**MEANING**
LOAD	1 2 3	irrecoverable I/O error program file not found program file too large
SCAN	1	high or equal key not present
SEARCH	1	equal key not present

Table 4.6.5 - Statements Subject to Exceptions (apart from CALL, PERFORM and file processing statements)

The ON EXCEPTION statement is used for checking for the exception condition that can be signalled in response to one of the statements listed in Table 4.6.5, as the result of CALLing or PERFORMing certain types of routine, file processing or console I/O operations. ON EXCEPTION must be coded as the statement immediately following the one generating the condition to be tested. If this is not done and an exception condition arises the program will be terminated in error.

A CALL or PERFORM statement may not necessarily be liable to an exception condition. Whether this is the case or not depends on the routine invoked by the statement. If it inevitably returns control by means of an EXIT statement coded as described in 4.5.5, then it will always signal normal completion, and the invoking CALL or PERFORM statement will never suffer an exception. However, it is possible (and often very useful) to write routines which generate exception conditions to indicate when special circumstances have arisen. This is explained in the Global Development System Subroutines Manual.

Global Cobol system routines nearly always signal exceptions in special circumstances. The meaning of the different conditions generated varies, of course, from routine to routine, and are defined as part of each routine's description.

When an exception occurs the system variable $$COND, the condition number, is set to a positive value. This is normally 1, except when the same statement can generate an exception for a variety of different reasons. In this case $$COND assumes values 1, 2... and so on, each of which distinguishes a different condition. System variable $$RES, the result code, may also be established when an exception occurs, to give further information about the cause of the exception.

If you need to process the condition number or the result code, handle $$COND and $$RES at the very beginning of the logic introduced by your ON EXCEPTION statement. Normally you should immediately save their values with MOVE statements, or branch on $$COND with a GO TO DEPENDING ON, or code a sequence of IF statements. This is because the majority of Global Cobol statements, summarised in the list below, cause the values in $$COND and $$RES to be destroyed by the time they return control:

ACCEPT (+ variants)	FIRST	READ NEXT
BELL	LAST	RELEASE
CALL	LOAD	REWRITE
CHAIN	LOCK	SCAN
CLEAR	MAPIN	SCROLL
CLOSE	MAPOUT	SEARCH
DELETE	OPEN	SORT
DISPLAY (+ variants)	PERFORM	SUSPEND
EDIT	PRIOR	UNLOCK
EXEC	READ	WRITE
EXIT	RETURN	WRITE NEXT

Note, in particular, that the statement:

```
     DISPLAY $$COND
```

which you might be tempted to write for debugging purposes, should not in fact be used. If you code it by mistake as part of your exception handling logic it will not have the desired effect since $$COND will be reset before it comes to be displayed and all that will appear at the console will be zero. The correct technique is indicated by the first example below. The second shows the coding required if you simply wish to ignore an exception condition:

```
     CALL EXRTN
     ON EXCEPTION                       * IF EXCEPTION, AND
     AND TESTFL POSITIVE                * IF TEST FLAG ON,
          MOVE $$COND TO Z-WORK         * COMPUTATIONAL WORK FIELD
          DISPLAY "EXCEPTION CODE "          * DISPLAYED,
          DISPLAY Z-WORK SAMELINE       * NOT $$COND
     ELSE
          DISPLAY "NORMAL COMPLETION"
     END

     CLOSE PAYFILE                      * IGNORE IRRECOVERABLE
     ON EXCEPTION                       * I/O ERROR ON CLOSE
     END
```

4.6.6 The ON NO OVERFLOW and ON NO EXCEPTION Statements

These statements check for the reverse condition to those checked by the ON OVERFLOW and ON EXCEPTION statements. They should be used in preference to the construct:

```
     ON OVERFLOW
     ELSE
          statements to be executed when no overflow has occurred
     END
```

4.6.7 The ACCEPT...NULL Statement

The ACCEPT...NULL statement establishes a condition which is true only when the operator has keyed the special null string (i.e. <CR>) rather than a normal numeric or character input, in response to a prompt. The ACCEPT statement and its variants are described in more detail in the Global Screen Presentation Manual. The following example shows a format 1 conditional introduced by ACCEPT...NULL:

```
     ACCEPT NAME NULL
     OR COUNT = 20                      * INPUT ENDS WITH 20'th
          GO TO ENDINP                  * ENTRY OR NULL KEYED
     END
```

4.6.8 Statements for Console Input/Output

Global Cobol provides 7 statements to support elementary console handling at the field level. The DISPLAY and DISPLAY...LINE statements, the display operations, output a single field no more than 80 characters in length. ACCEPT and ACCEPT...LINE, the accept operations, input a field of up to 80 characters. The BELL statement sounds the console bell, clears the type-ahead buffer and terminates job management if it is in control: the statement is used to signal errors. CLEAR and SCROLL, respectively, erase the contents of a display screen and re-establish teletype-compatible working. These statements are described in full in chapter 2 of the Global Cobol Screen Presentation Manual.

4.7 Table Handling

A Global Cobol table consists of a number, n, of fixed length entries occupying contiguous storage. Each entry is identified by its index, a number between 1 and n. The first entry has index 1, the second index 2 and so on. Tables are defined by repeating groups or elementary items with OCCURS clauses.

The table handling operations cause a rapid examination of the table to take place, a selected field from each entry being compared with a key, whose length and value you specify. When comparison takes place the key and the current entry are treated like character variables and compared byte by byte from left to right.

After the table handling operation completes you will either be returned the index of the entry which satisfied your request or, if the request was not satisfied, a table operation exception will take place.

4.7.1 The SEARCH Statement

The SEARCH statement is used to identify the first entry of a table **equal** in value to a supplied key. It is coded:

```
     SEARCH tc table key [entry-length]
```

Here tc is the name of a table control area of the following format:

```
     01   TC
       03 TCKEYL    PIC 9(2) COMP       * SUPPLIED KEY LENGTH
       03 TCTERM    PIC X                    * SUPPLIED TERMINATOR
       03 TCINDEX   PIC 9(4) COMP       * RETURNED INDEX
```

You must set up the key length and terminator yourself, but will be returned in the index field.

The table parameter is a variable identifying the location at which the search is to begin, and key is a literal or variable containing the supplied key.

The fourth parameter, the entry length, must be supplied if you are searching a repeating group, when the key length and entry length will be different. It must be an integer literal, a PIC 9(4) COMP variable, or the special system variable $$ENTL.

System variable $$ENTL can only be coded as the fourth parameter of a SEARCH (or SCAN) statement, and its value is the length of an entry in the table supplied as the second parameter. It should be used in preference to specifying the length as an integer literal, so it will remain correct even if the definition of the table is changed. If the parameter is omitted the entry length is assumed to be equal to the key length, as will normally be the case if you search a table of elementary items.

The key field is assumed to be located at the start of each table entry, and the search operation proceeds as follows:

A. The first table entry is selected;

B. If its first byte contains (TCTERM), the terminator value, exception condition 1 is signalled and processing terminates;

C. If the key field, the (TCKEYL) bytes at the start of the entry, is equal to the key supplied as the third parameter of the SEARCH statement, processing terminates normally;

D. Otherwise the next entry is selected, using the fourth parameter or (TCKEYL) if it is omitted, and processing continues as at (b).

When the table operation terminates the search returns the index of the entry it last processed in the TCINDEX field. This will either identify the first entry satisfying your request or, if an exception took place, it will identify a dummy entry starting with the terminator value. You must ensure that such an entry is placed immediately following the table if it is possible to search for a key which is not present, otherwise the search operation will not be properly delimited, and the search will continue examining the memory following the table, with unpredictable results.

If there is a possibility that the key value is not present in the table the SEARCH statement should be immediately followed by an ON EXCEPTION statement to process this condition. If the ON EXCEPTION statement is not coded and the key is not present the program will be terminated in error.

4.7.2 The SCAN Statement

The SCAN statement is used to identify the first entry of a table whose value is **equal to, or greater** than a supplied key. It is coded:

```
     SCAN tc table key [entry-length]
```

where tc, table, key and the optional entry-length parameter are as defined for the SEARCH statement.

SCAN functions identically to SEARCH, except that the criterion for terminating the search normally is that the key field, the (TCKEYL) bytes at the start of each entry, should be equal to or greater than the key supplied by the third parameter. If the fields involved are character data items then the ASCII collating sequence determines the result of the comparison. If the fields are both non-negative computational items of the same precision the result will be determined by the numeric value as you would expect. However, table scans involving negative or display numeric keys, or computational keys of different precision, should be avoided since the outcome is difficult to predict.

If it is possible that no key field in the table will satisfy the scan, you must delimit the table with a dummy entry starting with a byte containing the terminator value, and you should follow the SCAN statements involved with ON EXCEPTION statements to trap possible table operation exceptions.

```
PROGRAM EXAMP
DATA DIVISION
.
.
.
* TABLE CONTROL AREA
*
01    TC
  03  FILLER      PIC 9(2) COMP
                  VALUE 8
  03  FILLER      PIC X
                  VALUE LOW-VALUES
  03  INDEX       PIC 9(4) COMP
*
* KEY OBTAINED FROM OPERATOR
*
77    KEY         PIC X(8)
*
* TABLE WITH 16 ENTRIES PLUS END MARKER
*
```

```
01    TABLE
  03  ENTRY OCCURS 16
   05 KSTORE      PIC X(8)
   05 COUNT       PIC S9(2) COMP
  03  ENDMK       PIC 9 COMP
*
PROCEDURE DIVISION
.
.
.
      DISPLAY "PLEASE SUPPLY A KEY"
      ACCEPT KEY
      SEARCH TC TABLE KEY $$ENTL
      ON EXCEPTION
            IF INDEX GREATER 16
                  DISPLAY "TABLE EXHAUSTED"
                  STOP RUN
            ELSE
                  MOVE KEY TO KSTORE(INDEX)
            END
      END
      ADD 1 TO COUNT(INDEX)
      DISPLAY "KEY STORED IN ENTRY "
      DISPLAY INDEX SAMELINE
      IF COUNT(INDEX)>1
            DISPLAY COUNT(INDEX)
            DISPLAY " TIMES"
      END
.
.
.
```

Figure 4.7 - Table maintenance example

4.7.3 Example

Figure 4.7 shows a fragment of a program used to maintain a table, TABLE, of sixteen 9-byte entries, ENTRY. Each entry contains space to save an 8-byte key field, KSTORE, together with a 1-byte count field, COUNT. The entries are automatically initialised to binary zeros since they form a repeating group following the first VALUE clause of the program. A dummy entry at the end of the table consists of the single byte ENDMK, which is also automatically initialised to binary zeros.

The table control area to be used in the subsequent search operation is set up to specify a key length of 8 bytes and a terminator value which is binary zeros (to match ENDMK).

The procedure division begins by outputting the prompt:

```
     PLEASE SUPPLY A KEY:
```

The operator replies with a key of between one and eight characters and this is placed in the KEY field. Then the SEARCH statement is executed. If the key value is contained in the table the message:

```
     KEY STORED IN ENTRY nnnn mm TIMES
```

is output, where nnnn is the entry number, and mm is the value of the COUNT field, which will be greater than 1 since it is always incremented whenever an entry is selected. The program then continues with other processing not shown in the figure.

If the key value is not already in the table, by checking the value of the index returned the program sees whether any free entries remain. If not the message:

```
     TABLE EXHAUSTED
```

is output and the program stops.

If the index is between 1 and 16 it identifies a free entry to which the key is moved. (Since the key consists of ASCII characters entered at the console it cannot start with a binary zero byte so this action automatically extends the table.) The count is incremented for the very first time and the message:

```
     KEY STORED IN ENTRY nnnn
```

is then output and the program continues.

4.7.4 Programming Notes

Although the key field should normally be the first field of a table entry, you may use a field from the middle of the entry providing you pass the first occurrence of it as the table parameter of the SEARCH or SCAN statement, and construct the appropriate dummy entry at the end of the table. For example, if you required to find the first entry in the table of Figure 4.7 possessing a COUNT field greater than 5, you would need to specify a second table control area:

```
      01    TC2
        03  FILLER      PIC 9 COMP              * KEY LENGTH 1
                        VALUE 1
        03  FILLER      PIC 9 COMP              * TERMINATOR 0
                        VALUE 0
        03  INDEX2      PIC 9(4) COMP           * INDEX
```

You would also need to extend the ENDMK field so it overlaid the 17th dummy occurrence of the COUNT field, by replacing its definition by:

```
        03  ENDMK       PIC X(9)
                        VALUE LOW-VALUES
```

for example. The required table operation involves code of the form:

```
      SCAN TC2 COUNT(1) SIX $$ENTL
      ON EXCEPTION
            processing when there is no entry with a COUNT field greater than 5
      END
```

The quantity SIX is defined in working storage as a PIC 9 COMP field whose value is 6. It would have been wrong to code an integer literal instead. For example:

```
      SCAN TC2 COUNT(1) 6 $$ENTL
```

because this would cause a two-byte PIC 9(4) COMP field to be set up as the key. The scan would then use only the senior byte of this field in comparisons (because the key length has been specified as 1 byte) and this byte, of course, contains 0, not 6. The conclusion is that it is wise to avoid using literal keys in table operations involving computational quantities. It is better to employ a variable, which you initialise explicitly so that you can ensure that the correct byte length is set up. Table 3.3.3 can be used to determine how many bytes are established by any particular computational picture clause.

Note that it is not necessary for the key length always to be less than the entry length. You can, for example, search a string of characters for the occurrence of a particular word by treating the string as a table with entry length 1 and defining the key to have the length and value of the word you are searching for. For example, to scan STRING, an ASCII character string terminated by a null, binary zero, byte, to see if it contains the characters "Global", define the table control area as:

```
01    TC3
  03  FILLER      PIC 9(2) COMP          * KEY LENGTH 3
                  VALUE 6
  03  FILLER      PIC 9 COMP             * TERMINATOR 0
                  VALUE 0
  03  INDEX3      PIC 9(4) COMP          * INDEX
```

The required table processing statements are:

```
SEARCH TC3 STRING "Global" 1
ON EXCEPTION GO TO NOTFOUND
processing when the string contains "Global"
```

You may search a string **backwards** by specifying a **negative** entry length as the fourth parameter supplied to a SEARCH or SCAN statement. You **must** supply a fourth parameter to perform backwards table processing, since you are not allowed to specify a negative value in the first byte of the table control area. Suppose you wish to display the last non-blank character in the 80-byte table RECORD, which you know always contains graphics characters. ASCII blank is the lowest value graphic, with a decimal value of 32, so you need only scan RECORD backwards to find the first character greater than or equal to 33. Lay out working storage as follows:

```
01    FILLER
  03  FILLER      PIC X                  * TERMINATOR OF
                  VALUE LOW-VALUES       * BACKWARD SCAN
  03  RECORD OCCURS 80 PIC X
*
01    TC4
  03  FILLER      PIC 9(2) COMP
                  VALUE 1                * KEY LENGTH
  03  FILLER      PIC X
                  VALUE LOW-VALUES       * TERMINATOR VALUE
  03  INDEX4      PIC 9(4) COMP
*
77    NBLANK      PIC 9(2) COMP
                  VALUE 33               * CHARACTER > BLANK
```

Then the following code displays either a message identifying the character in question, or one indicating that the record consists only of blanks:

```
SCAN TC4 RECORD(80) NBLANK -1
ON EXCEPTION
      DISPLAY "RECORD CONSISTS ONLY OF BLANKS"
ELSE
      SUBTRACT INDEX4 FROM 81 GIVING INDEX4
      DISPLAY "LAST NON BLANK CHARACTER IS "
      DISPLAY RECORD (INDEX4) SAMELINE
END
```

Note how, even when the table is searched backwards, the entries are identified counting from 1. This is why it is necessary to subtract the result returned by SCAN from 81 in this example in order to generate the correct value with which to index RECORD.

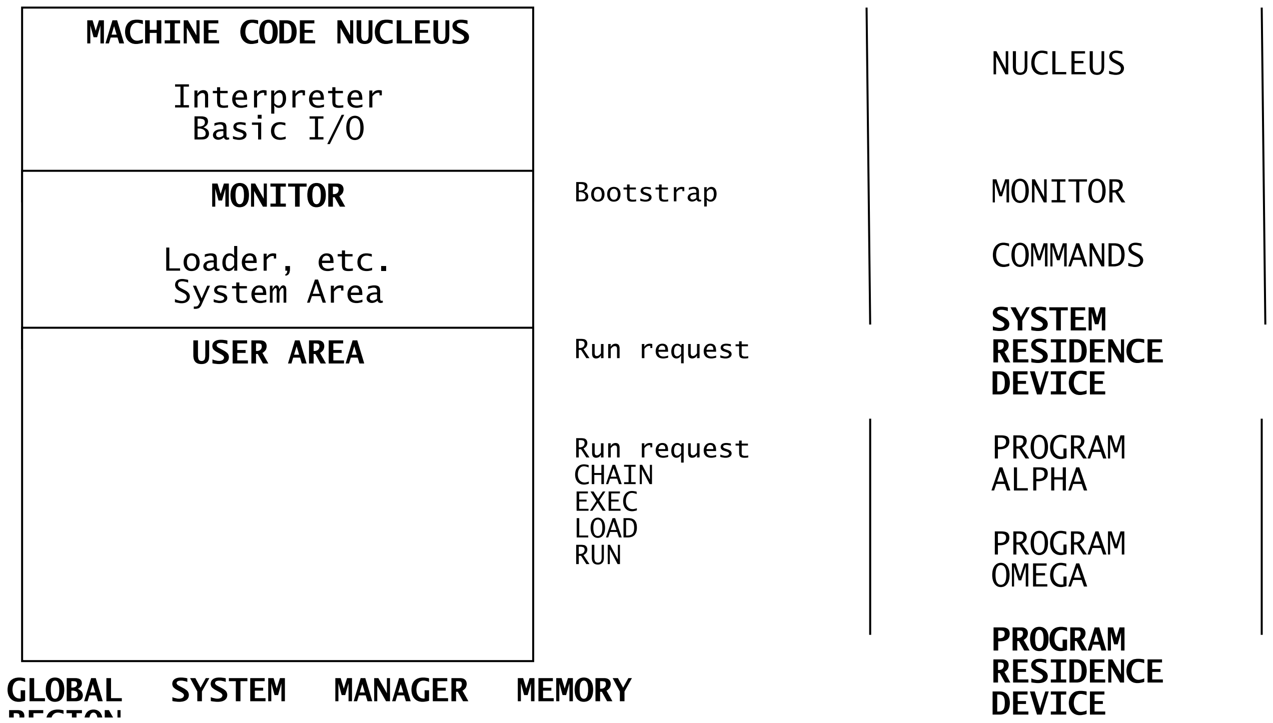

Figure 4.8 - The Global System Manager run-ti   t

4.8 Program Management

4.8.1 The Global System Manager Memory Region

All variants of Global System Manager execute programs from a contiguous area of random access storage known as the Global System Manager memory region (Figure 4.8). This region is itself subdivided into three parts.

The beginning of the region is occupied by the permanently resident components of Global System Manager: a **nucleus** of machine code which interfaces Global System Manager with a particular target configuration, and a small **monitor** written in Global Cobol itself. The machine code nucleus references the Global Cobol interpreter, routines for basic input/output, and assembler language services such as table search and scan.

The monitor includes functions such as the loader, command handler, open/close routines, console handler and I/O error retry routines. The monitor contains a data area called the **System Area**, which serves as a communication region for the Global System Manager software, the interpreter, and user programs.

The remainder of the Global memory region is the **user area** in which application programs and monitor command programs execute. Irrespective of the actual memory address of the user area it begins at Global Cobol location zero, the interpreter being responsible for the mapping of Global Cobol locations to machine addresses.

4.8.2 Program Preparation

User programs are held as files on a volume which resides on the program residence device. Each file on this volume is identified by a unique 8 character name which, for a program file, is termed the **program-id**. Alternatively, up to 100 programs can be held in a single library file, but even so each program is still identified by its unique program-id.

The linkage editor is used to create program files by combining one or more compilations together with access methods and system routines from the system library. When the linkage edit takes place you can specify:

- the user compilation(s) that are to be combined;
- the program-id to be used for the resulting file;
- the storage area the program is to occupy.

The first procedure division statement of the first compilation submitted to the linkage editor becomes the program's entry point.

Note that the program-id bears no relation to the program name which appears in the PROGRAM statement introducing each compilation. The program name identifies an individual compilation; the program-id identifies the program file that results from linking one or more compilations.

The Global Cobol User Manual describes program preparation in more detail.

4.8.3 Command Programs

The greater part of Global System Manager consists of command programs which execute in the user area just like user application programs. In this way the functions of a comprehensive Global System Manager are provided without a consequent increase in main storage requirements.

Command programs reside on the system residence device in the command library. The first character of the program-id of a command program is always the $ character. This serves to distinguish command programs from user programs: an application programmer must **never** create a program whose program-id contains the $ character.

Global System Manager commands are described in the Global System Manager and Global Utilities manuals.

4.8.4 Running a Program

To start a session the operator initiates Global System Manager using the start-up procedure described in the Global Operating Manual. (This will usually be little more complicated than loading a diskette and pressing a button.) Once you have signed on, Global System Manager responds with the ready prompt or main menu.

The operator can either supply the name of a command program (on the system residence device), the name of an application program (on the program residence device) or type the appropriate option number in the menu, if one is displayed. For example:

- Reply $COBOL to invoke the Global Cobol compiler;
- Reply SALES to run your sales ledger program;
- Type 1 to select option one from the menu, if a menu is displayed.

In either case a program is loaded into the user area and control passed to its entry point. (The characters keyed are padded with rightmost blanks, if necessary, to make up the eight necessary for the program-id.)

If the program terminates in error, executes a STOP RUN statement, or EXITs from the highest level of control, the resident monitor is re-entered. The monitor will display diagnostic information, if it is required, and allow the debugging facility to be invoked, should this prove necessary. Eventually, however, the ready or main menu prompt will be redisplayed to allow the operator to execute a command or run another application program.

4.8.5 The CHAIN and RUN Statements

Control can be passed from one program to another without operator intervention by means of the CHAIN or RUN statement. RUN will in addition:

- close any files that are currently open without completing any outstanding write operations;
- release any regions locked by the LOCK statement;
- re-establish teletype mode;
- set system variable $$RUN to 1;
- set system variable $$ESC to 0;
- reset the library index pointer, $$INDE, to its initial value addressing the top 1134 bytes of the user area;
- release any free memory acquired by the FREE$ routine;
- unload any temporary user stack items created by the LOAD$ system routine.

Normally you use CHAIN to pass control between programs of the same application, and RUN to invoke the first program of a separate system. You code:

```
     CHAIN A
or:
     RUN A
```

where A is a character variable or literal containing the program-id of the program to which you wish to pass control. If A is more than 8 bytes in length only the first 8 will be used for the program-id. If A is less than 8 bytes long it will be extended with rightmost blanks to make up the program-id.

Note that the RUN statement will cause a program check if used on a pre-5.1 system.

The effect of the statement is to load and pass control to the program it identifies. However, if any error arises during the loading process (because the program is not present on the program residence device, for example) the program that attempted the statement will simply be terminated in error.

A program entered via a CHAIN or RUN statement can itself issue a CHAIN or RUN statement. An EXIT statement issued at its highest level of control, a STOP RUN, or an error will cause control to be returned to the monitor as described in 4.8.4.

Chained programs are normally linkage edited to start at a common location and therefore overlay each other. However, you can pass up to 16 bytes of parameters between the programs of a chain by using $$AREA for this purpose. Alternatively, a larger parameter area may be provided by defining the area, which can contain any number of **uninitialised** data declarations, at the very beginning of the working storage of the very first compilation of each program involved.

The safest technique is to use the COPY statement to include a common copy book in each of the initial compilations. The parameter area thus defined will begin at the common starting location of each program. The uninitialised area should always be of even length; if not the last byte will be initialised to binary zero.

Within the parameter area it is vital that **none** of the data definitions contains a VALUE clause. Providing this is the case the storage occupied by the parameter area remains unchanged when the CHAIN statement is executed. This is because the bytes preceding the first VALUE clause, map or file definition in a compilation are not initialized (as explained in 3.4.5). The parameter area can therefore be used to pass a data structure of any complexity between chained programs.

4.8.6 The EXEC Statement

The EXEC statement provides the capability for conventional overlay handling. You code:

```
EXEC A [USING B C ...]
```

where A is the character variable or literal used to construct the program-id of the program to which you wish to pass control. The entry point of that program must be an ENTRY statement whose USING clause (if any) matches the USING clause (if any) of the EXEC statement. The operands B, C.... within an EXEC statement's USING clause may assume any of the formats valid in a CALL statement's USING clause, as described in 4.5.4.

The EXEC statement is very similar to CALL. The difference is that the entry point address is determined at run-time, rather than by the linkage editor. If any error arises during the loading process (because the requested program is not present on the program residence device, for example) the program that attempted the EXEC statement will be terminated in error.

An EXIT statement issued from the highest control level of a program entered via EXEC returns control to the instruction following the EXEC statement. A STOP RUN statement, on the other hand, bypasses the calling program and returns control to the monitor directly, as described in 4.8.4.

When you linkage edit an overlay program to be entered via EXEC you are responsible for making sure that the program does not overlay any part of the calling program that will be needed later. For this reason chaining is simpler to use than EXEC, and is to be preferred except when the called program needs to make use of service routines within the caller.

4.8.7 The LOAD Statement

The LOAD statement is coded:

```
     LOAD A
```

where A is a character literal or variable to be used in constructing the program-id, as described above.

The effect of the LOAD, if successful, is to fetch the program into the user area **without** passing control to it. A pointer to the program's entry point is remembered for you in the system variable $$EPT so that it can subsequently be entered by a pointer-based CALL. Indeed, the sequence:

```
     LOAD A
     CALL $$EPT USING B C ...
```

is identical in effect to:

```
     EXEC A USING B C ...
```

LOAD differs from EXEC in the processing that takes place if it is **unsuccessful.** Whereas EXEC terminates the calling user program in error, LOAD generates a load exception which can be tested for by an ON EXCEPTION statement immediately following the LOAD. If this statement is not coded and a load error occurs the program is terminated in error just as if EXEC had been used.

When a load exception takes place $$EPT remains unchanged but the system variable $$COND is set to indicate the reason for the load error:

- $$COND = 1 means that an irrecoverable I/O error occurred when attempting to load the program;

- $$COND = 2 means that the program was not present on the program residence device;

- $$COND = 3 means that the program file was too large to fit in the available user area.

4.8.8 System Variables Used by the Loader

Whenever program loading takes place, either because the operator supplied the name of the program to run in response to the ready prompt, or because a CHAIN, EXEC, LOAD or RUN statement was executed, the system variable $$PGM is set to the program-id involved. The system variable $$RUN is set to 1 if the load operation was in response to an operator request or RUN statement and to 0 if it was due to a CHAIN, LOAD or EXEC.

$$RUN can be tested to prevent programs which should only be executed by controlling programs from being run by mistake.

As explained in the File Management manual each direct access value used by Global System Manager is identified by a unique volume-id, a name of up to 6 characters. You may set the PIC X(6) system variable $$PVOL to the volume-id of the disk or diskette containing your program immediately before executing the CHAIN, RUN, LOAD or EXEC statement responsible for loading it to cause Global to check that the identified volume is online and prompt the operator to mount it if necessary. $$PVOL is then reset to its normal state, LOW-VALUES, so that no further checking takes place until you set up another volume-id. Thus, for example, to ensure that volume PRGRES is online before attempting to chain to program SALES:

```
MOVE "PRGRES" TO $$PVOL
CHAIN "SALES"
```

The volume-id checking causes an extra access to the program volume, so it should be requested only if necessary.

Following any successful load operation $$EPT is set to point at the entry point of the newly loaded program. However, if the operation fails $$EPT remains undisturbed. If the statement affected is a LOAD $$COND is set to 1, 2 or 3 to indicate the reason for the failure, as described in 4.8.7.

4.8.9 The Loader's Use of High Memory

When a load operation involves a program library the top 1134 bytes of the user area are temporarily acquired for the library index record, a table containing the program-id and file address of each member of the library. The high area is only used for part of the operation and can, indeed, be overwritten by the incoming program.

Normally the temporary overwriting of the top 1134 bytes has no effect on chained programs, or conventionally developed overlay systems in which the deeper levels of overlay occupy the higher memory locations. However, some special programs may require to load overlays, yet still retain information at the top of the user area, for their own purposes. Global Cobol therefore provides a PIC PTR system variable named $$INDE which you can set to address an alternative area which the loader can use for the index record when high memory is already occupied. Use of $$INDE is explained in detail in the Global Development System Subroutines Manual.

4.9 The EDIT Statement

The EDIT statement allows a numeric value to be edited into special formats. It is coded:

```
     EDIT A INTO B FORMAT C
```

where A is a computational or display numeric variable, or a computational literal. It can have a maximum of 12 digits preceding the decimal point, and a maximum of 6 following the decimal point. B must be a character variable with a maximum length of 30 characters. C may be a character variable or literal.

The value given by A is edited according to the format specified by C and the result is placed, right justified, in B. The number of decimal places specified in the picture clause of A (or the number of digits following the decimal point if a literal) determines the number of decimal places that will appear in the result.

4.9.1 The Edit Format

The edit format is a character variable or literal of 1 to 4 characters specifying the editing to be performed. If it is shorter than 4 characters the remaining positions are treated as spaces. The format consists of 3 separate fields:

- character 1: the float or fill character;
- character 2: the comma insertion indicator;
- characters 3 & 4: the sign editing parameters.

Fill characters cannot be used with negative numbers, and the 0 fill character should not be used in combination with comma insertion. The use of each of these parameters is given in Table 4.9.1, and examples of their use are shown below:

VALUE (A)	**FORMAT (C)**	**RESULT (B)** **xxxxxxxxxxx**
1234.56	"$"	$1234.56
1234.56	"$C"	$1,234.56
1234.56	"*"	****1234.56
1234.56	"*C"	***1,234.56
1234	"0"	00000001234
1234.56	"NC"	1,234.56
123456	"NC"	123,456
1234567	"NC"	1,234,567
1234.56	"ND"	1.234,56
1234567	"ND"	1.234.567
1234.56	"BC"	1,234.56
0.00	"BC"	
-1234.56	"NC"	-1,234.56
-1234.56	"$C"	-$1,234.56
-1234.56	"NC-"	1,234.56-
1234.56	"NC-"	1,234.56
-1234.56	"NC()"	(1,234.56)
-1234.56	"$C()"	($1,234.56)
-1234.56	"NCDB"	1,234.56DB

1234.56	"NCDB"	1,234.56
-1234.56	"NCCR"	1,234.56CR
-1234.56	"NDCR"	1.234,56CR

FORMAT CHARACTERS	VALUE	MEANING
1	"*"	Replace leading spaces by "*"s (number must be +ve).
	"0"	Replace leading spaces by "0"s (number must be +ve).
	" " or "$" or "x"	Prefix number with a " " "$", or "x" symbol.
	"B"	Blank out number if zero.
	"N"	None of the above.
2	"C"	Insert a comma between each group of 3 digits to the left of the decimal point, which appears as a period (full stop).
	"D"	Insert a period between each group of 3 digits to the left of the decimal point, which appears as a comma.
	"N" or " "	Do not insert commas or dots.
3 & 4	" "	Prefix number with a "-" if negative.
	"()"	Enclose number in parentheses if negative.
	"- "	Suffix number with "-" if negative.
	"x "	Suffix number with "x" if negative.
	"CR" or "DB" or "DR" or "xx"	Suffix number with 2 characters specified if negative.

x represents any single character other than special characters, described separately.

Table 4.9.1 - Edit Formats

4.9.2 Overflow

Overflow will take place if operand A is greater than or equal to 10^{13}. Overflow will also take place if B is not long enough to hold the result of the edit operation. Table 4.9.2 gives the minimum number of characters required for the various edit formats and values to be edited.

Overflow will also occur if an attempt is made to edit a negative value using a format which specifies a 0 or * fill character.

SIGN EDITING	Leading – sign and value always	leading – or single character	( ) or double trailing

	positive				trailing sign				sign, e.g. CR, DR, DB			
COMMA INSERTION?	Y		N		Y		N		Y		N	
CURRENCY SYMBOL?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
NUMBER OF DIGITS (ignoring leading zeros)												
1	2	1	2	1	3	2	3	2	4	3	4	3
2	3	2	3	2	4	3	4	3	5	4	5	4
3	4	3	4	3	5	4	5	4	6	5	6	5
4	6	5	5	4	7	6	6	5	8	7	7	6
5	7	6	6	5	8	7	7	6	9	8	8	7
6	8	7	7	6	9	8	8	7	10	9	9	8
7	10	9	8	7	11	10	9	8	12	11	10	9
8	11	10	9	8	12	11	10	9	13	12	11	10
9	12	11	10	9	13	12	11	10	14	13	12	11
10	14	13	11	10	15	14	12	11	16	15	13	12
11	15	14	12	11	16	15	13	12	17	16	14	13
12	16	15	13	12	17	16	14	13	18	17	15	14

NOTE: For decimal numbers use the number of digits before the decimal point, then add (1 + number of decimal places) to the size given.

Table 4.9.2 - Sizes of Edited Integer Values

4.10 Statements for Networking and Multi-user Working

All variants of Global System Manager (except for a single user system) are capable of running a number of co-operating programs at the same time. The concurrent version of Global System Manager allows up to 9 programs to time-share the processor on a single screen computer, and each of them will continue running and writing to the screen even when its partition is not actually being displayed. The multi-user version of Global System Manager provides a conventional multi-user environment in which several programs, each of them running on a separate screen or partition, time-share a single processor. Global System Manager running on a network allows a number of single and multi-user computers to be linked together in a local area network in order to share programs and data files.

Programs which time-share a processor are suspended at the end of a time slice, or when waiting for a slow I/O operation to complete, for example a print or accept. There is also a SUSPEND statement which allows a program to explicitly request its own suspension for a period of time.

Programs will often require to share data files. In order to ensure consistency during the updating of shared files two special statements, LOCK and UNLOCK, are provided. These statements are described in the Global File Management Manual and Global Data Management Manual.

Programs containing LOCK, UNLOCK and SUSPEND statements will run unchanged under the single-user version of Global System Manager.

4.10.1 The SUSPEND Statement

SUSPEND causes the program to be suspended for a period of time. It is coded:

```
SUSPEND [seconds]
```

The optional seconds parameter is the name of a PIC 9(4) COMP variable or integer literal containing the number of seconds for which the program is to be suspended. If the parameter is omitted, or the value supplied is less than 1, then the program is simply suspended for a brief period, as if it had reached the end of its time-slice, allowing other users of the processor to execute.

If characters are keyed at the console when a program is suspended, then the suspend will be cancelled so that the program can process the input if necessary. (The CHECK$ routine described in the Global Development System Subroutines Manual is used to determine whether unsolicited console input, not requested by a prompt, is present.)

A SUSPEND seconds statement executed in a single-user environment will operate as described above, providing that the system supports a timer. If it does not, the SUSPEND request will simply be ignored.

SUSPEND is always effective under other variants of Global System Manager, since timer support is a prerequisite for multi-user versions of Global System Manager.

4.10.2 Programming Notes

The SUSPEND statement should be used when you know your program cannot proceed until some other user completes an activity. Since the only way of co-operating jobs communicating is by means of shared files, a typical use of SUSPEND by, say, a special purpose spooling program, might be as follows:

- read the shared communications file to see if a report requires printing;
- if no report is available, execute a SUSPEND for 60 seconds, then repeat the first step;
- otherwise, print the report then update the communications file to indicate it has been printed (normally the file will have to be locked, to prevent simultaneous updates by two users);
- repeat this process.

The $SP command employs a similar technique using:

```
SUSPEND scan-rate
```

which causes the program to wait for a user-definable number of seconds when there are no files available to print. Little time is wasted in practice since, in a highly active system, $SP will seldom manage to clear the backlog of files awaiting its attention, and the SUSPEND statement which causes it to wait will only be executed infrequently.

Note that it is **not** necessary to issue SUSPEND statements at frequent intervals during batch processes to allow interactive programs the chance to operate because the time slicing mechanism automatically ensures that no processor-bound job can monopolise the system.

4.11 File Processing

The Global Cobol File Management Manual contains full information regarding Relative Sequential Files, Indexed Sequential Files, Variable Length Record Files, Text Files and Basic Direct Files, together with Physical Sector Access, Native File access, Data Security, advanced file handling and the Multi-key Sort command.

The Global Cobol Data Management Manual contains all the relevant information concerning the Global Data Management System (DMAM).

5. Using Pointers, Based Areas and Global Symbols

5.1 Procedure Division Statements for Pointer Handling

5.1.1 The MOVE Statement

A MOVE statement of the form:

```
     MOVE P TO Q [R S ...]
```

can be used to transfer the value within a pointer variable P to one (or more) pointer variables Q (R S ...). A maximum of 7 variables may follow the word TO. If there is more than one, the MOVE statement is said to be multi-target and is equivalent to a number of simple MOVEs. For example:

```
     MOVE W TO X Y Z
```

has exactly the same effect as:

```
     MOVE W TO X
     MOVE W TO Y
     MOVE W TO Z
```

The pointer variables appearing in the MOVE statement may be either simple or indexed.

5.1.2 The POINT Statement

The POINT statement allows you to set a pointer to address a particular location dynamically. You code:

```
     POINT P AT A
```

where P is a simple or indexed pointer variable, and A is a paragraph name, section name, entry name, variable name, literal, filename or mapname.

If A is a paragraph or section name, P is set to address the first executable instruction of the paragraph or section, which must appear in the current compilation.

If A is an entry name, P is set to address the first executable instruction at the entry point. The entry name must appear in an ENTRY statement or CALL statement in the current compilation.

If A is a variable name or a literal, P is set to address the first byte of the area occupied by that variable or literal. If A is a filename or mapname, P will address the first byte of the corresponding file or map definition. A variable name, filename or mapname may appear in either working storage or the linkage section, and a variable may be indexed.

5.1.3 Transfer of Control Statements

You may use a pointer as the first (or only) operand of a GO TO, PERFORM or CALL statement. For example:

```
     GO TO P
     PERFORM P
     CALL P [USING B C ...]
```

In the case of GO TO and PERFORM, the pointer should address the paragraph or section to which control is to be passed. For a CALL the pointer should address an entry statement whose USING clause, if any, matches that of the CALL statement.

5.1.4 IF and DO Statements

Pointers can be compared using any of the following conditions in an IF or DO statement, as described in section 4.6 (P and Q are both simple or indexed pointer variables):

```
     P EQUAL Q           or        P = Q
     P NOT EQUAL Q       or        P NOT = Q
     P LESS Q            or        P < Q
     P NOT LESS Q        or        P NOT < Q
     P GREATER Q         or        P > Q
     P NOT GREATER Q          or        P NOT > Q
```

The comparison takes place treating the pointer's senior bit as a 32K unit bit rather than a sign bit, so that the result of the compare is as you would expect. For example, if P contains the value 33,502 and Q the value 28,000 then the statement:

```
     IF P GREATER Q GO TO AA090
```

will cause control to pass to AA090.

5.1.5 Intermediate Code Statements using Pointers

Two intermediate code statements, $PUSH and $POP, also manipulate pointers. Their main use is in selecting a particular entry from a table, when the length of each entry is only known at run-time. $PUSH and $POP are explained in section 6.7.

5.2 Pointer Arithmetic

There are no special procedure division instructions to help you to perform arithmetic on pointers. This is somewhat complex because a pointer cannot be treated as a straightforward computational item since its senior bit is interpreted as a 32K bit rather than a sign bit. This makes arithmetic operations rather tortuous because it is necessary to use three-byte computational quantities whose junior two bytes hold the actual pointer values. This can be accomplished by the careful use of redefinitions.

The examples which follow show the most straightforward, rather than the most efficient, way of performing the common operations involving two pointers, P and Q. It is assumed that the following data declarations have been made in working storage:

```
     77    QARITH      PIC S9(6) COMP
     77    PARITH      PIC S9(6) COMP
     01    FILLER REDEFINES PARITH
```

```
        03  FILLER      PIC X
        03  POINTER     PIC PTR
```

5.2.1 Adding an Offset to a Pointer

This example increments P by the computational value OFFSET giving the result in Q, which now addresses an area (OFFSET) bytes from that addressed by P:

```
      MOVE 0 TO PARITH
      MOVE P TO POINTER
      ADD OFFSET TO PARITH
      MOVE POINTER TO Q
```

5.2.2 Determining the Distance between Two Pointer Locations

In this example we calculate the number of bytes separating the location addressed by pointer Q from the location addressed by pointer P. The result is placed in the computational variable DISP:

```
      MOVE 0 TO PARITH
      MOVE Q TO POINTER
      MOVE PARITH TO QARITH
      MOVE P TO POINTER
      SUBTRACT PARITH FROM QARITH GIVING DISP
```

5.3 Based Areas

A based area is defined by a level 01 group coded in the linkage section. The level 01 data declaration itself must use the BASED clause and appear as follows:

```
      01 name BASED pointer
```

where pointer is the data name of a simple pointer variable defined in working storage.

By assigning a value to the base pointer you effectively position the fields of the level 01 group to overlay the area that the pointer addresses. Once this is done other Global Cobol statements can be used to manipulate the fields of the group, and thus the area, in the normal way.

5.3.1 Example

Figure 5.3 shows the complete code of a simple subroutine making use of a based area. The routine is invoked by a CALL statement of the form:

```
      CALL UPDATE USING element standard-area
```

where element is a 12-byte area containing an element type and, if the type is "EX", a pointer to a 23-byte extension area. The second parameter, standard-area, identifies the extension to be used if the type is not "EX".

If the element type is "EX" the routine uses a MOVE statement to set EXAREA's base pointer from EPTR to address the current element's extension area. Otherwise a POINT statement is used to set EXAREA's base pointer to address the standard extension area supplied as the second parameter. The extension area flag, EXFLAG, is then tested and if it is set to ASCII space no further processing is required and the routine exits. If the flag is not space the extension area count EXCOUNT is incremented and then the routine exits.

To keep this example simple it has had to be rather artificial. However, it does serve to illustrate the power of Global Cobol based area handling in system programming applications.

```
PROGRAM EXSUB
DATA DIVISION
*
77    P-EXAREA    PIC PTR           * POINTS AT CURRENT EXTENSION
*
LINKAGE SECTION
*
* GROUP DESCRIBING AN ELEMENT SUPPLIED BY CALLING PROGRAM
*
01    ELEMENT
  03  ETYPE       PIC X(2)          * ELEMENT TYPE
  03  ENAME       PIC X(8)          * ELEMENT NAME
  03  EPTR        PIC PTR           * POINTER TO EXTENSION AREA
*
* STANDARD EXTENSION AREA, SUPPLIED BY CALLING PROGRAM
*
01    SEXT
  03  FILLER      PIC X(23)
*
* GROUP DESCRIBING AN EXTENSION AREA
*
01    EXAREA BASED P-EXAREA
  03  EXCOUNT     PIC 9(4) COMP
  03  EXFLAG      PIC X
  03  EXDATA      PIC X(20)
*
PROCEDURE DIVISION
*
ENTRY UPDATE USING ELEMENT SEXT
*
      IF ETYPE = "EX"
            MOVE EPTR TO P-EXAREA   * SELECT EXTENSION IF PRESENT
      ELSE
            POINT P-EXAREA AT SEXT  * ELSE USE STANDARD EXTENSION
      END
      IF EXFLAG SPACE EXIT
      ADD 1 TO EXCOUNT              * INCREMENT
      EXIT
ENDPROG
```

Figure 5.3 - Based area usage example

5.4 The BASE Statement

The BASE statement can be used to position any proper linkage section group to overlay a given area, or to overlay an area addressed by a pointer. A proper linkage section group is either a file or map definition (identified by its filename or mapname), a based area, or a level 01 or 77 item. As its name implies, it must be coded in the linkage section. In addition, to qualify as a proper linkage section group a level 01 or 77 item must not itself be a redefinition of a subordinate data item (level 02 to level 49) or a system variable.

There are two forms of the BASE statement:

```
BASE group ON P
BASE group AT A
```

The BASE...ON form causes the proper linkage section group identified by its first operand to overlay the area addressed by its second operand, which must be a simple or indexed pointer variable.

The BASE...AT form causes the proper linkage section group identified by its first operand to overlay the area starting at the first byte of its second argument. The second argument may be either a simple or indexed variable, a file or map definition (identified by its filename or mapname) or a literal.

5.4.1 Example - Handling a 2-dimensional Array

Figure 5.4.1A is an example showing the use of the BASE statement to process a 2-dimensional array. The technique can readily be extended to multi-dimensional arrays.

The array to be processed is a 10 by 12 matrix of PIC 9(4) COMP elements, each of which is therefore 2 bytes in length. The arrangement of the elements in main storage is shown in Figure 5.4.1B, and the actual storage area involved is reserved by the field MATRIX of the example program of Figure 5.5.1A. The data declaration in fact defines the matrix as 10 rows of 24 bytes, so that MATRIX(I), for example, selects row I.

Within the linkage section a level 01 group ROW, corresponding to a single row of the matrix, has been set up. The subordinate field ROW-EL is defined as an indexed variable, in order that ROW-EL(J) will select the J'th element of any row overlaid by ROW. The logic to zeroise the I,J'th element is therefore simply:

- Base ROW at the I'th row of MATRIX (BASE ROW AT MATRIX(I))
- Zeroise the J'th element of that row (MOVE 0 TO ROW-EL(J)))

```
PROGRAM MAIN
DATA DIVISION
*
* MATRIX DEFINED AS 10 ROWS EACH 24 BYTES (12 ELEMENTS) LONG
*
77       MATRIX OCCURS 10 PIC X(24)
*
77       I     PIC 9(4) COMP
77       J     PIC 9(4) COMP
.
.
.
LINKAGE SECTION
01       ROW
  03     ROW-EL OCCURS 12 PIC 9(4) COMP
.
.
.
PROCEDURE DIVISION
.
.
.
*
* SET I,J'TH ELEMENT OF MATRIX TO ZERO
*
BASE ROW AT MATRIX(I)
MOVE 0 TO ROW-EL(J)
```

Figure 5.4.1A - Example of 2-D array access using base

```
       1         2         3         4         5         6         7
8        9        10        11        12
      (1,1)    (1,2)    (1,3)    (1,4)    (1,5)    (1,6)    (1,7)    (1,8)
(1,9)    (1,10)   (1,11)   (1,12)

       13

      (2,1)
                                               24

                                              (2,12)
   .                                                        .
   .                                                        .

       109

      (10,1)
                                                210

                                               (10,12)

   <-------------------------------------------------------->
                                                               24-bytes
```

Figure 5.4.1B - Memory map of 2-D array

5.4.2 Programming Notes

If B is the name of a based area, with P as its base pointer, then the statement:

```
     BASE B ON Q
```

has as exactly the same effect as:

```
     MOVE Q TO P
```

whilst:

```
     BASE B AT A
```

is the same as:

```
     POINT P AT A
```

The advantage of the BASE statement is that it is not necessary to introduce a special additional pointer in order to use it: it can be employed to position a linkage section group which is not defined as a based area. However, the storage allocated by the compiler is the same, since an internal pointer is set up in this case.

Note, however, that use of the BASE statement is mandatory when it is necessary to position a file or map definition, since neither can be defined as a based area. You must code either:

```
     BASE filename ON P
     BASE mapname ON P
```

or:

```
     BASE filename AT A
     BASE mapname AT A
```

where filename and mapname are, respectively, the first operands of an FD or MD statement coded in the linkage section.

5.5 The GLOBAL Statement

The GLOBAL statement causes its operand to be defined as a global symbol. You code:

```
     GLOBAL symbol
```

It may appear anywhere within the data division. The statement, which is simply a directive to the compiler, occupies no storage. Any number of GLOBAL statements may be coded providing the number of global symbols in use by a compilation does not exceed 127.

The symbol used in a GLOBAL statement can be a data name, filename, mapname, paragraph name or section name, and need not necessarily be defined in the current compilation. However, if the symbol is defined then it must be part of either working storage or the procedure division: during compilation any GLOBAL statement referring to a symbol defined in the linkage section will be flagged with a warning message.

Entry names appearing in CALL and ENTRY statements in the program are automatically made global symbols, and there is therefore no need to code GLOBAL statements for them. (If you do code a GLOBAL statement with such an entry name as its operand the compiler will accept it, but the statement will have no special effect and might as well be omitted.) Names of common and external sections are also automatically made globals.

Each global symbol should be **defined** in just one of the compilations supplied to the linkage editor. If the linkage editor detects a global symbol with two or more conflicting definitions this is treated as a fatal error and no program file is produced.

Although a global symbol may only be defined once, it can of course be **referenced** by as many compilations as require it. To refer to a global symbol which is not defined in the current compilation you must introduce it in a GLOBAL statement and then use the symbol in the VALUE clause of a pointer data item. Alternatively, if the symbol is a paragraph name, section name or entry name you can pass control to the first instruction of the paragraph, section or subroutine by using the global as the first operand of a GO TO, PERFORM or CALL statement.

If the symbol is a data name you must define the area it labels as a based area using the pointer you have established.

If the symbol is a filename or mapname you must code an appropriate FD or MD in the linkage section. Then, before using this FD or MD, you must execute a BASE statement for it, to position it to overlay the area addressed by the pointer.

In practice, the use of common and external sections provides easier and more efficient access to externally defined data than the GLOBAL statement, and should be used in preference. One remaining use of a GLOBAL statement is to provide a dummy reference to the entry point of a routine in a root segment, to force that routine to be included from a library during a linkage-edit. Thus, for example, if you want to include the PASS$ routine in the root program, you simply code:

```
     GLOBAL PASS$
```

at the start of the root program, and it will be included even though it is not referenced by any CALL statement.

5.5.1 Programming Note - Undefined Global Symbols

A global symbol is said to be undefined when it does not appear within the working storage or procedure division of any compilation processed by the linkage editor during the construction of the program file. Any pointer with a value clause referring to an undefined global will be initialised to #0000.

When the linkage editor detects undefined globals it reports the number found and the actual missing symbols both on screen and in the map listing. A program file is produced since in some cases (typically during testing) the globals may intentionally be undefined.

6. Intermediate Code Support

6.1 Introduction

To achieve portability Global Cobol statements are compiled into an intermediate code which is interpreted at run-time. Intermediate code is a simple, one-address, order code. The statements described in this section and summarised in Table 6.1 allow you to interface with the interpreter at a more basic level than normal Global Cobol. By using them you can perform efficiently certain operations which are either very slow, or simply not available, in Global Cobol itself.

For example, you can move or compare character strings whose lengths are not known until run-time with a short sequence of intermediate code statements. The same processing in Global Cobol would involve defining both strings as single-character arrays and using a DO loop or equivalent to move or compare the strings character by character, incrementing and testing an index. This might be **several hundred** times slower than the intermediate code implementation.

Similarly, you can use intermediate code to perform conversions from computational to display numeric, or vice versa, of quantities whose format (i.e. size and number of decimal places) is only known at run-time.

You may write intermediate code statements anywhere within the procedure division of a Global Cobol program and mingle them freely with other Global Cobol statements. An intermediate code statement is readily identified because its verb begins with a $ character. The verb may be preceded by a paragraph name, and the statement may be commented, in the normal way.

For complete and detailed documentation of the intermediate code instructions you should refer to Chapter 4 of the Global Development Toolkit Manual.

INTERMEDIATE CODE STATEMENT	**REF**	**DESCRIPTION**
	Character processing instructions	
$SET X [L]	6.3.1	Set source string register
$MOVE X [L]	6.3.2	Move source string to destination string
$COMP X [L]	6.3.3	Compare source string with destination string
$EXCH X [L]	6.3.4	Exchange source string with destination string
	Transfer of Control Instructions	
$JUMP cond label	6.4.1	Jump to label depending on cond
$CALL cond label	6.4.2	Call label depending on cond
$STOP cond code	6.4.3	Stop with code depending on cond
$EXIT cond code	6.4.4	Exit with code depending on cond
	Computational Instructions	

$LOAD C [F]	6.5.1	Load computational variable into accumulator
$STORE C [F]	6.5.2	Store computational variable from accumulator
$ADD C [F]	6.5.3	Add to accumulator
$ADDS C	6.5.4	Add to accumulator and store
$SUB C [F]	6.5.5	Subtract from accumulator
$SUBS C	6.5.6	Subtract from accumulator and store
$MUL C [F]	6.5.7	Multiply accumulator
$MULS C	6.5.8	Multiply accumulator and store
$DIV C [F]	6.5.9	Divide into accumulator
$DIVS C	6.5.10	Divide into accumulator and store
$AND C [F]	6.5.11	Perform logical AND
$ANDS C	6.5.12	Perform logical AND and store
$RND C [F]	6.5.13	Round accumulator
$RNDS C	6.5.14	Round accumulator and store
$STUS C [F]	6.5.15	Store unsigned from accumulator
Numeric Conversion Instructions		
$BIN N [F]	6.6.1	Convert display numeric variable into binary in accumulator
$DEC N [F]	6.6.2	Convert accumulator to decimal display numeric
Pointer Handling Instructions		
$PUSH A [L]	6.7.1	Push pointer onto parameter stack
$POP P	6.7.2	Pop pointer from parameter stack
Indexed Variable Management Instructions		
$LOADI C	6.9.1	Load Index (I register)
$LOADQ C	6.9.2	Load Qualifier (Q register)
$LOADL C	6.9.3	Load Length (L register)
Other Instructions		
$TRAP "trap-id"	6.10.1	Force trap program check
$RESUME	6.11.1	Resume failed program

Table 6.1 - Intermediate Code Statements

6.2 The Virtual Machine Concept

To understand how to use intermediate code you require an outline appreciation of how the interpretive system works. This is best approached by considering the interpreter to be a rather simple "virtual computer". The instructions the virtual CPU obeys, and the internal registers with which it works, are, naturally enough, particularly suitable for implementing Global Cobol. This section therefore provides an overview. Sections 6.3 onward describe the individual intermediate code statements in detail, in terms of the concepts introduced below, and give examples of their use.

6.2.1 Instructions, Operands and Qualifiers

Each Global Cobol statement may expand into several instructions to be executed by the virtual computer. Similarly this applies to intermediate code statements, which generate multiple instructions when they have indexed operands, for example. Nevertheless, in the case of an intermediate code statement only one **main** instruction is generated. In the subsequent description the Move instruction, for example, is the main instruction associated with the $MOVE statement, the Push instruction that associated with the $PUSH statement, and so on.

The virtual machine is a simple one-address computer. Each main instruction processed supplies the virtual CPU with an **operand** and a **qualifier**. The qualifier indicates the length, in bytes, of a character operand, or the format of a display numeric or computational operand. When the qualifier represents a **length** in bytes then it is a positive integer no greater than 65,535 (32,767 under V5.0 or earlier).

A **format** qualifier is also an integer, encoded so as to contain the sign option and number of decimal places before and after the point. The value, v, of a format qualifier is given by the formula:

$$v = 128s + 16q + p$$

where:

s is 1 if the item is signed, 0 otherwise;

q is the number of places following the decimal point, between 0 and 7 inclusive;

p is the number of places before the decimal point, between 1 and 15 inclusive.

Further, to be valid, $0 < p + q < 19$. For example, the format value for a PIC S9 (4,2) item is 4 + 32 + 128 = 164.

The quantity s is not used in format values applying to computational operands, which are always considered to be capable of holding negative values.

The format value also implies the length in bytes of the display numeric or computational operand to which it applies. This is a function of the quantities s, p and q.

6.2.2 The Accumulator

Arithmetic and numeric conversion is performed using an internal register of the virtual machine known as the accumulator. The accumulator holds the binary value of a quantity in its mill, and the format of this quantity in its scaling register. Whenever any arithmetic or conversion instruction is executed its format qualifier determines the scaling or conversion requirements of the operand, and its size in bytes.

6.2.3 The Overflow Flag

The Store, Decimal and Binary instructions concerned with numeric conversion can all fail to complete properly because the format of the operand is incompatible with the value in the accumulator. For example, the accumulator might contain a value that is too large to be stored in a PIC 9(4) COMP field specified as the operand of a Store instruction. In such a case the offending instruction is terminated and the virtual machine's overflow flag is set on. The contents of the accumulator, which may have been corrupted before the condition was detected, are usually unpredictable.

The arithmetic instructions (ADD, SUB, MUL and DIV) can suffer overflow for reasons described in Section 4.3.2.

The overflow flag is tested and cleared by the Global Cobol ON OVERFLOW statement **immediately** following the intermediate code statement which causes the condition. If such a statement is not present the offending program will be terminated in error.

6.2.4 Conditional Transfer of Control

The transfer of control instructions conditionally operate according to the magnitude of the accumulator.

6.2.5 Character String Handling

The character instructions process strings which are defined by the starting location of their operand together with its length qualifier. In character operations the virtual machine makes use of an internal **source string register** which is able to hold the location and length of a string. This register is initialised by the Set instruction.

The source string may be moved to, compared with or exchanged with a destination string specified by the operand and qualifier of a Move, Compare or Exchange instruction. The sign of the accumulator indicates the result of a comparison, so that a conditional instruction can be used following the compare instruction.

6.2.6 Pointer Handling

Pointer manipulation takes place using the **parameter stack**, an internal work area within the virtual machine. The Push instruction places a pointer to its operand on the stack, whereas the reverse instruction, Pop, removes the top pointer from the stack and stores it in the first two bytes of its operand.

The parameter stack is also employed to transfer parameters supplied in the USING clause of a CALL statement to the USING clause of an ENTRY statement.

6.2.7 The Trap Flag

Each instruction processed by the virtual machine contains a one-bit trap flag. When this flag is on, the interpreter generates a trap program check which is reported by Global System Manager. Ordinary Global Cobol procedure division statements are compiled into instructions in which the trap flag is off, and normally the programmer employs the debugging system to manipulate trap flags. However, using the intermediate code $TRAP instruction (see 6.9.1), you can generate an instruction with its trap bit set so as to establish an automatic, identified breakpoint.

6.3 Character Processing Instructions

Four intermediate code statements are provided to allow you to move or compare character strings whose lengths are not known until run-time. The statements consist of a verb followed by one or two arguments and are coded:

```
$SET X [L]
$MOVE X [L]
$COMP X [L]
$EXCH X [L]
```

The second argument of each, the length qualifier, L, is optional. When present it must be the name of a computational literal or a simple (i.e. non-indexed) computational variable containing the length in bytes of the character string operand, which may be zero except under Global 5.0 or earlier.

Important note: The first argument to $SET, $MOVE, $COMP or $EXCH instructions is treated as a character variable. However, the compiler does not check that it is a character variable and if it is not then the compiler will accept it but will generate a qualifier equal to the byte length of the variable concerned. For example, If $SET has a PIC 9(4) COMP computational variable as an argument then it will treat it as if it were a PIC X(2) variable. This is permitted so as to facilitate complex operations without proliferating the redefinition of the fields concerned. You must be particularly careful when coding literal arguments as $SET 1234 and $SET "1234" have very different meanings; the first sets a PIC X(2) field with the decimal value of 1234 (i.e. #04D2) whereas the second sets a PIC X(4) of value 1234 (i.e. #31323334).

The first argument, X, is a character variable or, for $SET and $COMP only, a literal. If a variable, it may be indexed. When the second argument is **not** present the length of X is derived from its Global Cobol picture clause. However, if the second argument **is** present its value overrides the picture clause information, which is ignored. In particular, if the second argument is present and X is indexed, then the length used for indexing is derived from the second argument, not the picture clause of X. As the length supplied overrides the qualifier, supplying an L field if X is a literal will have unpredictable results. A length field should therefore not be supplied if X is a literal.

Note that X and L must be variables or literals. Figurative constants are **not** allowed in intermediate code instructions.

6.3.1 $SET - Set Source String Register

The $SET statement initialises the source string register to identify its character string operand, X, as the current source string. The length qualifier, from L or the picture clause of X, determines the length in bytes of the source string.

The source string itself is not modified by the $SET statement.

6.3.2 $MOVE - Move Source String to Destination String

The $MOVE statement transfers the source string to the destination string beginning at the first byte of its character string operand, X. The length qualifier, from L or the picture clause of X, determines the length in bytes of the destination string.

The move takes place one byte at a time from the leftmost (low location) byte to the rightmost (high location) byte. If the source string is **longer** than the destination string character truncation takes place, the extra rightmost bytes of the source string being ignored.

If the source string is **shorter** than the destination string the extra rightmost bytes of the destination string are padded with ASCII blanks. The source string register and the source string itself are not modified by the $MOVE statement.

6.3.3 $COMP - Compare Source String with Destination String

The $COMP statement compares the source string with the destination string beginning at the first byte of its character string operand, X. The length qualifier, from L or the picture clause of X, determines the length in bytes of the destination string.

The comparison takes place one byte at a time, from the leftmost byte to the rightmost. The bytes being compared are treated, for the purposes of the comparison, as 8-bit unsigned numbers. If the source and destination strings are of unequal length then the shorter will be considered to be extended to the right with ASCII blanks for the comparison.

The result of the comparison is returned in the accumulator, which will be set to either -1, 0 or +1:

-1	means the source string is less than the destination string;
0	means the source string is equal to the destination string;
+1	means the source string is greater than the destination string;

The source string register, source string and destination string are not modified by the $COMP statement.

6.3.4 $EXCH - Exchange Source String with Destination String

The $EXCH statement exchanges the source string for the destination string and vice versa. It does this one byte at a time, starting from the left.

6.3.5 Character String Examples

In the examples below SOURCE and DEST are the data names of character strings whose lengths are held in the computational variables SL and DL respectively.

6.3.5.1 Variable Length Move

```
$SET SOURCE SL                      * ESTABLISH SOURCE STRING
$MOVE DEST DL                       * MOVE TO DESTINATION
```

SOURCE is moved to DEST and is itself unaffected by the operation. Character truncation will take place if SOURCE is longer than DEST. However, if DEST is longer than SOURCE the extra rightmost bytes are padded with ASCII blanks.

6.3.5.2 Setting a Variable Length String to Blanks

```
$SET ""                             * SOURCE STRING IS BLANKS
$MOVE DEST DL                       * MOVE TO DESTINATION
```

This is really just a special case of a variable length move. We may not use the figurative constant SPACES in an intermediate code instruction, so we code the literal "", representing a zero-length character string, instead. This has the desired effect because the Move instruction sets all characters not supplied by the source string to blanks in the destination string.

6.3.5.3 Variable Length Compare

```
$SET SOURCE SL                      * ESTABLISH SOURCE STRING
$COMP DEST DL                       * COMPARE WITH DESTINATION
$JUMP LT SRLOW                      * GO IF SOURCE LESS DEST
$JUMP GT SRHIGH                     * GO IF SOURCE GREATER DEST
                                    * PROCESSING WHEN STRINGS ARE EQUAL
```

SOURCE is compared with DEST and the accumulator is set up to indicate the result. Two $JUMP statements are used to route control to the appropriate processing logic when the source string is less than or greater than the destination string. However, control "drops through" the $JUMP statements when the two strings are equal.

6.3.5.4 Checking for a Blank Variable Length String

```
$SET ""                             * SOURCE STRING IS BLANKS
$COMP DEST DL                       * COMPARE WITH DESTINATION
$JUMP EQ BLANK                      * GO IF DEST IS BLANK
```

This is really just a special case of a variable length compare. We may **not** use the figurative constant SPACES in an intermediate code instruction, so we code the literal "", representing a zero-length character string, instead. This has the desired effect because the Compare instruction considers all characters not supplied by the source string to be blank for the purpose of the comparison with the destination string.

6.4 Transfer of Control Instructions

These instructions allow you to transfer control depending upon the condition of the accumulator. The following conditions are used:

EQ	perform the instruction if accumulator equals zero;
NE	perform the instruction if accumulator not equal to zero;
GT	perform the instruction if accumulator greater than zero;
LT	perform the instruction if accumulator less than zero;
GE	perform the instruction if accumulator greater than or equal to zero;

LE perform the instruction if accumulator less than or equal to zero.

These statements do not themselves modify the accumulator. However, most Global Cobol statements do corrupt it and hence a Transfer of Control statement should normally be coded immediately following the intermediate code statement responsible for establishing the accumulator value to be tested.

There is one exception to this rule: if the statement originally responsible for establishing the accumulator value to be tested can cause the overflow flag to be set, then an ON OVERFLOW conditional should be coded immediately following this statement. The Transfer of Control statement should then be the next statement to be executed when overflow does **not** occur.

Providing the overflow flag is not set, ON OVERFLOW leaves the value in the accumulator unchanged. Thus, in the case when there is no overflow, the Transfer of Control statement still obtains the accumulator value established by the original statement, even though overflow handling logic has been inserted between the Transfer of Control statement and the accumulator setting statement.

6.4.1 The $JUMP Statement

The $JUMP statement allows you to pass control to a section or paragraph conditionally, according to the value of the quantity in the accumulator. You code:

```
$JUMP cond label
```

where label is a section or paragraph name and cond indicates the condition under which the transfer of control takes place. For example:

```
$JUMP EQ AA090
```

will transfer control to AA090 if the accumulator is equal to zero, just as if you had coded GO TO AA090. However, if the accumulator is non-zero, the $JUMP statement will not transfer control and the statement immediately following it will be executed next.

6.4.2 The $CALL Statement

The $CALL statement allows you to perform a subroutine conditionally. You code:

```
$CALL condition label
```

Where condition is one of the six conditions described above and label is the section or paragraph name of the subroutine you wish to be performed if the condition is valid. For example:

```
$CALL NE BB080
```

will perform the section of the program starting at BB080 if the accumulator is not equal to zero. If the accumulator is equal to zero then the next instruction will be executed instead.

6.4.3 The $EXIT Statement

The $EXIT statement conditionally causes control of the program to be returned to the statement following the last outstanding CALL, PERFORM, EXEC or $CALL statement executed. You code:

```
      $EXIT condition code
```

Where condition is one of the six conditions listed above and code is the exit code to be displayed in the EXIT WITH... message.

6.4.4 The $STOP statement

The $STOP statement causes the program to terminate with the specified code if the condition stated is met. You code:

```
      $STOP condition code
```

Where condition is one of the six conditions listed above and code is the code to be displayed in the STOP message. (0 = STOP RUN)

6.4.5 Examples

In the following example control is passed to AA090 if the display numeric variable NUMST is not a positive numeric string compatible with the format specified in the format qualifier NF.

```
      $BIN NUMST NF                           * CONVERT TO BINARY
      ON OVERFLOW GO TO AA090                 * INCOMPATIBLE FORMAT
      $JUMP LE AA090                          * JUMP IF NOT POSITIVE
```

The $JUMP statement is most frequently used following a character string compare, as shown in the examples of 6.7.3 and 6.6.4 below.

6.5 Computational Instructions

6.5.1 $LOAD - Load Computational Variable to Accumulator

The $LOAD statement places the value of its computational operand, C, in the accumulator. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually transferred. C itself is not changed by the operation.

6.5.2 $STORE - Store Computational Variable from Accumulator

The $STORE statement transfers the value in the accumulator to its computational operand, C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes at C actually affected. The accumulator itself is **corrupted** by the operation.

Note that if the format qualifier indicates that the number in the accumulator exceeds the capacity of C, then the overflow flag will be set. In this case C itself remains unchanged.

6.5.3 $ADD - Add to Accumulator

The $ADD statement adds the value of its computational operand, C, to the value already held in the accumulator. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.4 $ADDS - Add to Accumulator and Store

The $ADDS statement adds the value of its computational operand, C, to the value already held in the accumulator and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.5 $SUB - Subtract from Accumulator

The $SUB statement subtracts the value of its computational operand, C, from the value held in the accumulator. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.6 $SUBS - Subtract from Accumulator and Store

The $SUBS statement subtracts the value of its computational operand, C, from the value held in the accumulator and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.7 $MUL - Multiply Accumulator

The $MUL statement multiplies the value held in the accumulator by its computational operand, C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.8 $MULS - Multiply Accumulator and Store

The $MULS statement multiplies the value held in the accumulator by its computational operand, C and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.9 $DIV - Divide into Accumulator

The $DIV statement divides the value held in the accumulator by its computational operand, C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.10 $DIVS - Divide into Accumulator and Store

The $DIVS statement divides the value held in the accumulator by its computational operand, C and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.11 $AND - Perform logical AND

The $AND statement performs a logical AND on the value held in the accumulator and the value of its computational operand, C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.12 $ANDS - Perform logical AND and Store

The $AND statement performs a logical AND on the value held in the accumulator and the value of its computational operand, C and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.13 $RND - Round Accumulator

The $RND statement scales the accumulator to match the target field and rounds the last digit of the result to the nearest digit. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed. C itself is not changed by the operation.

6.5.14 $RNDS - Round Accumulator and Store

The $RNDS statement scales the accumulator to match the target field, rounds the last digit of the result to the nearest digit and stores the result in C. The format qualifier, from F or the picture clause of C, determines the scaling assumed as well as the number of bytes of C actually processed.

6.5.15 $STUS - Store Unsigned

The $STUS statement stores the accumulator in the target field, C, without checking for overflow. The format qualifier, from F or the picture clause of C, determines the number of bytes actually transferred.

Important note: These instructions, although taking computational variables for final arguments, will permit the coding of **any** data type argument in this position. The compiler will deduce appropriate qualifiers for non-computational variables depending on byte length. For example, If a PIC PTR field with a length of 2 bytes is coded then the compiler will deduce a PIC 9(4) COMP qualifier for it. This is permitted so as to facilitate complex operations without proliferating the redefinition of the fields concerned. Because of this, however, you must take care when coding arguments using the arithmetic intermediate code statements as the compiler will not check that they are computational variables.

6.6 Numeric Conversion Instructions

Two intermediate code statements are provided to allow you to perform numeric conversions when the format of one or more of the quantities involved is not known until run-time. Using combinations of the statements, together with the $LOAD and $STORE instructions described above, you can convert computational quantities to display numeric and vice versa; normalise display numerics by converting valid numeric strings to standard numeric string form; and move variable format computational fields.

The numeric conversion statements consist of a verb followed by one or two arguments and are coded:

```
$BIN N [F]
$DEC N [F]
```

The second argument of each, the format qualifier, F, is optional. When present it must be a computational literal or the name of a **simple** (i.e. non-indexed) computational variable whose value specifies the format of the first, computational or display numeric, operand as described in 6.2.1.

The first argument, C or N, is, respectively, either a **C**omputational or display **N**umeric variable or for $BIN only, a literal. If a variable, it may be indexed. When the second argument is not present the format of C or N is derived from its Global Cobol picture clause. However, if the second argument is present its value overrides the picture clause information, which is ignored. In particular, if the second argument is present and C or N is indexed, then the length used for indexing is derived from the second argument, not the picture clause of C or N.

Note that C, N and F must be variables or literals. Figurative constants are not allowed in intermediate code instructions.

6.6.1 $BIN - Convert Display Numeric Variable to Binary

The $BIN statement converts its display numeric operand, N, to a binary (i.e. computational) value which it places in the accumulator. The format qualifier, from F or the picture clause of N, determines the scaling and the number of bytes of N which participate in the conversion. N itself remains unchanged by the operation.

If N is not a valid numeric string, compatible with the format qualifier, the operation will be terminated and the overflow flag will be set. This will occur if the string N is signed when the format specified indicates it should be unsigned; or contains a decimal point when it should be an integer; or is too large.

6.6.2 $DEC - Convert Accumulator to Decimal (Display Numeric)

The $DEC statement converts the binary value in the accumulator to a decimal (i.e. display numeric) value which it places in its display numeric operand, N. The format qualifier, from F or the picture clause of N, determines the scaling and the number of bytes of N actually affected.

The overflow flag is set, and N remains unchanged, if the value in the accumulator is too large for the format specified or if it is negative when an unsigned format is specified.

6.6.3 Numeric Conversion Examples

In the examples below NUMST is the data name of a numeric string whose format value is contained in NF, a computational variable. COMP is a computational variable whose format (yet another computational variable) is held in CF.

6.6.3.1 Computational to Display Numeric Conversion

```
$LOAD COMP CF                          * COMP TO ACCUMULATOR
$DEC NUMST NF                          * ACCUMULATOR TO DISP. NUM.
[ON OVERFLOW ... etc.                  * IF ERROR]
```

COMP is converted to a standard numeric string in NUMST unless the format value in NF is incompatible with the value in the accumulator following the $LOAD, in which case the overflow flag is set by $DEC.

6.6.3.2 Display Numeric to Computational

```
$BIN NUMST NF                              * DISP. NUM TO ACCUMULATOR
[ON OVERFLOW ... etc.                      * IF INVALID DISP. NUM]
$STORE COMP CF
[ON OVERFLOW ... etc.                      * IF COMP TOO SMALL]
```

Providing NUMST is a valid numeric string and COMP has sufficient capacity this sequence will convert the numeric string to a computational value in COMP.

The $BIN statement will set the overflow flag if NUMST is not a valid numeric string and in this case the first ON OVERFLOW processing indicated will take place. The $STORE statement will set the flag if COMP has insufficient capacity for the converted value and then the second ON OVERFLOW processing will gain control.

6.6.3.3 Display Numeric to Display Numeric Conversion

```
$BIN NUMST NF                              * DISP. NUM. TO ACCUMULATOR
[ON OVERFLOW ... etc.                      * IF INVALID DISP. NUM]
$DEC NUMST NF                              * STANDARD STRING TO NUMST
```

Providing NUMST is a **valid** numeric string this sequence will convert it to a **standard** numeric string. If NUMST is invalid the $BIN statement will set the overflow register.

6.6.3.4 Computational to Computational Conversion

(In this example W is defined as PIC S9(11,7) COMP)

```
$LOAD COMP CF                              * MOVE COMPUTATIONAL
$STORE W                                   * TO WORK FIELD
[ON OVERFLOW ... etc.                      * IF TOO LARGE]
ADD 1 TO W                                 * ARITHMETIC ON
[ON OVERFLOW ... etc.                      * IF TOO LARGE]
$LOAD W                                    * WORK FIELD
$STORE COMP CF                             * RETURN TO COMP
[ON OVERFLOW ... etc.                      * IF TOO LARGE]
```

The first load and store sequence moves COMP to W, a conventionally defined computational work field. $STORE sets the overflow flag if W is of insufficient capacity to contain COMP. Next conventional arithmetic takes place on W. In the example this only involves adding 1, but the principle of moving run-time defined computational fields to large fixed-format fields to perform arithmetic is widely applicable. Finally another load and store sequence is used to move W back to COMP. The overflow register will be set if the addition of 1 has caused the capacity of COMP to be exceeded.

6.7 Pointer Handling Instructions

There are two pointer handling statements, $PUSH and $POP. They are used in the processing of table entries where the length of each entry, in bytes, is only known at run-time.

6.7.1 $PUSH - Push Pointer onto Parameter Stack

The $PUSH statement is coded:

```
$PUSH A [L]
```

The first argument, A, can be a variable or literal. The optional second argument, the length qualifier, L, has no effect if the variable A is not indexed. When present it must be a computational literal or the name of a simple (i.e. non-indexed) computational variable containing the non-zero length in bytes of the entries belonging to the table referenced by the first argument.

$PUSH calculates an address from its arguments and stores a pointer to it at the top of the parameter stack. If the first argument is **not** indexed, the address used is simply that of the first argument itself. However, when the first argument is indexed, T(N) say, the pointer addresses the byte at location:

```
t + (n - 1) * length
```

where:

t	is the address of the first occurrence of T;
n	is the value of the index N;
length	is either given by the second argument, or if this is omitted, is derived from the first argument, as explained below:

- If A is a repeating group, or is contained within a repeating group, then the length is taken as the size of the repeating entry itself;
- If A is a non-repeating group or subgroup, the length is the size of that group or subgroup;
- If A is an elementary item, with or without an OCCURS clause, the length is the size implied by the item's picture clause.

6.7.2 $POP - Pop Pointer from Parameter Stack

The $POP statement is coded:

```
$POP P
```

Its argument must be a pointer variable, and it may be indexed. The effect of $POP is to remove the pointer currently at the top of the parameter stack and place it in its operand, P.

6.7.3 Programming Notes

Items are placed on the parameter stack by $PUSH statements and CALL statements with USING clauses. They are removed by $POP statements and ENTRY statements with USING clauses. Note that the following four sequences generate identical code:

```
CALL RTN USING A B C                    * ONE
```

```
      $PUSH A                                 * TWO
      CALL RTN USING B C                      *

      $PUSH A                                 * THREE
      $PUSH B                                 *
      CALL RTN USING C                        *

      $PUSH A                                 * FOUR
      $PUSH B                                 *
      $PUSH C                                 *
      CALL RTN                                *
```

In all four cases an ENTRY statement of the form:

```
      ENTRY RTN USING ALPHA BETA GAMMA
```

will pop the three pointers from the stack and use them to base the proper linkage section groups ALPHA (overlaying A), BETA (overlaying B) and GAMMA (overlaying C).

It is important that all information placed on the stack by $PUSH statements is removed by corresponding $POP or ENTRY statements. If this is not the case the effect will be to pass too many parameters to some entry point. For example, as we have already seen, the sequence:

```
      $PUSH X
      .
      .
      .
      [$POP statement erroneously omitted]
      CALL SUBR USING A
```

will pass to the entry point at SUBR not the parameter A, which it expects, but the two parameters X and A.

When an ENTRY statement receives the wrong number of parameters the overflow flag is set and therefore, unless the statement is immediately followed by an ON OVERFLOW statement (as explained in 3.5) the program will be terminated in error.

6.7.4 Pointer Handling Examples

6.7.4.1 Address Calculation using the Parameter Stack

This first example is rather artificial, and is intended to show that the parameter stack can be used to perform up to 7 address calculations at one time, using a variety of different data definitions.

Suppose the data division of a program contains the following statements, where the numbers on the left indicate the decimal address at which the first, or only, occurrence of each field begins:

```
      1000  77    I     PIC 9(4) COMP
      1002  01    Y     OCCURS 10               * 10 10-BYTE ENTRIES
      1002    03  YA    PIC X(5)
      1007    03  YB    PIC X(5)
      1102  01    Z                             * 14 BYTE TABLE
      1102    03  ZP    OCCURS 7 PIC PTR
```

Then the procedure division statements shown below have the effect indicated in their comments and eventually leave the parameter stack empty:

```
      $PUSH I                                   * 1000 TO PARAMETER STACK
      $PUSH YB(3)                               * 1027 TO PARAMETER STACK
      $PUSH Z(4)                                * 1144 TO PARAMETER STACK
      $PUSH ZP(3)                               * 1106 TO PARAMETER STACK
      $PUSH I(2) 5                              * 1005 TO PARAMETER STACK
      $PUSH YA(2) 5                             * 1007 TO PARAMETER STACK
      $PUSH Y(2) 5                              * 1007 TO PARAMETER STACK
      MOVE 1 TO I                               *
      DO UNTIL I = 8                            * 1007 TO ZP(1)
            $POP ZP(I)                          * 1007 TO ZP(2)
            ADD 1 TO I                          * 1005 TO ZP(3)
      ENDDO                                     * 1006 TO ZP(4)
                                                * 1144 TO ZP(5)
                                                * 1027 TO ZP(6)
                                                * 1000 TO ZP(7)
```

Note that, even though valid code will be generated, the compiler will flag the statements:

```
      $PUSH Z(4)
      $PUSH I(2) 5
```

with warning messages because their first arguments, although coded as indexed variables, are not actually data items with OCCURS clauses or members of repeating groups.

6.7.4.2 Dynamic Table Handling

In the following rather more realistic examples, TABLE is defined as a 2000-byte area by the following statement:

```
      77    TABLE OCCURS 2000 PIC X
```

TABLE actually consists of an array of contiguous equal-sized entries, but the length of each entry is not known until run-time. This length is held in the computational variable LENGTH. The computational variable N contains a positive integer, n, which is the number of the table entry you require to process, entry 1 being, of course, the first entry at the beginning of the table. P is a pointer variable, defined in working storage.

Each entry consists of a type flag, a counter, and a variable length key. The entry is described by the based area EN, defined as follows:

```
      01    EN BASED P
        02  ENTYPE      PIC X                   * ENTRY TYPE
        02  ENCNTR      PIC 9(4) COMP           * COUNTER
        02  ENKEY       PIC X                   * VAR LENGTH KEY
```

6.7.4.3 Basing a Selected Entry of the Table

```
      $PUSH TABLE(N) LENGTH                     * POINTER TO STACK
      $POP P                                    * STACK TO P,
      POSITIONING EN
```

The first $PUSH statement calculates the location of the n'th entry and places a pointer to it on the top of the parameter stack. The subsequent $POP statement transfers this pointer to P, thereby positioning the based area EN. Fields within EN can now be accessed using normal Global Cobol statements, although ENKEY would probably have to be handled by intermediate code, since its actual length is not given by its picture clause, but depends on the value of LENGTH.

6.7.4.4 Passing a Selected Entry to a Subroutine

```
$PUSH CA                               * PASS FIRST PARAMETER
$PUSH TABLE(N) LENGTH                  * PASS SECOND PARAMETER
CALL PROC                              * PASS CONTROL
```

The parameters are pushed onto the stack in the order that they are required by the ENTRY PROC statement, which will be of the form:

```
ENTRY PROC USING CA EN
```

6.8 Indexed Variable Management Instructions

When both an index and a qualifier are specified on an intermediate code statement, and the variable is not within a repeating group, the length used when indexing is derived from the qualifier, and not from the picture clause of the target. Hence if the table entry length is not the same as the length given by the qualifier, an extra linkage section variable must be used to access the field, as shown in the examples below. If, however, the variable is within a repeating group, the length of the repeating group is always used when indexing.

The examples all use character variables, although exactly the same principles apply to computational variables. In them TAB-1 is a PIC X OCCURS 24 character variable containing "ABC...X". TENT is an 01 level linkage section variable. All the examples consist of moving part of the table into a receiving character variable DEST. Computational variables TL, TI contain the table length and index.

6.8.1 Variable Length Table Entry

The table is to be treated as entries whose length is given by TL, and the TI'th entry is required.

```
$SET TAB-1(TI) TL                      * LENGTH TL FOR INDEX
$MOVE DEST
```

If TI contained 2, and TL 3, then this would set DEST to "DEF".

6.8.2 Variable Part of a String

DEST is to be set to the TL characters starting at position TI within the string.

```
BASE TENT AT TAB-1(TI)                 * LENGTH 1 FOR INDEX
$SET TENT TL
$MOVE DEST
```

If TI contained 2, and TL 3, then this would set DEST to "BCD".

6.8.3 Variable Length String from Table of Fixed Length Entries

The table is to be treated as 4 byte entries, and the first TL bytes of entry TI are required. In order to force a fixed entry length to be used, the table must be redefined as a PIC X(4) item within a repeating group:

```
      01    FILLER REDEFINES TAB-1 OCCURS 6
        03  TAB-4 PIC X(4)
```

and to access the required portion you code:

```
      $SET TAB-4(TI) TL
      $MOVE DEST
```

If TI contained 2, and TL 3, then this would set DEST to "EFG".

6.9 Register Load Instructions

6.9.1 $LOADI - Load Index

The $LOADI statements places the value of its computational operand, C, in the I register. The register must then be used for indexing by the next statement, unless the next statement is a $LOADQ or $LOADL. Once the I register has been used for indexing it is zeroised.

6.9.2 $LOADL - Load Length

The $LOADL statements places the value of its computational operand, C, in the L register. Once the L register has been used for indexing it is zeroised.

6.9.3 $LOADQ - Load Qualifier

The $LOADQ statements places the value of its computational operand, C, in the Q register. This will override the qualifier of the next intermediate code statement, and once it used the Q is zeroised.

6.9.4 Programming Notes

If all or any combination of these instructions are used, then they should be set up in the order described, that is: $LOADI, $LOADL, $LOADQ.

6.10 Other Instructions

6.10.1 The $TRAP Statement

The $TRAP statement simply causes a trap program check when it is executed. It is coded:

```
      $TRAP "trap-id"
```

where "trap-id" is a character literal which will appear on the diagnostic report. Only the first five characters of the trap-id are used: a sixth or subsequent character will simply be ignored by the compiler when it expands the $TRAP statement. If the trap-id is less than five characters in length then it will be padded to five characters by rightmost ASCII blanks.

6.10.1.1 Examples

When the statement:

```
      $TRAP "INSPECT"
```

is executed from within program SALES, control is returned to the Global Cobol monitor which outputs the following message and prompt:

```
      $91 TRAP AT 43FE
      $50 DEBUG:
```

The hexadecimal number following the word AT indicates the Global Cobol location of the $TRAP statement. If you now reply D to the debug prompt the first two lines of the resultant diagnostic report will be:

```
      PROGRAM CHECK TYPE TRAP
      AT INSPE IN SALES
```

The first five characters of the trap-id are displayed following the word AT in the second line.

You can use debug to inspect and modify the interrupted program, or set additional traps in it, as explained in the Global Cobol User Manual. Eventually, you can resume execution of the program at the statement following the $TRAP statement.

6.10.1.2 Programming Note

Traps are usually set interactively by using the debug facilities. The $TRAP statement is provided to allow you to code a request for a trap explicitly in those few cases when interactive working cannot achieve the result you require. For example, you might only require a trap to occur when a particularly rare combination of values was detected. In this case you would include some testing logic in your program which would execute a $TRAP when the condition you were interested in arose.

6.10.2 $RESUME - Resume Failed Program

The $RESUME statement is mentioned here only for completeness. It should never be used.

INSTRUCTION	**STATEMENT**	**$CC CODE**	**OPERAND TYPE**					**Notes**
			X	**9**	**COMP**	**PTR**	**LABEL***	
Set	$SET		Y	Y	A	B	B	
Move	$MOVE		Y	Y	A	B	B	
Compare	$COMP		Y	Y	A	B	B	
Exchange	$EXCH	26	Y	Y	A	B	B	E
Binary	$BIN			Y				
Decimal	$DEC			Y				
Load	$LOAD		C		Y	C	C	
Store	$STORE		C		Y	C	C	
Round	$RND	7	C		Y	C	C	
Round+Store	$RNDS	107	C		Y	C	C	
Add	$ADD	8	C		Y	C	C	
Add+Store	$ADDS	108	C		Y	C	C	
Subtract	$SUB	9	C		Y	C	C	
Subtract+Store	$SUBS	109	C		Y	C	C	
Multiply	$MULT	10	C	Y	C	C	C	
Multiply+Store	$MULTS	110	C		Y	C	C	
Divide	$DIV	11	C		Y	C	C	
Divide+Store	$DIVS	111	C		Y	C	C	
Store Unsigned	$STUS	27	C		Y	C	C	
Logical AND	$AND	28	C		Y	C	C	
Logical AND+Store	$ANDS	128	C		Y	C	C	

Jump	$JUMP Q					Y	Y	
Call	$CALL Q	13				Y	Y	
Stop	$STOP Q	14			Y			F
Exit	$EXIT Q	15	Y		Y			G
Resume	$RESUME	16						I
Escape	None	17						L
Trace	None							J
Push	$PUSH		Y	Y	Y	Y	Y	
Pop	$POP					Y		
Pop List	None							K
Load Index	$LOADI	22	C		Y	C	C	
Load Qualifier	$LOADQ	23	C		Y	C	C	
Load Length	$LOADL	24	C		Y	C	C	
Call Display	None							H

* - undefined globals are classed as labels
A - treated as character string of appropriate length
B - treated as PIC X(2)
C - treated as computational integer of appropriate length
E - length of operand is ignored
F - operand conventionally PIC 9(4) COMP
G - operand conventionally PIC X(2); second byte contains condition code as PIC 9(2) COMP
H - Use DISPLAY statement I - No operand
J - SECTION and $TRAP statements generate trace
K - ENTRY and (UN)PRIVILEGED statements generate Pop List
L - Generated at start of assembler code routines (#8800)
Q - Must be one of EQ, NE, LT, GT, LE, GE, 8 (on exception) or 16 (on no exception) Y - Valid

Table 6.12 - Intermediate code instructions and $CC equivalents

6.11 The $CC Statement and SVC Statements

6.11.1 The $CC Statement

The $CC statement generates an intermediate code instruction as if the equivalent intermediate code statement had been coded. Intermediate code instructions with their equivalent $CC statements is shown in figure 6.12. For example, $CC 8 is equivalent to $ADD. This statement is provided only for compatibility with pre-V6.1 Global System Manager.

6.11.2 The SVC Statement

The SVC statement is similar to a CALL statement, but it invokes a routine (known as a 'system service') which is permanently resident within the monitor. It is coded as follows:

```
SVC nn [USING parameter...]
```

Where nn is a 1 or 2 digit number identifying the system service to be executed.

In practice there is little or no need for these services to be invoked directly as they are nearly all available as Global Cobol statements or as system subroutines. You may need to use an SVC statement if there is a special - machine dependent – facility available within the nucleus (for example, an interface to the native operating system).

There are three SVC calls that may occasionally useful in ordinary programs and these are described in the appropriate manuals:

SVC 14 - Search for lowest key (System Subroutines Manual)

SVC 25 - Display (Screen Presentation Manual)

SVC 48 - Calculate Cursor Position (Screen Presentation Manual)

SVC statements should be used with care and only by experienced programmers.

Appendix A - The ASCII Character Set

CHARACTER	DECIMAL CODE	HEX. CODE	HEX. CODE OF EQUIVALENT EBCDIC CHARACTER	REMARKS
NUL	0	00	00	CTRL SHIFT P. Null.
SOH	1	01	01	CTRL A.
STX	2	02	02	CTRL B.
ETX	3	03	03	CTRL C.
EOT	4	04	37	CTRL D.
ENQ	5	05	2D	CTRL E.
ACK	6	06	2E	CTRL F.
BEL	7	07	2F	CTRL G. Rings the bell.
BS	8	08	16	CTRL H. Backspace.
HT	9	09	05	CTRL I. Horizontal tab.
LF	10	0A	25	CTRL J. Line Feed.
VT	11	0B	0B	CTRL K. Vertical tab.
FF	12	0C	0C	CTRL L. Form Feed.
CR	13	0D	0D	CTRL M. Carriage Return.
SO	14	0E	0E	CTRL N.
SI	15	0F	0F	CTRL O.
DLE	16	10	10	CTRL P. (DC0).
DC1	17	11	11	CTRL Q. X-ON.
DC2	18	12	12	CTRL R.
DC3	19	13	(none)	CTRL S. X-OFF.
DC4	20	14	3C	CTRL T.
NAK	21	15	3D	CTRL U.
SYN	22	16	32	CTRL V.
ETB	23	17	26	CTRL W.
CAN	24	18	18	CTRL X.
EM	25	19	19	CTRL Y.
SUB	26	1A	3F	CTRL Z.
ESC	27	1B	27	CTRL [(or CTRL SHIFT K).
FS	28	1C	22	CTRL (or CTRL SHIFT L).
GS	29	1D	(none)	CTRL] (or CTRL SHIFT M).
RS	30	1E	35	CTRL ^ (or CTRL SHIFT N).
US	31	1F	(none)	CTRL - (or CTRL SHIFT O).
SP	32	20	40	Space.
!	33	21	5A	
"	34	22	7F	
#	35	23	7B	Sometimes pounds sign.
$	36	24	5B	
%	37	25	6C	
&	38	26	50	
'	39	27	7D	Apostrophe or Acute Accent.
(	40	28	4D	
)	41	29	5D	
*	42	2A	5C	
+	43	2B	4E	
,	44	2C	6B	Comma.
-	45	2D	60	Hyphen.
.	46	2E	4B	
/	47	2F	61	
0	48	30	F0	
1	49	31	F1	
2	50	32	F2	
3	51	33	F3	
4	52	34	F4	
5	53	35	F5	
6	54	36	F6	
7	55	37	F7	
8	56	38	F8	
9	57	39	F9	
:	58	3A	7A	
;	59	3B	5E	
<	60	3C	4C	
=	61	3D	7E	

CHARACTER	DECIMAL CODE	HEX. CODE	HEX. CODE OF EQUIVALENT EBCDIC CHARACTER	REMARKS
>	62	3E	6E	
?	63	3F	6F	
@	64	40	7C	
A	65	41	C1	
B	66	42	C2	
C	67	43	C3	
D	68	44	C4	
E	69	45	C5	
F	70	46	C6	
G	71	47	C7	
H	72	48	C8	
I	73	49	C9	
J	74	4A	D1	
K	75	4B	D2	
L	76	4C	D3	
M	77	4D	D4	
N	78	4E	D5	
O	79	4F	D6	
P	80	50	D7	
Q	81	51	D8	
R	82	52	D9	
S	83	53	E2	
T	84	54	E3	
U	85	55	E4	
V	86	56	E5	
W	87	57	E6	
X	88	58	E7	
Y	89	59	E8	
Z	90	5A	E9	
[	91	5B	AD	
\	92	5C	E0	
]	93	5D	BD	
^	94	5E	5F	Exponentiation symbol.
_	95	5F	6D	Underline.
□	96	60	79	Sometimes grave accent.
a	97	61	81	
b	98	62	82	
c	99	63	83	
d	100	64	84	
e	101	65	85	
f	102	66	86	
g	103	67	87	
h	104	68	88	
i	105	69	89	
j	106	6A	91	
k	107	6B	92	
l	108	6C	93	
m	109	6D	94	
n	110	6E	95	
o	111	6F	96	
p	112	70	97	
q	113	71	98	
r	114	72	99	
s	115	73	A2	
t	116	74	A3	
u	117	75	A4	
v	118	76	A5	
w	119	77	A6	
x	120	78	A7	
y	121	79	A8	
z	122	7A	A9	
{	123	7B	8B	
|	124	7C	6A	
}	125	7D	9B	Sometimes ALTMODE.
~	126	7E	A1	Tilde. Often a lead-in character.
DEL	127	7F	07	DELETE. RUBOUT.

Appendix B – Compiler and Cross-Reference Options

This appendix explains the various compiler and cross-reference options which can be controlled through OPT statements coded at the beginning of your program, as described in 2.3.1. In most cases the options are specified by pairs of codes of the form xx or Nxx, one member of the pair automatically becoming the default if neither xx nor Nxx is supplied. Thus if you do not provide an OPT ED or OPT NED statement, OPT NED applies automatically and the compiler will not force even byte data area alignment.

As well as supplying options by means of the OPT statement you can provide them at run-time by keying the appropriate option codes in response to the option prompt. This is described in the section of the Global Cobol User Manual which explains how to run $COBOL and $XREF. Those options you specify at run-time override any contradictory ones specified in OPT statements.

Many options, such as NSL which suppresses the listing of source lines, you will probably wish to supply only at run-time, whilst others, such as LN, are more appropriately made a feature of the program by means of an OPT statement.

Table B lists all the possible options, and indicates whether they apply to the compiler, or the cross-reference utility, or both. Any compiler option may be specified to the utility, and vice versa, meaningless options simply being ignored.

OPTION	**DESCRIPTION**	**COMPILER OPTION**	**X-REF OPTION**
BL/NBL	Print generated code in hexadecimal	Y	
CG/NCG	Display number of line being processed if <CTRL G> is keyed	Y	
CX/NCX	List contents of copy books*	Y	
ED/**NED**	Force even byte data alignment	Y	
LN/**NLN**	Use long (31 character) names	Y	Y
L$/**NL$**	Produce listing on first pass	Y	Y
PL=nnn	Page length (default $$PAGE)	Y	Y
PR/**NPR**	All ENTRY statements are privileged	Y	
SD/NSD	Generate symbolic debug information	Y	
SL/NSL	List all source lines*	Y	
SN/**NSN**	X-reference by section-id		Y
ST/**NST**	Print symbol table	Y	
TC/**NTC**	Print table of contents	Y	
TE/NTE	Terminate job management if errors are detected	Y	Y
TO/**NTO**	Print addition internal statistics	Y	
TR/NTR	Generate trace information	Y	
TW/**NTW**	Terminate job management if warnings are detected	Y	Y
XR/**NXR**	X-reference unreferenced items in copy books		Y

The default option is emboldened.

* These options may appear anywhere within the source.

Table B - Compiler and Cross-Reference Options

In the descriptions which follow, the option code pairs appear in parentheses following the headings to which they apply, and the default option is underlined.

List Contents of Copy Books (<u>CX</u>, NCX)
NCX (no copy book expansion) may be used to prevent the compiler from listing source lines introduced into the compilation as a result of COPY statements, unless such a line is flagged in error. The default, CX, causes the statements introduced by each COPY to be listed unless it is from a COPY with the SUPPRESS option.

If option NSL (no source lines) is in force the NCX/CX option is irrelevant since the copied statements, along with all the other source lines, will only be listed if they are flagged in error.

Binary List (BL, <u>NBL</u>)
The generated intermediate code is printed in hexadecimal on the right hand side of the listing. In order to understand this you will need to refer to chapter TK-4 of the Global Cobol Toolkit Manual. Any relocatable items, such as addresses, appear as a two byte offset followed by a single quote. Any references to globals also appear in this form. Note that specifying option BL causes program addresses printed in the data division to be incorrect in a few cases. In particular, the address associated with an 01 or 77 level item, or an FD or MD, is likely to be incorrect if either it or the preceding item is a redefinition, or if it is in the linkage section. Also, the hexadecimal printed against a statement can relate to the preceding statement, in particular if the previous statement was an uninitialised or partially initialised variable, or was a conditional statement (when the jump instruction may appear against the next line).

Line Number Display on <CTRL G> (<u>CG</u>, NCG)
The default setting of this option causes the line number of the line currently being processed by the compiler to be displayed on the screen if the operator keys <CTRL G>. The option can be disabled by selecting compiler option NCG.

Force Even Byte Data Area Alignment (ED, <u>NED</u>)
The ED option may be used to cause the compiler to force all file definitions, map definitions and level 01 group items to begin on an even byte boundary. It does this by inserting an extra binary zero byte before a file, map or level 01 definition if this is necessary to achieve the required alignment. This means, for example, that the fields of two adjacent level 01 items will not necessarily occupy contiguous storage, since if the first group declared an odd number of bytes the compiler will have inserted a binary zero byte after it.

Note that this option should be used with caution, and should not be used with Common or External Sections.

Option ED is for use when writing programs for machines such as the DEC PDP-11 where file access is improved whenever an integral number of 2-byte words, beginning on an even-byte boundary, participate in a read or write operation. Program files resulting from code using option ED will be aligned optimally, and will in general load faster than those which have not used the option. Relative sequential blocks established by the BLOCK CONTAINS clause will also be properly aligned, so sequential file processing times will be reduced.

Note that if you use option ED to optimise program loading time, but the first initialised data item in working storage begins on an odd boundary, then the option will not have the desired effect and, in consequence, the compiler will output a warning message. To avoid this possibility, code partially initialised groups and level 77 items **after** the first map definition, file definition, or fully initialised group appearing in the program.

Use Long Names (LN, NLN)
This option is common to the compiler and cross-reference utility. LN (long names) should be used if you wish to treat the first 31 characters of each symbol as a unique identifier.

The default, NLN, is that only the first 6 characters of each symbol are used as an identifier: symbols may be longer, but their seventh and subsequent characters are ignored by the compiler.

You should note that more table space is needed to handle long names, and this can cause the tables to overspill more frequently to backing storage, extending compilation times for large programs. This is unlikely to present a problem on configurations with a user area of 30,000 bytes or more.

Even when long names are specified the first 6 characters of each **global** symbol must be unique. Also, only the first 6 characters are made available in the tables used by symbolic debugging, so if two or more identifiers have their first 6 characters the same, only the first of them to appear in the program can be referenced symbolically at debug time.

First Pass Listing (L$, NL$)
The L$ option is mainly of use if the compiler fails during the first pass, since by specifying L$ and writing the listing directly to the printer, the line causing the failure can be determined. Option PR is described below along with the PRIVILEGED statement.

Generate Symbolic Debug Tables (NSD, SD)
NSD (no symbolic debug tables) may be used to prevent the compiler producing the tables that it normally prepares for the symbolic debugging system. These tables do not increase the size of the loaded program in any way, but they do occupy space in the compilation files and program files that reside on backing storage. Approximately 11 bytes of table space is required for each symbol used in the program.

Option SD, which is the default, causes symbolic debug tables to be produced in the normal way.

The Page Length (PL=nnn)
By default the listing is printed assuming that the page size in lines is the configuration standard (the value in $$PAGE: normally 66) defined when Global is installed or redefined using $CUS. A few lines

at the foot of each page are left blank in case the stationery is slightly misaligned.

The PL=nnn (page length) option allows you to change the page size by specifying the new length in lines, nnn, which must be an integer between 10 and 127 inclusive. For example, for a page size of 50 lines the option code would be PL=50.

All ENTRY Statements Privileged (PR, NPR)
In its normal mode of operation, the interpreter prevents you from updating fields which are outside the user area. This is to prevent you from accidentally corrupting system data or memory belonging to another user. However, the interpreter can be put into a privileged mode in which this check is bypassed, in order to allow system routines and the monitor to update protected fields. If you load a program or data onto the system stack this will be in a protected area of memory, and so you need to make programs that update fields on the system stack privileged.

If you load a Global Cobol program onto the system stack, any data in its working storage will be protected. Therefore you must compile it with option PR to allow it to access its working storage. Such a program must not call any unprivileged routines, as this will cause it to lose privileged status. In particular, most system routines, including access methods, are unprivileged and cannot be used. However, any form of the ACCEPT or DISPLAY statements can be used.

In order for a normal program to become temporarily privileged - for example, so that it can update data areas on the system stack - it must execute the statement:

```
    PRIVILEGED
```

and to relinquish this status, it must execute the statement:

```
    UNPRIVILEGED
```

Privileged status will also be lost if it executes any routine which is not itself privileged, or if the program exits to a higher level of control. You are recommended to put PRIVILEGED and UNPRIVILEGED statements around just those statements which update the protected fields.

Note that if a program is in privileged mode this has the side effect of preventing time-slicing until this status is relinquished. The program may still be swapped if it attempts I/O on a device which is busy or not ready, or if it executes a SUSPEND statement.

A detailed account of the privileged mechanism appears in section 2.6 of the Global Toolkit Manual.

Cross-Reference by Section Name (SN, NSN)
SN (section name) causes all the line numbers printed in the cross-reference listing to be prefixed with the first 2 characters of the name of the section in which they occur. If you have used the naming conventions described in Appendix E the first two characters of the section name will be a unique section code. References to working

storage items are prefixed "D.", and references to linkage section items are prefixed "L.".

The default, NSN, causes just the line numbers to be listed.

Produce a Symbol Table (ST, NST)
ST (symbol table) may be used if you require the compiler to produce a sorted symbol table following the compilation listing. The table contains the address, type and, where appropriate, picture clause information for each symbol appearing in the program. An example symbol table is shown in Appendix A of the Global Cobol User Manual.

The default, NST, is not to produce the symbol table. (The symbolic debugging feature obviates the need for the table in most cases.)

Produce a Table of Contents (TC, NTC)
TC (table of contents) may be used if you wish to compiler to produce a table of contents at the start of the listing, which lists all the page titles supplied in PAGE statements within the program, together with their line numbers. The table of contents has a maximum length of 58 lines.

The default, NTC, is not to produce the table of contents.

Terminate Job Management on Errors (TE, NTE)
NTE (no termination on errors) means that errors detected during a compilation will not cause job management to be terminated.

The default, TE, means that the compiler or cross-reference will terminate job management if it has detected any errors. Thus, for example, if a job file compiles and links a program, the default means that the linkage edit will not take place if any errors are detected.

Generate Trace Identifiers (NTR, TR)
NTR (no trace identifiers) may be used to reduce the amount of code the compiler generates for ENTRY and SECTION statements: each expands 6 bytes less code than when option TR, the default, is in force.

Because of the implications for debugging, you should normally only specify option NTR when storage is particularly critical: if a routine or section compiled with no trace identifiers is in the control path when the program is terminated in error, the diagnostic report you obtain will not contain the relevant entry or section identifier, although it will still supply you with the hexadecimal location of the entry point or section.

The NTR option is primarily intended for programmers developing storage-critical parts of Global System Manager. It should not normally be necessary in applications work.

Terminate Job Management on Warnings (TW, NTW)
TW (terminate on warnings) means that job management will be terminated at the end of the compilation or cross-reference if any warnings were generated.

The default, NTW, means that warnings will not cause job management to be terminated.

List Unreferenced Items in Copy Books (XR, NXR)
XR (cross-reference) may be used to cause symbols defined within copy books but not referenced in the program to appear in the cross-reference listing.

The default, NXR, will suppress unreferenced items defined in copy books, giving a more compact listing. Note that unreferenced items defined in the source program are always listed, as this often highlights an unused variable or label which could be deleted.

Appendix C - Summary of Restrictions

This appendix summarises the restrictions to be observed when developing general-purpose Global Cobol systems to run on as wide a possible range of configurations. Limitations imposed by the linkage editor and librarian are included for completeness, but the use of these programs is only described in detail in the Global Cobol User Manual.

The limits on program size, number of files open at any one time, and number of files per volume may be relaxed on larger systems. However, programs which exceed them do so at the risk of restricting the range of target configurations on which they may run.

Program Size
The compiler cannot handle a compilation larger than 32,767 bytes. In practice, however, the size of the available user area is a more restraining factor. On 8-bit machines, the Global System Manager user area is often restricted to slightly over 30K, and therefore we recommend that portable programs should be coded to execute within this limit. The largest possible Global System Manager user area is about 56K, but this is only realisable on machines which can directly address more than 64K.

These limits are not as onerous as they might seem given that the average amount of code generated by a procedure division statement is under 8 bytes. (When making program size estimates, however, you must remember that modules such as access methods and system routines will be included in your program, if it needs them, when it is linkage-edited. This topic is explained in more detail in Appendix D.)

Number of Levels of Subroutine
A maximum of 16 levels of subroutine are allowed within a program, or 12 if the lowest level itself calls a system routine, or uses SORT. A subroutine in this context is a group of procedure division statements entered by means of a CALL or PERFORM statement. The limit means that a program can have 16 CALL or PERFORM statements outstanding at any one time: a system routine may itself use another 3 levels of subroutine internally, hence the lower limit of 12.

Number of Nested Conditionals
Format 1 conditional structures can be nested up to 32 times. For example:

```
IF A=B
      :
      ON EXCEPTION
            :
            ON OVERFLOW
                  :
                  ACCEPT...NULL
                        :                  (up to 32 nestings allowed)
                        :
                  END
                  :
            END
            :
      END
      :
END
```

Number of Nested DO statements
DO statements can be nested up to 16 times. This nesting is independent of whatever nested conditionals are in force.

Number of Simultaneously Open Files
As a rule of thumb at least seven direct access files may be open at any one time by a program. However, the actual limit depends on the number of users of the system, and is no more restrictive than implied by the following formulae:

Maximum number of open FDs at any one time = 8 * number of users

Maximum number of **different** files open at any one time
= 8 * SQRT (number of users) rounded up.

These limits are system-wide, so under multi-user or networking versions of Global System Manager it is possible to mix programs which use a very large number of files with more modest jobs. For example, a four-user system will allow, at any one time, a maximum of 32 open FDs accessing 16 different files. At a particular instant 3 programs might each have 2 private work files and share 4 files with a fourth program which has another 10 files opened exclusively. In this case there are 3(2+4)+14 = 32) FDs open, accessing 4+10 = 14 different files.

The limit of 7 rather than 8 open files is recommended for a single user system because one file may be used by Global System Manager to access the program library, if one is attached.

All the above limits can be increased by updating the configuration file used to parameterise Global System Manager, as explained in the Global Configurator Manual. The figures quoted are the minimum values, used in the distributed configuration file.

Note that only one real printer file can be open for each printer the configuration possesses.

Number of Locked Regions
A maximum of 15 different file regions can be locked at any time using the LOCK statement described in the introduction to the File Maintenance Manual. This is a system-wide restriction. A program attempting to lock a region when the limit has been reached will be told that the region in question is already locked - and this situation will prevail until one of the existing locks is removed.

Number of Files per Volume
At least 63 files may be present on any direct access volume. Furthermore, up to 100 programs, compilations or copy books can be stored in a single library file of the appropriate type.

Number of Global Symbols per Compilation
The number of different global symbols defined or referenced by a program cannot exceed 127.

Total Number and Size of Symbols
The compiler overspills its internal tables onto a work file so that providing an irreducible minimum of main storage is available, this is

not a limiting factor as far as the symbols used per compilation is concerned. However, size constraints on the work file itself mean:

- The number of symbols per compilation is limited to about 1200;
- The total number of characters that make up the significant portions of the different symbols used is limited to about 10,000.

It is unlikely that either of these limits will affect you unless you develop huge programs using the long names option.

Symbolic Debugging
Only the first 650 symbols encountered in a compilation are available for use in symbolic debugging.

Copy Libraries
Each copy library can contain up to 100 books. Any number of copy libraries may be used, but the size of the compiler work file is increased by about 600 bytes for each library specified.

A copy book may itself contain a COPY statement, and the book thus referenced may contain a COPY statement, but that is as far as the nesting may go. Thus just two levels of internal nesting are allowed.

Linkage Editing Considerations
A maximum of 100 compilation files and compilation library files can be combined by the linkage editor to produce a program file. There must be no more than 100 compilations included in the program and they must not together contain more than 250 global definitions. The linkage edit will fail if any global symbol is defined more than once.

Compilation Libraries
You can store a maximum of 100 compilations in a single compilation library file, but the number of globals defined by the compilations, taken together, must not exceed 250.

Program Libraries
You can store a maximum of 100 linkage-edited programs in a single program library file. At run-time one application library file can be "attached" at a time. Global System Manager will then retrieve all application programs from this library.

If a program is not present in the library, Global System Manager will determine if there is a program file of the same name available on the program residence device, but it will not search a second library for the missing program.

Appendix D – Programming Notes

This appendix is intended for programmers who require to develop efficient, non-trivial applications in Global Cobol. It is not a Global Cobol programming standard, but, as the title implies, consists of notes based on practical program development experience.

Compiler Options
In order to compile and run efficiently on as many configurations as possible start your programs with:

```
       OPT ED                                    * FORCE EVEN DATA ALIGNMENT
```

This option need not be used if you do not intend the programs to run on a machine such as the DEC PDP-11 where even data alignment can improve performance.

Layout of the Data Division
If you are using option ED you should be careful that the first initialised data item you code starts on an even byte boundary. The easiest way to ensure this is to code the file definitions and map definitions together at the very beginning of the data division, because these blocks, which are initialised, automatically begin on the correct boundary. As a side-effect, the information they contain is least likely to become inadvertently corrupted. Note that the first two bytes of an FD or MD contain a pointer to an access method or system routine and if this is overwritten unpredictable errors will occur when a file or map processing statement using the FD or MD is attempted.

Note that if you code FDs together as we suggest it becomes very easy to introduce a catalogue later, should this prove desirable.

If a program contains large tables which do not require initialisation you can reduce the size of the resulting program file, and therefore the time taken to load the program, by defining the tables at the start of the data division in front of any FD or MD statements or variables which are initialised by VALUE clauses. This advice, of course, contradicts the previous suggestion about coding FDs and MDs first. For it to be worthwhile the tables in question should be large both in absolute terms (a thousand bytes at least) and relative to the size of the program which uses them.

In general you should always initialise the very first byte of any subroutines you develop, because this allows the linker to combine them into one program record which can be loaded by a single read operation. The only exception is when a routine contains a very large data area, typically 4K bytes or more in length, when it may be worthwhile placing the table at the front and not initialising it since, although an additional program file record will be produced requiring an extra read, there will be significantly less data to load. In addition the direct access storage requirements of the compilation file and any program files which use it will be reduced by the size of the non-initialised data area.

Use of Copy Books

Copy books are mainly employed to hold record definitions. These are stored as collections of subordinate (i.e. level 02-49) data items, possibly with VALUE clauses. Because the compiler ignores VALUE clauses in copy books in the linkage section, such copy books can equally well be included in either working storage or the linkage section. When a copy book only contains subordinate items, its COPY statement must be preceded by a level 01 data declaration. This allows you to redefine the record area, should you so wish. For example:

```
01 SR REDEFINES RECORD-AREA
COPY SR
```

It is often necessary to include several copies of a particular record layout in a single program. Therefore it is good practice to make the first two characters of each data name a parameter, written &&, so that different values can be substituted when it is copied, allowing the book to be included in the program a number of times. For example, in book SR the definition:

```
03 SRTOT PIC 9(9) COMP
```

should be coded as:

```
03 &&TOT PIC 9(9) COMP
```

Normally the book would be copied by a:

```
COPY SR
```

statement, but another version could be included by a:

```
COPY SR SUBSTITUTING "S2"
```

which would then contain data names starting "S2", for example S2TOT. In this way you avoid the necessity to hold multiple copies of a record layout in your copy library.

Size of the Data Division
To calculate the size of the data division you add together the space occupied by all working storage data declarations and file definitions and add two bytes for each level 01 or level 77 item coded in the linkage section, which is not itself the redefinition of a previous item. You must also add space for a high-values/low-values pool if you have used the figurative constants HIGH-VALUES and LOW-VALUES anywhere in your procedure division. The size of the pool is the sum of the length in bytes of the largest variable tested or set to HIGH-VALUES, plus that of the largest variable tested or set to LOW-VALUES.

The size of each data declaration is defined in section 3.3. The size of each FD is specified in the part of the chapter devoted to the relevant file organisation which describes the file definition. The size of a map definition is defined in the Screen Presentation Manual.

Size of the Procedure Division
Each Global Cobol statement appearing in the procedure division generates a number of instructions to be executed by the Global Cobol interpreter. Usually one or two instructions are generated per operand: two if the operand is indexed; one otherwise. Statements with no operands generate a single 4-byte instruction, apart from END and

ENDPROG which generate no instructions and occupy no storage. A paragraph name occupies no storage.

Some statements have an **internal statement overhead** of one or more instructions in addition to the overhead induced by operands. Examining a number of different programs we have found that the code generated per procedure division statement (excluding comments and directives) averaged about 8 bytes. So a good "rule of thumb" for estimating the size of any procedure division code you create seems to be:

```
size in bytes = number of statements * 8
```

(Remember that when estimating program size you must take into account not only the instructions you actually code, but the additional space requirements of any routines included in your program from the system library.)

Labels in the Procedure Division
Avoid unnecessary paragraph names in the procedure division. Although paragraph names cause no extra code to be generated they are still extra symbols to be processed by the compiler and, as such, occupy valuable table space, increase the time the compiler spends searching its name table, and generally make the compilation process slower than it need be. If you use structured programming few, if any, paragraph names will be required.

Linkage Section Processing Considerations
If an operand is located in working storage or an external section a single 4-byte instruction is usually generated to process it. If it is located in the linkage section a 6-byte instruction is needed. The difference in execution times of the 4-byte and 6-byte instructions is negligible. However, since a character to character MOVE is a very efficient instruction in Global Cobol, if a small linkage section group is very frequently accessed and storage is critical it will pay you to move it to working storage, process it, then return it to the linkage section. Of course you may be able to avoid linkage section processing altogether in some cases by passing information using the common/external section mechanism. There is no extra overhead associated with processing items in common and external sections.

Use of Literals and Figurative Constants
Use the language word SPACE (or SPACES) wherever appropriate because it makes the program easier to follow and in many cases generates less code because only a 2-byte instruction is needed when SPACE appears as an operand.

HIGH-VALUES and LOW-VALUES also make a program easier to understand, but these should be employed with caution and only used to test or initialise fairly small variables since the compiler actually sets up a high-values/low-values pool at the end of the data division containing a string of #FF bytes and #00 bytes equal in size, respectively to the largest variable used with HIGH-VALUES, and the largest one used with LOW-VALUES. Thus if, for example, your program contained:

```
MOVE HIGH-VALUES TO Y                * Y PIC X(500)
MOVE LOW-VALUES TO Z                 * Z PIC X(800)
```

then your data division would be expanded with a high-values/low-values pool at least 1300 bytes in length. The correct way, of course, to initialise Y (or Z) in this example is to use an overlapping, "ripple" move. Subdivide Y thus:

```
02    Y
  03  Y1    PIC X
  03  Y2    PIC X(499)
```

and then code:

```
MOVE HIGH-VALUES TO Y1              * ONLY 1 BYTE IN POOL
MOVE Y TO Y2                        * INITIALISE REST OF Y
```

Handling of literals in Global Cobol is particularly efficient, with the code specially optimised for single-character literals and integers between -128 and 127 inclusive. You only save space by using an initialised variable in working storage in place of a literal if it is three bytes or more in length **and** is used more than once in the program.

Indexing Considerations
Two instructions are generated to process an indexed elementary item, but an additional 2-byte instruction is needed if the indexing applies to a field within a repeating group. The two basic indexing instructions are 4 or 6 bytes long according to whether or not the simple variables which make up the indexed variable are located in working storage or the linkage section. The extra one or two instructions required to handle an indexed variable are very efficient providing **scaling** is not required and therefore indexing does not usually incur an appreciable overhead. However, if the same indexed item is referenced frequently, consider moving it to working storage, processing it, then returning it.

Scaling
Scaling of fixed point numbers is costly in terms of execution time although, since most scaling is performed internally by the interpreter, extra instructions are not usually required. Scaling takes place:

- when indexing with a non-integer value;
- when variables with different numbers of digits after the decimal point are used in ADD, SUBTRACT or MOVE;
- when non-integer values are used in MULTIPLY or DIVIDE statements.

Avoid scaling wherever possible, particularly in indexing.

The Arithmetic Statements
Wherever possible use the two-operand form of the arithmetic statements rather than the three-operand form since less instructions are usually generated. For example, code:

```
ADD A TO B
```

rather than:

```
      ADD A TO B GIVING B
```

The two-operand form of SUBTRACT generates an additional instruction compared with a two-operand ADD. Therefore, to decrement by a constant (3, say) code:

```
      ADD -3 TO B
```

rather than:

```
      SUBTRACT 3 FROM B
```

On machines without hardware multiply the execution time of a MULTIPLY statement, which can be quite high, depends on the magnitude of the second operand. Thus always code a MULTIPLY statement so that the **smaller** operand comes **second**. For example:

```
      MULTIPLY A BY -9 GIVING B
```

rather than:

```
      MULTIPLY -9 BY A GIVING B
```

On machines without hardware divide, DIVIDE is usually the most costly instruction of all in terms of execution time. It can typically take fifty times as long as an ADD. It can expand up to four instructions in addition to those required for its operands. Try, therefore, to avoid DIVIDE wherever possible.

The MOVE Statement
Character to character or computational to computational moves without scaling are very efficient. Indeed on many machines a small number of characters (say up to 60) can be moved in about the same amount of time that it takes for an ADD.

Display numeric to computational moves are slow, and the reverse process is slower, so try to avoid them as much as possible.

Transfer of Control Statements
The transfer of control statements are all very efficient. If you can, use a GO TO DEPENDING ON construct rather than an IF statement when there are three or more conditions involved. The resulting code will be more compact and faster.

The CALL/ENTRY mechanism used to transfer control to entry points in other compilations is efficient, but the number of instructions generated by the CALL depends on the number of operands in the USING clause, if any.

The EXIT and STOP RUN statements both generate a single short instruction. It is preferable to code EXIT or STOP RUN inline rather than GO TO an EXIT or STOP RUN statement. For example:

```
      IF A ZERO EXIT
```

rather than:

```
      IF A ZERO GO TO AA990
      ...............
      ...............
      ...............
      AA990.
      EXIT
```

The second example not only introduces an unnecessary label and GO TO instruction but is more difficult to understand.

Conditional and Iterative Statements
A format 2 conditional generates one less instruction than the equivalent format 1 conditional. Thus, whenever possible, write IF and ON statements on a single line. For example, code:

```
      IF A ZERO EXIT
```

rather than:

```
      IF A ZERO
            EXIT
      END
```

The most efficient condition to test is whether a computational variable is POSITIVE, NEGATIVE, ZERO or NOT POSITIVE, NOT NEGATIVE or NOT ZERO. Therefore, when a flag with up to three settings is required define it as a PIC S9 COMP variable and use the values -1, 0 and +1 as the different settings.

Make full use of IF/ELSE/END and DO/ENDDO to develop structured programs with a minimum of GO TO statements and labels.

Statements for Console Input/Output
The teletype-compatible console I/O statements generate less code than the DISPLAY...LINE and ACCEPT...LINE statements used for formatted display support. Furthermore the presence of CLEAR, DISPLAY...LINE, ACCEPT...LINE or SCROLL statements causes about 2 Kbytes of subroutines from the system library to be included in the program. In consequence, you should only use formatted displays in programs which are not storage critical.

A teletype-compatible DISPLAY statement which operates on a character variable, character literal or display numeric variable generates only a single instruction, whereas a DISPLAY of a computational item expands to three instructions. In practice character DISPLAYs occur much more frequently than computational displays, so DISPLAY is, on average, implemented very efficiently. This is important, since the character DISPLAY is often the most highly used statement in a program.

The NULL clause on an ACCEPT statement is handled very efficiently by Global Cobol. Therefore, wherever possible, employ the null string as an input option indicating some special processing condition such as end of data, request for a new service, or so on.

Use the BELL statement before displaying input error messages or terminal error messages which are followed by a STOP RUN.

When using formatted DISPLAY...LINE statements to create a menu you can save a small amount of code by producing the output column by column and therefore avoiding the need for COL clauses in some of the DISPLAY...LINE statements. This also makes it easier to alter the column positioning later, should this become necessary, since less statements require their COL clauses to be changed.

Table Handling Statements
The table handling statements, SEARCH and SCAN, generate only a linkage to a machine code system service routine. The statements are therefore extremely efficient and you should always use them in preference to coding a table lookup in Global Cobol. Indeed, when designing an application, always try to develop table structures which make full use of SEARCH or SCAN.

Program Management Statements
The CHAIN, RUN, LOAD and EXEC statements generate only a linkage to the loader. Their execution time depends, of course, on the size of the program file being loaded.

The EDIT Statement
The EDIT statement generates sixteen bytes, and also causes a 700 byte system routine to be linked into the program.

Sort Statements
The SORT, RELEASE and RETURN statements expand a linkage to a 4K-byte service routine which controls the sort. This routine employs the unoccupied part of the user area as a work area, and the amount of space available determines the maximum number of records capable of being sorted. If you invoke the sort from an overlay structure, you may need to use the FREE$ routine described in Chapter 5 of the Global Development System Subroutines Manual to ensure that it obtains the maximum amount of free space actually available.

Multi-User Statements
SUSPEND generates 8 bytes when the seconds operand is coded, or 4 bytes otherwise. LOCK and UNLOCK generate 16 bytes each, and cause a 400 byte routine to be linked into the program.

File Processing Statements
The file processing statements; OPEN OLD, OPEN NEW, WRITE NEXT, WRITE, REWRITE, READ NEXT, READ and CLOSE, generate only a call to an access method routine included in the program containing the FD when it is linkage edited. The compiler sets the first two bytes of the FD to address the entry point of the access method routine and the file processing statements simply pass control to the routine using an internally generated:

```
CALL pointer USING FD operation-type record-area
```

System Routines
The CALL statement provides an efficient linkage to the system routines. You should note that the COPY$ routine employs the unoccupied part of the user area at the top of the Global Cobol memory region as a temporary buffer area, and the amount of space actually available determines the efficiency of the copy operation. If you invoke COPY$ from an overlay structure you may need to use the FREE$ routine described in Chapter 5 of the Global Development System

Subroutines Manual to ensure that the routine obtains the maximum amount of free space actually available.

Modular Programming Considerations

The SECTION statement should be used to subdivide the procedure division into a number of concise, meaningful routines, each of which, typically, would occupy only one or two pages of the listing. A PAGE directive is usually coded immediately before each SECTION statement to cause each section to begin on a new page. The overhead for each section is simply a 6-byte instruction, expanded by the SECTION statement, which Global Cobol uses to provide the control path trace which appears on the diagnostic report following a program error. Even this overhead can be eliminated, at the expense of suppressing information which might prove useful in debugging, by coding OPT NTR at the very start of your program, as explained in Appendix B.

Full use should be made of the linkage editor to construct complex programs from a number of simpler compilations. It is good programming practice to limit the size of individual compilations to about ten pages of listing. The CALL and ENTRY statements used in passing control between different compilations have been optimised to be particularly efficient, as has the handling of the linkage section.

Overlay Programming Techniques

For particularly large programs you may need to adopt an overlay scheme. In the simplest type a single **root** module invokes a separately linked **overlay** module by means of the Global Cobol LOAD or EXEC statement. At any instant the root will be present in storage together with the **current** overlay module. When the processing requirement changes a new overlay will be LOADed or EXECed from the root and will occupy the space used by its predecessor.

Usually the access method routines, and possibly other subroutines, will be employed by the root and most overlay programs. In this case you must ensure that the routines are included in the root, and that these copies are used by the overlays which need them. If you have them linkage edited into every overlay you will waste storage because they will be present twice: once in the root and once in the current overlay. In addition the program files for the overlays will be larger than they need be because they include the access routines unnecessarily.

You can use the linker, $LINK, to create so-called dependent programs which reference modules in other overlays known to be resident when they, the dependents, are loaded. This feature enables you to create programs which use access methods, system routines or application subroutines from a previously loaded, separate root program.

You may wish to include an access method or subroutine in a root program which does not access that routine. This is best done by including a statement of the form:

```
GLOBAL routine-name
```

at the start of the DATA DIVISION of the root program. The GLOBAL statement is described in detail in chapter 8 of this manual.

If the root program shares common data, such as master file definitions, with its overlays then this data is best handled using common and external sections. The data in question should be defined in a copy book to ensure consistency. The root will then define the common section by, for example, code such as:

```
COMMON SECTION SAMF
COPY MF
```

and each overlay which references it will do so by including the corresponding external section:

```
EXTERNAL SECTION SAMF
COPY MF
```

This technique reduces the number of parameters that require to be passed in CALL statements and at the same time avoids the longer instructions required for linkage section access.

Appendix E – Recommended Naming Conventions

ELEMENT	NAME	FORMAT
pp	product code	two alphabetic characters
nnn	program number or paragraph number	three digits
m...m	mnemonic	up to the indicated number of alphabetic characters
ss	section code	two alphabetic characters
rr	record code	two alphabetic characters
ffff	field code	alphabetic mnemonic of any length (first four characters significant)

SYMBOL USED AS ...	FORMAT
program name program-id	ppnnn
common section name external section name	ppmmmm
entry name filename	mmmmmm
section name	ss-description
paragraph name	ssnnn
copy book name 01 data name	rr
02 data name	rrffff
77 data name accumulator count switch constant table other	 A-ffff C-ffff S-ffff K-ffff T-ffff Z-ffff
volume-id data file-id	ppmmmm ppmmmmmm

Figure E.1 - Recommended Naming Conventions

If you require to compile programs efficiently on a wide range of different configurations, some of which may have rather limited main storage, you should optimise the compiler's use of table space by not using the "long names" option. This requires that the first six characters of each symbol used by a program be unique. The naming conventions discussed below and summarised in Figure E.1 will prevent naming conflicts between development teams.

Product Codes
Each development team is allocated a unique two character product code, pp. For example, SA might be the product code allocated to the team producing a sales ledger application.

Every volume-id or file-id identifying a volume or file produced by the development team starts with the product code, and this prevents file names used by one project conflicting with those of another.

Program Names and Program-ids
Every program the development team produces is assigned a three digit program number, nnn, unique to the product. The PROGRAM statement is then of the form:

```
PROGRAM ppnnn
```

For example, SA013 could be the sales statistics print program. When a number of compilations are linkage edited to produce a program file the program-id is chosen to be the same as the program name of the main program.

Common and External Section Names
The names given to common sections and their corresponding external sections should be of the form:

ppmmmm

where mmmm is a m numeric identifying the common section. For example, a common section containing master file definitions for product SA might be named SAMF.

Entry Names
Entry names for subroutines entered via the CALL statement are chosen to be meaningful alphabetic mnemonics up to six characters long. Thus a common validation routine might be given the entry name CHECK and invoked by a CALL of the form:

```
CALL CHECK USING...
```

A central list of the subroutine entry names used by the product should be kept by the team to prevent inadvertent duplication. Note that an entry name may not be the same as a filename (see "Naming Data Files").

Section Names
Each section used in a program is allocated a section code, ss, consisting of two alphabetic characters which reflect the section's position in the control hierarchy. For example, AA is normally used for the main line, BA might be the highest level logic of a subroutine function and CA might be a routine of BA.

Section names are constructed by appending a hyphen and a meaningful description to the section code. For example:

```
AA-MAINLINE

BA-OPEN-ALL-FILES

CA-DELETE-UNWANTED-WORKFILE
```

Paragraph Names
A paragraph name defined within section ss is assigned a name of the form:

```
ssnnn.
```

where nnn is three digits. The names appear on the listing in ascending order and are allocated from ss010. onwards, in steps of 10 for ease of maintenance. For example, the AA-MAINLINE section may contain the paragraphs named AA010. and AA020..

Public Data and Copy Book Names
Each level 01 group describing a record or control block shared by a number of programs is allocated a two alphabetic character record code, rr, unique within the product. The first character of rr should not be Z.

The data name assigned to the level 01 item itself is simply rr. The data name of each subordinate item is of the form:

rrffff

where ffff is a mnemonic qualifier, not all numeric, the first four characters of which render the data name unique within the level 01 item. For example:

```
01    SR                                  * SALES RECORD
  02  SRDSID...                           * SUBORDINATES
  02  FILLER...
  02  SRKEY...
  02  SRCODE...
   03 SRDEPT...
   03 SRQUAL...
```

The subordinate items will be saved as a copy book, and the two-character book name allocated will be the same as the record code, rr. The product copy library should be maintained in alphabetic order of book name so as to serve as an automated central index of allocated record codes.

Private Data Names
Private level 01 structures, local to a single program, are allocated a record code beginning with Z, and the subordinate items within them are prefixed accordingly:

```
01    ZS                                  * STORAGE ELEMENT
  02  ZSFLAG...
  02  ZSTYPE...
  02  ZSLENGTH...
```

Level 77 data names are constructed from a single character indicating the way in which the item is used, followed by a hyphen and an alphabetic field code. The examples show the standard meanings of the first letter:

```
77    A-TOTAL...                          * ACCUMULATOR

77    C-CLIENTS...                        * COUNT

77    S-OVERDRAWN...                      * SWITCH

77    K-PI...                             * CONSTANT

77    T-CODE...                           * TABLE

77    Z-REPLY...                          * OPERATOR REPLY
```

Naming Volumes

Each volume used by a product is assigned a unique volume-id, ppmmmm, where pp is the product code and mmmm is a mnemonic, four characters or less in length, indicating the function of the volume. For example:

SAPROG	a volume containing the programs of the sales ledger suite;
SAMAST	a volume containing the sales ledger master files;
SAWORK	a work volume.

Naming Data Files
Each data file used by a product is assigned a unique alphabetic filename, mmmmmm, six characters or less in length. This name appears as the first operand of its FD and in file processing statements which access the file.

The file-id associated with the file (i.e. the external name by which it is known in the volume's directory) is of the form:

ppmmmmmm

The file-id is prefixed by the product code to prevent conflicts with files developed by other projects. Here is an FD and an OPEN statement for the example sales ledger STATS file:

```
FD STATS ORGANISATION RELATIVE-SEQUENTIAL
ASSIGN TO UNIT "DSK" FILE "SASTATS" VOLUME "SAMAST"
.
.
OPEN OLD STATS
```

A central list of the filenames used by the product should be kept by the team to prevent inadvertent duplication. Note that a filename may not be the same as a subroutine entry name (see "Entry Names").

Naming Program Development Files
The commands described in the Global Cobol User Manual uses file prefixes (each a single letter and full stop, beginning the file-id) to distinguish between the various files required during program construction. Table E.2 shows the file-ids used in developing main program SA013 if the recommended standard naming conventions are adopted. A subroutine, CHECK say, would give rise only to the additional files S.CHECK, B.CHECK and L.CHECK, since it would never be linkage edited in isolation.

FILE-ID	**USED FOR**
S.SA013	the file containing the most up-to-date Global Cobol source of SA013.
B.SA013	the previous version of SA013, as it was before the last edit took place. This version can be used as a backup should S.SA013 be inadvertently destroyed.
S.SA	the copy library used by every program of the sales ledger suite, including SA013.
C.SA013	the compilation file produced by the Global Cobol compiler when S.SA013 is compiled.
L.SA013	the compilation listing produced by the Global Cobol compiler when S.SA013 is compiled.

C.SA	the subroutine library used when linkage editing C.SA013 and every other program of the sales ledger suite.
M.SA013	the map listing produced by the Global Cobol linker when C.SA013 is linkage edited.
SA013	the program file produced by the Global Cobol linker when C.SA013 is linkage edited.
P.SA	the program library which will eventually contain SA013 and every other program of the sales ledger suite.
K.SA	the AutoClerk control file used by the sales ledger suite.

Table E.2 - Example Program Development File Names

Appendix F – Copy Library Formats

A copy library is made up of statement lines rather like a normal Global Cobol program, and is structured as follows:

```
[optional comments]
.BOOK b1
(Statements to be included in the program when the statement
COPY b1 is compiled)
.END
[Optional comments]
.BOOK b2
(Statements to be included in the program when the statement
COPY b2 is compiled)
.END
.
.
.
.
[Optional comments]
.BOOK bn
(Statements to be included in the program when the statement
COPY bn is compiled)
.END
[optional comments]
```

The special statements .BOOK and .END are used to name and delimit the different books which make up the library. The optional comments statements which may be supplied at the beginning or the end of the library, or between the books, can never be copied into a Global Cobol program, and, if coded at all, will simply be used to annotate the listing of the library itself.

The books themselves can contain any Global Cobol statements, including the COPY statement. Although a copied book may itself contain a COPY statement, and the book thus copied may also possess further COPY statements, this is as far as the nesting may go. Thus, at most, two levels of internal nesting are supported.

The quantities b1, b2, ... bn which appear in the .BOOK statements match the two-character book names used in COPY statements. The books may appear in any order of book name, but if two or more are erroneously given the same name only the first one on the file will ever be referenced. The compiler can only process up to three copy libraries per compilation, and each library must contain no more than 100 books.

Each book may contain a single parameter, coded as a string of &s. This parameter may appear as often as required within the book, and will be substituted by the string supplied in the SUBSTITUTING clause of the COPY statement. It is good practice to make all the parameters within a copy book the same length, although this is not strictly necessary as the substituted string will be truncated or padded with hyphens to match the length the parameter string.

```
* SALES LEDGER COPY LIBRARY
*
.BOOK MF
*
* FILE DEFINITION FOR MASTER FILE
FD MASTER ORGANISATION RELATIVE-SEQUENTIAL
ASSIGN TO UNIT "DSK" FILE "SAMASTER" VOLUME "SADATA"
```

```
.END
*
* THE FOLLOWING COPYRIGHT TEXT MUST BE INCLUDED
* IN EVERY PROGRAM
*
.BOOK CR
01    FILLER
  02  FILLER      PIC X(26)
                  VALUE "COPYRIGHT-KLUDGE-KORP-1988"
.END
*
* ALL PROGRAMS OF THE MASTER FILE
* PROCESSING SUITE SHOULD BEGIN WITH A
* COPY MP STATEMENT FOLLOWING THE PROGRAM STATEMENT
*
.BOOK MP
*
* MASTER FILE PROCESSING SUITE
*
DATA DIVISION
COPY CR
*
COPY MF
*
.END
*
.BOOK ST
*
* SALES TRANSACTION RECORD
*
  02  &&CUST      PIC X(8)                * CUSTOMER CODE
  02  &&AMT       PIC S9(6) COMP          * AMOUNT IN POUNDS
  02  &&TYPE      PIC X                   * CUSTOMER TYPE
(etc, etc, etc)
.END
.
.
```

Figure F - An Example Copy Library

If a copy book itself copies a book containing substitution parameters, then the substitution string used for the inner copy book is that specified on its COPY statement, or if the substituting clause is omitted, the substitution string for the outer copy book is used.

Figure F shows part of a copy library which might be used by the programs of a Sales Ledger system. A typical program of the master file suite might begin:

```
      PROGRAM SA003
      COPY MP
      *
      01 ST
      COPY ST
```

The result would be that the following statements would be included in the compilation:

```
      PROGRAM SA003
      *
      * MASTER FILE PROCESSING SUITE
      *
      DATA DIVISION
      01    FILLER
        02  FILLER      PIC X(26)
                        VALUE "COPYRIGHT-KLUDGE-KORP-1988"
```

```
*
* FILE DEFINITION FOR MASTER FILE
FD MASTER ORGANISATION RELATIVE-SEQUENTIAL
ASSIGN TO UNIT "DSK" FILE "SAMASTER" VOLUME "SADATA"
*
01    ST
*
* SALES TRANSACTION RECORD
*
  02  STCUST      PIC X(8)          * CUSTOMER CODE
  02  STAMT       PIC S9(6) COMP    * AMOUNT IN POUNDS
  02  STTYPE      PIC X             * CUSTOMER TYPE
(etc., etc., etc.)
```

You will note in the example library that the book names have not been arranged in alphabetical order. Although this is legitimate as far as Global Cobol is concerned, we would not recommend it in practice. Unless you maintain the books in a sensible order, as the library grows it will become more and more difficult to find them when they require updating, and the chance of creating a duplicate name by mistake will increase.

Note that the first two characters of all the data names in book ST are coded as a parameter, allowing the book to be copied several times in one program, substituting different record identifier codes. This technique means that multiple copies of record layouts are not required in the copy library, avoiding problems caused by duplicate copies not being updated correctly.

Appendix G – Compilation Listing Error and Warning Messages

This appendix describes the error and warning messages produced by the Global Cobol compiler, $COBOL, on its listing file. **Errors** are in general serious faults, and result in an unusable compilation file. The compiler is able to recover following a **warning** however, and the description of the warning message specifies the recovery action taken. Compilation files subject to warnings only are executable, but it is of course good practice to correct the source so that clean compilations can be obtained.

Each message consists of the error or warning number of up to 3 digits and a short explanation. This is preceded by one or three asterisks which conveniently allow the user to search for error or warning lines within a listing file using the $INSPECT command.

Messages associated with a specific character position within the faulty line are preceded by a positioning line containing a single up-arrow character, located under the offending character or symbol. It should be noted that the up-arrow character is displayed on some printers or terminals as a circumflex, or a logical NOT symbol. Messages not associated with a specific character position within a line are not preceded by the positioning line.

The notes below are referred to, when appropriate, from the message descriptions.

Note 1: This error message is included for completeness only. It should not occur if the compiler is functioning correctly.

Note 2: If an attempt is made to linkage edit and execute the program following this error, the result will be unpredictable.

***** ERROR 1 - PROGRAM STATEMENT MISSING**
The first recognised statement of a Global Cobol program is not a PROGRAM statement. Only comments, PAGE, OPT or COPY directives may precede the PROGRAM statement. See note 2.

*** WARNING 2 - 'PROC. DIV.' ALREADY FOUND**
A PROCEDURE DIVISION statement has been found after the beginning of the procedure division. The PROCEDURE DIVISION statement should delimit the data division and begin the procedure division. The statement is ignored.

*** WARNING 3 - 'DIVISION' EXPECTED**
The language word DIVISION cannot be detected in a PROCEDURE DIVISION or DATA DIVISION statement. DIVISION must follow PROCEDURE or DATA. The statement is assumed to be a DATA DIVISION or PROCEDURE DIVISION statement as appropriate.

*** WARNING 4 - 'LINKAGE SECTION' MISPLACED**
A LINKAGE SECTION statement has been found after the beginning of the procedure division. If present, the LINKAGE SECTION statement should occur within the data division. The statement is ignored.

*** WARNING 5 - 'LINKAGE SECTION' ALREADY FOUND**
A LINKAGE SECTION statement has already been found. If present, LINKAGE SECTION should occur only once within the data division. The statement is ignored.

*** WARNING 6 - 'SECTION' MISSING**
The language word SECTION cannot be detected in a LINKAGE SECTION statement. SECTION must immediately follow LINKAGE. The statement is assumed to be a LINKAGE SECTION statement.

***** ERROR 7 - MULTIPLE LABELS NOT ALLOWED**
Only one paragraph name may be coded on a line. A paragraph name can be coded on a line by itself, or can be followed by a comment or a Global Cobol statement.

***** ERROR 8 - PROGRAM NOT EXECUTABLE**
This error only appears at the end of the source listing. It normally appears as a consequence of a previously reported error. It indicates that a very serious error condition has arisen, and that a compilation file will not be created.

*** WARNING 9 - LABEL NOT ALLOWED**
A paragraph name has been found labelling a non-procedural statement. Paragraph names must be coded in the procedure division only.

*** WARNING 10 - DATA FOUND IN PROC. DIV.**
A data division statement has been found within the procedure division. Space has been reserved at the end of the data division for this declaration, but it will not have been initialised and any VALUE statements will be ignored.

*** WARNING 11 - PROC. DIV. STATEMENT ASSUMED**
A procedure division statement has been found before the incidence of the PROCEDURE DIVISION statement. PROCEDURE DIVISION should delimit the data division and begin the procedure division. A PROCEDURE DIVISION statement is assumed to have immediately preceded the statement, i.e. the procedure division is assumed to have started.

***** ERROR 12 - INVALID PROGRAM NAME**
The item following the language word PROGRAM is not a valid Global Cobol symbol. A symbol must begin with a letter (A to Z or $) and be followed by zero or more alphanumeric characters (A to Z, $, 0 to 9 or -). See note 2.

***** ERROR 13 - PROGRAM NAME DEFINED TWICE**
An attempt has been made to use the program name in another symbol declaration elsewhere in the program. Each user defined symbol must have only one definition.

*** WARNING 14 - UNEXPECTED END OF FILE**
An ENDPROG statement has not been detected at the end of the program. The last statement in a program must be the ENDPROG statement. An ENDPROG statement is assumed.

***** ERROR 15 - STATEMENT NOT RECOGNISED**
A line which does not begin with a paragraph name must start either with the first word of a Global Cobol statement, or with an asterisk

to denote a comment. The word or asterisk may be preceded by spaces, tabs or both.

If a line begins with a paragraph name then the remainder of the line must either be a blank, or start with the first word of a Global Cobol statement, or an asterisk to denote a comment. The word or asterisk must be separated from the preceding paragraph name by one or more spaces or tabs.

***** ERROR 16 - TOO MANY GLOBAL VARIABLES**
The total number of global symbols defined or referenced in a single compilation must not exceed 127. Global symbols include the program name, entry points defined within the program, and externals such as file access methods, system subroutines and entry points in other user compilation units which are CALLed from this program. See note 2.

***** ERROR 17 - COMPILATION EXCEEDS 32K**
The total size of the compilation exceeds 32767 bytes. Notes on how much code is generated per statement can be found in Appendix D. The size of the compilation unit must be reduced, possibly by dividing it into two or more modules.

*** WARNING 18 - 'DATA DIVISION' EXPECTED**
A DATA DIVISION statement has not been detected after the PROGRAM statement. Ignoring any comments, PAGE, OPT or COPY statements, DATA DIVISION should be the second statement of the program. A DATA DIVISION statement is assumed.

***** ERROR 19 - PROGRAM STATEMENT MISPLACED**
A PROGRAM statement has been detected out of context. Ignoring any comments, PAGE, OPT or COPY directives, the PROGRAM statement should be the first of the program.

***** ERROR 20 - 'DATA DIVISION' MISPLACED**
A DATA DIVISION statement has been detected out of context. Ignoring any comments, PAGE, OPT or COPY directives, DATA DIVISION should be the second statement of the program.

*** WARNING 21 - ILLEGAL GLOBAL DECLARATION**
An item declared as a global by means of a GLOBAL statement coded at the start of the data division has been mistakenly defined in the linkage section. This warning (which only appears on the compilation listing) will be output immediately before the PROGRAM statement itself is listed, and will appear once for every global statement affected.

***** ERROR 22 - END OF FILE EXPECTED**
There are non-blank lines following the ENDPROG statement. These lines have not been processed.

***** ERROR 23 - MUST PRECEED LINKAGE SECTION**
A COMMON SECTION or EXTERNAL SECTION statement has been detected following the linkage section. The correct order is common sections (if any), then external sections (if any) then the linkage section.

***** ERROR 24 - 'PROGRAM' ALREADY FOUND**
A program should only contain one PROGRAM statement.

***** ERROR 25 - 'DATA DIVISION' ALREADY FOUND**
A second DATA DIVISION statement has been found, and ignored.

***** ERROR 26 - "PROC. DIV' ALREADY FOUND**
A second PROCEDURE DIVISION statement has been found, and ignored.

***** ERROR 27 - 'SECTION' EXPECTED**
The language word SECTION was not found in a COMMON SECTION or EXTERNAL SECTION statement.

***** ERROR 31 - 'NAME' EXPECTED**
The language word NAME is missing or misplaced in an INDEX statement. It should be coded after INDEX.

***** ERROR 32 - 'ERROR' OR 'FAILURE' EXPECTED**
The language words ERROR or FAILURE cannot be found after the word ON in an FD declaration.

***** ERROR 33 - 'ADDRESS' OR 'LENGTH' EXPECTED**
The language words ADDRESS or LENGTH cannot be found after the word RECORD in an FD declaration.

*** WARNING 34 - INDEX NAME ALREADY FOUND**
More than one INDEX NAME IS statement has been detected in a file declaration. Only one such statement should appear; the first will be compiled, the rest ignored.

*** WARNING 35 - RECORD ADDRESS ALREADY FOUND**
More than one RECORD ADDRESS IS statement has been detected in a file declaration. Only one such statement should appear, the first will be compiled, the rest ignored.

*** WARNING 36 - ON FAILURE ALREADY FOUND**
More than one ON FAILURE statement has been detected in a file declaration. Only one such statement should appear; the first will be compiled, the rest ignored.

***** ERROR 37 - 'INDEX NAME' ONLY FOR DMAM**
An INDEX NAME IS statement has been detected in a file declaration that is not for data management files. This is not allowed.

***** ERROR 38 - 'RECORD ADDRESS' ONLY FOR DMAM**
A RECORD ADDRESS IS statement has been detected in a file declaration that is not for data management files. This is not allowed.

***** ERROR 39 - 'ON FAILURE' ONLY FOR DMAM**
An ON FAILURE statement has been detected in a file declaration that is not for data management files. This is not allowed.

***** ERROR 40 - 'BLOCK' INVALID FOR DMAM FILES**
A BLOCK CONTAINS statement has been detected in a data management file definition. This is not allowed.

***** ERROR 41 - CHARACTER NOT RECOGNISED**
The specified character is an ASCII control character, and is not permitted in a Global Cobol source program. Reference should be made to the ASCII character set in Appendix A.

***** ERROR 42 - ITEM NOT RECOGNISED**
The specified item is not a legal Global Cobol language word, symbol, numeric string or character string. Each item must be delimited by space, tab, or in certain cases by a bracket or comma. Other characters are not allowed.

*** WARNING 44 - ITEM HAS INVALID DELIMITER**
The specified item has been recognised as a legal Global Cobol language word, symbol, numeric string or character string but is not followed by a valid delimiter. Each item must be delimited by space, tab, or in certain cases by a bracket or comma. Other characters are not allowed. A valid delimiter has been assumed preceding the invalid one, and the invalid delimiter now starts the following item.

***** ERROR 46 - INVALID NUMERIC STRING**
A valid numeric string consists of optional leading blanks, an optional sign, 1 - 15 digits (which may be omitted if the decimal point is present), an optional decimal point which, if present, must be followed by 1 - 7 digits, and optional trailing blanks. The total number of digits must not exceed 18.

*** WARNING 47 - STRING NOT TERMINATED BY "**
The specified character string is not terminated by a quote character. The string will be processed as if a quote character had been inserted immediately before the end of line character.

*** WARNING 48 - NON-PRINTABLE CHAR IN STRING**
The specified character is an ASCII control character. It should be noted that unpredictable results will occur if the string is subsequently DISPLAY'ed or printed.

*** WARNING 49 - ODD NUMBER OF HEX CHARS**
A hexadecimal literal has been discovered containing an odd number of hexadecimal characters. A leading zero has been inserted. Hexadecimal literals must contain an even number of hexadecimal characters.

*** WARNING 50 - ITEM TOO LONG**
The specified item contains more than 64 characters. Only the first 64 characters will be recognised, and the remaining characters ignored. The compiler will not accept Global Cobol symbols or character strings greater than 64 characters long. Note that in the case of a symbol only the first 6 or 31 characters are significant, depending on whether or not the "long names" option is in force.

***** ERROR 51 - NUMERIC STRING TOO LARGE**
A numeric string has been discovered with too many digits. There may be no more than 15 digits preceding the decimal point, nor more than 7 following the decimal point. In addition the total number of digits must not exceed 18.

*** WARNING 52 - EXTRA CHARS ON END OF LINE**
The specified Global Cobol statement is not followed by end of line or a legal comment. All comments must begin with an asterisk. Characters following the statement are ignored up to the end of the line.

***** ERROR 53 - ILLEGAL COPY STATEMENT**

The item following the language word COPY is not a valid Global Cobol copy book name. A copy book name must be a one or two character Global Cobol symbol.

***** ERROR 54 - COPY NESTING LEVEL EXCEEDED**
A COPY statement has been discovered which would exceed the maximum copy nesting level.

*** WARNING 55 - EXTRA CHARS ON END OF LINE**
The specified PAGE or COPY statement is not followed by end of line or a legal comment. All comments must begin with an asterisk. The characters following the statement are ignored up to the end of the line.

***** ERROR 56 - COPY BOOK NOT FOUND**
The required copy book is not present on any of the copy libraries submitted with this compilation.

***** ERROR 57 - NO COPY LIBRARY**
A COPY statement has been detected, but no copy library was specified at the beginning of the compilation.

*** WARNING 58 - COPY BOOK NAME TRUNCATED**
A COPY statement containing a book name longer than 2 characters has been detected. Only the first two characters of a book name are significant.

***** ERROR 59 - STRING EXPECTED**
No character string was found following the SUBSTITUTING clause of a COPY statement.

*** WARNING 60 - ITEM TOO LONG**
The string specified in the SUBSTITUTING clause of a COPY statement contains more than 31 characters. Only the first 31 characters of the string will be used during substitution.

***** ERROR 61 - ATTEMPTED TO CHANGE VAR. TYPE**
The compiler symbol table has been corrupted. See note 1.

***** ERROR 62 - PROGRAM NOT EXECUTABLE**
This message usually is a result of a preceding ERROR 201. It means that the compiler cannot calculate the addresses of paragraph and section names. If this is the only error in a compilation then see note 1. The error indicates that a very serious error condition has arisen, and that a compilation file will not be created.

***** ERROR 63 - >20 VARIABLES ON A LINE**
The compiler's symbol buffer has overflowed. See note 1.

*** WARNING 70 - SYMBOLIC DEBUG RECORD OVERFLOW**
The compilation unit contains more than 1300 named symbols. The symbolic debug record generated for this unit will only contain the first 650 symbols referenced in the compilation.

***** ERROR 72 - UNRECOGNISED OPTION**
One of the options specified in the OPT statement is not a member of the valid options set. Options specified before this on the line will be effective. Options specified after on the same line will not.

*** WARNING 73 - TOO MANY PROPER L.S GROUPS**
The compilation contains more than 100 proper linkage section groups. (Each level 01 or level 77 item coded in the linkage section, which is not itself a redefinition, counts towards the total of 100.) The compiler is unable to generate symbolic debug information in this case, but, apart from this, the compilation is unaffected.

***** ERROR 98 - CODE TABLE OVERFLOW**
The capacity of the compiler's code generation table has been exceeded. See note 1.

***** ERROR 99 - UNRECOGNISED TOKEN MPR-MPS**
The compiler's control module has passed a line of source to the wrong processing module. See note 1.

***** ERROR 101 - LINE OUT OF CONTEXT**
A Global Cobol data division statement has been detected out of context. A VALUE statement must be preceded by a valid elementary or 77 level item declaration. A valid 01 level declaration must be the first statement of a group declaration. Valid FD and ASSIGN statements must be the first two statements of a file description. A valid MD statement must be the first statement of a map definition.

***** ERROR 103 - ILLEGAL ORGANISATION**
The specified organisation has not previously been defined in an ORGANISATION statement. All ORGANISATION statements must be coded immediately following the DATA DIVISION statement.

***** ERROR 104 - ASSIGN STATEMENT NOT FOUND**
No ASSIGN statement has been detected in a file definition. An ASSIGN statement should follow an FD statement in every working storage file definition, although this is optional in the linkage section.

*** WARNING 105 - 'ON ERROR' ALREADY FOUND**
More than one ON ERROR statement has been detected in a file definition. The ON ERROR statement may appear at most once in each file definition. Further ON ERROR statements will be ignored.

*** WARNING 106 - KEY STATEMENT ALREADY FOUND**
More than one KEY statement has been detected in a file definition. The KEY statement may appear at most once in each file definition. Further KEY statements will be ignored.

*** WARNING 107 - OPTION IGNORE NOT ALLOWED**
The OPTION IGNORE statement is only allowed in indexed sequential file definitions.

*** WARNING 108 - RECORD STATEMENT ALREADY FOUND**
More than one RECORD statement has been detected in a file definition. The RECORD statement may appear at most once in each file definition. Further RECORD statements will be ignored.

*** WARNING 111 - OPTION RESET NOT ALLOWED**
An OPTION RESET statement has been detected in an indexed sequential file definition.

*** WARNING 113 - SIZE STATEMENT ALREADY FOUND**

More than one SIZE statement has been detected in a file definition. The SIZE statement may appear at most once in each file definition. Further SIZE statements will be ignored.

***** ERROR 114 - ENTRYNAME NOT ALLOWED**
The symbol parameter in an AREA or ON ERROR statement cannot be an entryname.

*** WARNING 115 - LENGTH MUST BE VARIABLE**
An incorrect RECORD LENGTH statement has been detected in an indexed-sequential file definition. The length may not be coded as a literal. The correct form is RECORD LENGTH IS symbol. The statement has been ignored.

*** WARNING 116 - KEY FIELD NOT ALLOWED**
An incorrect KEY LENGTH statement has been detected in an indexed-sequential file definition. The correct form is KEY LENGTH IS keylength. The statement has been ignored.

*** WARNING 117 - KEY MUST BE SYMBOL**
An incorrect KEY statement has been detected in a file definition. The correct form is KEY IS symbol. The statement has been ignored.

***** ERROR 119 - ITEM EXCEEDS 32K**
The declaration of the current FD or data item will cause the total compilation size to exceed 32767 bytes. This is the maximum size of a single compilation unit permitted by the compiler. The declaration has been ignored.

*** WARNING 120 - VALUE TOO LONG**
The number of bytes specified in this and any earlier associated VALUE statements has exceeded the length of the previous elementary or 77 level item. For a character item the current value will be truncated to the right and processed. Further VALUE statements will be ignored within this item.

***** ERROR 121 - PICTURE AND VALUE DO NOT MATCH**
A data item has been assigned a value that it cannot accept. Check the picture clause definition with the value clause description.

*** WARNING 122 - INCORRECT HEX LENGTH**
The specified hexadecimal string does not have the correct length. When a hexadecimal string is used to initialise a computational item it must establish every byte of the item and no more. The compiler has truncated or zero-filled to the right as appropriate.

***** ERROR 123 - VALUE TOO LARGE**
The specified numeric or character string is too large for the data item being initialised. The PIC clause of the data item should be changed to permit additional leading digits and/or a sign.

***** ERROR 124 - PTR VALUE MUST NOT BE IN L.S**
A pointer data item may not be initialised to address a data item defined in the linkage section.

***** ERROR 125 - SYMBOL NOT DEFINED**
The symbol in a REDEFINES clause, or a VALUE statement for a pointer data item, is not defined in the compilation. Items to be redefined

must be declared in the same compilation as, and earlier than, the REDEFINES clause.A pointer item may be initialised to address a data name, file name, paragraph name, section name or entry name defined in this compilation, or an external name defined in a GLOBAL statement.

***** ERROR 126 - ITEM MAY NOT BE REDEFINED**
An illegal name has been detected in a REDEFINES clause. A data item may only redefine another previously declared data item or filename.

***** ERROR 127 - ITEMS IN DIFFERENT SECTIONS**
A data item in the linkage section may not redefine a data item or file name declared before the LINKAGE SECTION statement. A data item declared before the LINKAGE SECTION statement may not redefine a data item or filename declared in the linkage section.

*** WARNING 129 - VALUE TRUNCATED**
The specified numeric or character string contains too many decimal places for the data item being initialised. The compiler has truncated the string to match the data item's PIC clause.

***** ERROR 130 - TOO MANY GLOBALS**
The total number of global symbols defined or referenced in a single compilation must not exceed 127. See note 2.

***** ERROR 131 - DOUBLY DEFINED VARIABLE**
The specified symbol has been defined more than once in this compilation. The compiler only recognises the first six characters of a user symbol (or 31 if the long names option has been specified). Each unique symbol must only be defined once in the compilation, though it may be referenced many times. Each definition will be flagged with an error.

***** ERROR 132 - SYSTEM VARS CANNOT BE GLOBAL**
A system variable has been specified in a GLOBAL statement. System variables are not permitted to be defined as global.

*** WARNING 133 - INITIALISATION NOT ALLOWED**
No initialisation can be performed for linkage section items, redefinitions, and items within repeating groups. The attempted initialisation is ignored.

*** WARNING 134 - REDEFINITION TOO LARGE**
The declaration of the current data item will cause the length of the preceding 01 level data item to exceed the length of the data item specified in the REDEFINES clause. The compiler will allow the redefinition to continue, although unpredictable results may occur when the program is executed.

***** ERROR 136 - PROGRAM NOT EXECUTABLE**
This error will appear normally as a consequence of other errors in the data division, particularly ERROR 131 and ERROR 126. When it appears on its own, it implies an invalid REDEFINES clause, normally ERROR 126. It indicates that a very serious error condition has arisen, and that a compilation file will not be created.

***** ERROR 137 - REDEFINITION TOO LARGE**
The declaration of the current data item will cause the length of the preceding 01 level data item to exceed 32767 bytes. This is the

maximum size permitted by the compiler. The declaration has been ignored.

Alternatively the declaration of the current data item will cause the length of the preceding 01 level data item to exceed the length of the absolute system variable specified in the REDEFINES clause. This is not allowed.

***** ERROR 139 - ILLEGAL BASE**
The item specified in a BASED clause was not a pointer data item declared in working storage. The BASED clause is ignored.

***** ERROR 140 - 'BASED' ALLOWED IN L.S ONLY**
A BASED clause has been detected on an 01 or 77 level declaration which is coded before the LINKAGE SECTION statement. 01 or 77 level declarations with BASED clauses must be coded in the linkage section.

***** ERROR 141 - ITEM MAY NOT REDEFINE ITSELF**
A statement of the form:

```
     01    A     REDEFINES A
```

is meaningless.

***** ERROR 142 - LEVELS NESTED TOO DEEPLY**
The current group data definition contains more than 19 nested levels.

***** ERROR 143 - NESTED OCCURS NOT ALLOWED**
An OCCURS clause has been found on a data item which is part of a repeating group.

*** WARNING 144 - ODD INITIAL ADDRESS**
The first initialised byte of the program is at an odd address within the compilation. This warning is only generated if the program is compiled with option ED (even data) specified.

*** WARNING 145 - REDEFINITION TOO LARGE**
See WARNING 134 for details

*** WARNING 146 - GROUP OF ZERO LENGTH**
A group has been found which has zero length. Indiscriminate use of such items can lead to unpredictable results.

***** ERROR 147 - VALUE STATEMENT EXPECTED**
A PIC X(?) item must be followed by a VALUE statement.

***** ERROR 148 - ILLEGAL USE OF PIC X(?)**
A PIC X(?) declaration cannot be a redefinition, or have an OCCURS clause, or be part of a repeating group.

***** ERROR 151 - 'TO' EXPECTED**
The language word TO is missing or misplaced in an ASSIGN statement. It should be coded after ASSIGN.

***** ERROR 152 - 'UNIT' EXPECTED**
The language word UNIT is missing or misplaced in an ASSIGN statement. It should be coded after ASSIGN TO.

***** ERROR 153 - BRACKET MISSING**
Either "(" or ")" is required at this position in the line for one of the following reasons:

(a) In a PICTURE clause X or 9 can only be followed by a comment or the character (.

(b) The character (must then be followed by an integer and the character).

***** ERROR 154 - LABEL NOT ALLOWED IN D.D**
Lines in the data division may not be labelled.

***** ERROR 157 - INTEGER EXPECTED**
The statement is flagged where an integer value was expected. For appropriate values, see the notes to ERROR 160.

***** ERROR 158 - 'IS' EXPECTED**
The language word IS is missing or misplaced in a KEY statement. It should be coded after KEY.

***** ERROR 159 - STRING EXPECTED**
In a VALUE or ASSIGN statement a literal string was not found where expected. A literal string begins and ends with a " character.

***** ERROR 160 - VALUE OUTSIDE VALID RANGE**
This results from one of the following rules being broken:

In an OCCURS clause the language word OCCURS must be followed by an integer in the range 1 to 32767;

In a PICTURE clause the length of a character string must be an integer in the range 1 to 32767;

In a PICTURE clause for a numeric item, ie 9(p,q) or S9(p,q), p must be in the range 1 to 15, q in the range 1 to 7 and p + q must be less than 19;

In a KEY LENGTH statement, the key length must be unsigned and in the range 1 to 99;

In a RECORD LENGTH statement, the record length must be an integer in the range 1 to 32767;

In a SIZE statement, the size must be in the range 0 to 999999999;

In an ORGANISATION statement the type operand must be between 0 and 255 inclusive and the extension between 0 and 1016 inclusive;

The integer in a BLOCK CONTAINS statement must be in the range 1 to 32760.

***** ERROR 162 - 'PIC' EXPECTED**
The language word PIC cannot be found in a level 77 data declaration. It should be inserted following the OCCURS clause (if present) or the data name. PIC must be on the same line as the level number. It cannot start a line.

***** ERROR 163 - INVALID PICTURE CLAUSE**
The item type identifier X, 9, S9 or PTR is missing or misplaced in a picture clause. It should be coded immediately after PIC. Only one 9 is allowed.

***** ERROR 164 - INVALID OPTION**
The language word IGNORE, ERROR or RESET must follow the language word OPTION in an OPTION statement.

***** ERROR 165 - 'ERROR' EXPECTED**
The word ERROR was not found in an ON ERROR statement in an FD declaration.

***** ERROR 166 - 'TYPE' EXPECTED**
The language word TYPE has not been recognised in an ORGANISATION statement. It should be coded following the organisation being declared.

***** ERROR 167 - 'EXTENSION' EXPECTED**
The language word EXTENSION has not been recognised in an ORGANISATION statement. It should be coded following the type.

***** ERROR 168 - 'LENGTH' OR SYMBOL EXPECTED**
The language word LENGTH has not been recognised in a RECORD LENGTH statement in an FD, or a symbol did not follow the word RECORD in an MD.

***** ERROR 169 - SYMBOL OR NUMERIC EXPECTED**
In the KEY LENGTH, RECORD LENGTH and the SIZE statements the key length, record length and size respectively must be specified as a numeric literal or an identifier.

***** ERROR 171 - ILLEGAL USE OF A RESERVED WORD**
A reserved word cannot be used as a file name in a FD statement or as a data name. See section 2.1.2.

***** ERROR 172 - 'ORGANISATION' EXPECTED**
The language word ORGANISATION (or ORGANIZATION) is missing or misplaced in an FD statement. It should be coded after the filename.

***** ERROR 173 - TOTAL NO. OF PLACES > 18**
A numeric data item cannot be declared with more than 18 decimal digits in total.

***** ERROR 174 - 'FILE' EXPECTED**
The language word FILE is missing or misplaced in an ASSIGN statement. It should follow the unit-id.

*** WARNING 175 - STRING TOO LONG**
A string specified in the ASSIGN statement is too long, and has been truncated according to the maximum lengths given below:

```
     unit-id       maximum length 3;
     file-id       maximum length 8;
     volume-id maximum length 6.
```

***** ERROR 176 - ILLEGAL LEVEL NUMBER**

A data declaration does not contain a legal level number.

***** ERROR 177 - MULTIPLE LABELS NOT ALLOWED**
A line may not have more than one label.

***** ERROR 178 - MORE THAN FIVE AREA NAMES**
The AREAS (or AREA) statement flagged in error introduces a list containing more than five area names, thereby exceeding the maximum number allowed.

***** ERROR 179 - IDENTIFIER EXPECTED**
The specified item is not a valid Global Cobol symbol. A Global Cobol symbol is always acceptable at this point, but in some cases other items may also be acceptable (e.g. a string).

*** WARNING 180 - ROUNDED UP TO MULTIPLE OF 8**
The length specified in the EXTENSION clause of an ORGANISATION statement should be an integral multiple of 8. It has been rounded up accordingly.

***** ERROR 181 - 'CONTAINS' EXPECTED**
The language word CONTAINS is missing or misplaced in a BLOCK statement. It should be coded after BLOCK.

***** ERROR 182 - 'CHARACTERS' EXPECTED**
The language word CHARACTERS is missing or misplaced in a BLOCK statement.

*** WARNING 183 - BLOCK STATEMENT ALREADY FOUND**
More than one BLOCK CONTAINS statement has been detected in a file definition. The BLOCK CONTAINS statement may appear at most once in a file definition. Further BLOCK CONTAINS statements will be ignored.

***** ERROR 184 - 'BLOCK' INVALID FOR ISAM FILES**
A BLOCK CONTAINS statement has been detected in an indexed sequential file definition. This is not allowed.

*** WARNING 185 - RECORD STATEMENT ALREADY FOUND**
More than one RECORD statement has been detected in the same map definition. Since at most one such statement should appear, only the first RECORD statement of the definition will be compiled, and any others will be ignored.

***** ERROR 186 - DATA NAME UNDEFINED**
A non-symbolic name of the form:

"*data-name*"

has been coded in an AREAS (or AREA) statement, MAP clause, or ON ERROR statement, but the data-name has neither been defined in the current program, nor been declared a global symbol by a preceding GLOBAL statement. It is impossible to set up the pointer to address the area, and hence this error is signalled.

***** ERROR 187 - VARIANT ALREADY FOUND**
More than one VARIANT statement has been detected in the same map definition. Since at most one such statement should appear, only the

first VARIANT statement of the definition will be compiled, and any others will be ignored.

*** WARNING 188 - AREAS STATEMENT ALREADY FOUND**
More than one AREAS (or AREA) statement has been detected in the same map definition. Since at most one such statement should appear, only the first AREAS (or AREA) statement will be compiled, and any others will be ignored.

***** ERROR 190 - CANNOT BE LINKAGE SECTION ITEM**
A non-symbolic name of the form:

"*data-name*"

has been coded in an AREAS (or AREA) statement, MAP clause, or ON ERROR statement, but the data name involved is defined in the linkage section. It is impossible to initialise the pointer to address a linkage section item, and in consequence this error is signalled.

***** ERROR 191 - 'MAP' EXPECTED**
The language word MAP is missing or misplaced in a map definition coded in working storage. MAP may only be omitted from an MD occupying the linkage section. In a working storage map definition it must immediately follow the mapname.

***** ERROR 201 - DOUBLY DEFINED LABEL**
A label previously found in the procedure division or declared as a data item in the data division has been detected. The full stop does not form part of the label name. The label should be altered to make it unique.

*** WARNING 202 - DOUBTFUL SECTION TERMINATION**
A SECTION, ENTRY or ENDPROG statement has been encountered which is not preceded by a STOP, EXIT or GO TO statement. Valid code will be generated for the section or entry but this is not recommended coding practice.

*** WARNING 203 - CONSECUTIVE ENTRY STMTS FOUND**
An ENTRY statement followed immediately by another ENTRY statement has been detected. Code will be generated for both ENTRY statements but execution via the first entry point will be doubtful.

*** WARNING 204 - 'DO' NEST LEVEL NON-ZERO**
The DO nest level was found to be non-zero upon encountering a SECTION or ENTRY statement, probably indicating a missing ENDDO statement in a previous section.

*** WARNING 205 - 'IF' NEST LEVEL NON-ZERO**
The IF nest level was found to be non-zero upon encountering a SECTION or ENTRY statement, probably indicating a missing END statement in a previous section.

*** WARNING 206 - 'END' EXPECTED**
The range of an IF, ACCEPT or ON structure declared within a DO structure extends beyond the range of the DO structure.

***** ERROR 207 - NO MATCHING 'IF' FOUND**

An END statement has been encountered for which no corresponding IF, ACCEPT or ON statement was previously detected.

*** WARNING 208 - 'ENDDO' EXPECTED**
The range of a DO structure declared within an IF, ACCEPT or ON structure extends beyond the range of the IF, ACCEPT or ON structure.

***** ERROR 209 - 'IF..ELSE..ELSE' DETECTED**
Consecutive ELSEs attached to the same IF, ACCEPT or ON statement have been detected. No code will be generated for the second ELSE.

***** ERROR 210 - MORE THAN 16 NESTED "DO'S"**
The maximum nesting level for DO structures has been exceeded. No code will be generated for any subsequent DO statements until an ENDDO statement is encountered.

***** ERROR 211 - NO MATCHING 'DO' FOUND**
An ENDDO or FINISH statement has been encountered with no corresponding DO statement found previously.

***** ERROR 212 - MORE THAN 32 NESTED "IF'S"**
The maximum nesting level for IF, ACCEPT and ON structures (32 overall) has been exceeded. No code will be generated for subsequent IF, ACCEPT and ON block condition statements until an END statement is encountered.

***** ERROR 213 - OPERAND OVER 80 CHARACTERS**
The sending field of a DISPLAY statement or the receiving field of an ACCEPT statement must not be greater than 80 bytes in length.

*** WARNING 215 - INACCESSIBLE CODE LINE**
This indicates an unlabelled line of code which is not a SECTION or ENTRY statement following an unconditional program jump. Object code is generated for the line.

***** ERROR 216 - LINE OUT OF CONTEXT**
A TO statement must be preceded by a GO TO DEPENDING ON statement. An AND or OR statement must be preceded by an initial conditional statement.

***** ERROR 217 - 'TO' STATEMENT EXPECTED**
A GO TO DEPENDING ON statement not followed by any TO statements has been encountered. The statement is ignored. Every GO TO DEPENDING ON must be immediately followed by at least one TO statement.

***** ERROR 218 - 'AND' AND 'OR' CANNOT BE MIXED**
Compound conditions may contain either AND or OR statements but not both.

*** WARNING 220 - TARGET IS AN ENTRYNAME**
The target of a PERFORM or GO TO statement is a program entry point.A valid jump will be generated but this is not recommended programming practice.

***** ERROR 221 - INVALID OPERAND TYPE**
This type of operand is not allowed in this position. Allowable operands will depend on the statement type, and Chapter 4 should be consulted for details.

*** WARNING 222 - NOT AN INDEXED VARIABLE**
An index has been specified for a variable declared without an OCCURS clause. Object code to process the index will nevertheless be generated.

*** WARNING 223 - INDEX EXPECTED. (1) ASSUMED**
A variable defined with an OCCURS clause has been referenced with no index specified. A default index of one is assumed.

***** ERROR 224 - INTEGER OR COMP INDEX ONLY**
An index of the wrong type has been encountered. Valid indices must be either unindexed computational items or unsigned, non-zero integers.

***** ERROR 225 - TARGET MUST BE COMP**
The target of an ADD, SUBTRACT, MULTIPLY or DIVIDE statement with no GIVING clause specified must be a computational item.

***** ERROR 226 - DOUBLY DEFINED SECTION/ENTRY**
A section name or entry name has been encountered which was previously declared as a label, section name or entry name in the procedure division or as a data item in the data division. The name should be altered to make it unique.

*** WARNING 227 - VALID FOR ISAM AND DMAM ONLY**
An index area has been specified in an OPEN operation for a file not declared as indexed-sequential or data management. Index area specification is valid only for the indexed-sequential and data management access methods. The specification is ignored.

***** ERROR 228 - VARIABLE TOO SHORT**
The index area specified in an OPEN statement is too short. Index areas must be at least 256 bytes long for indexed-sequential files, and at least 516 bytes long for data management files.

***** ERROR 229 - FILENAME EXPECTED**
The first operand of an I/O operation statement must always be a filename.

***** ERROR 231 - L.S VARIABLE EXPECTED**
All operands of a USING clause in an ENTRY statement and the first operand of a BASE statement must be linkage section items.

***** ERROR 232 - MUST NOT REDEFINE A SUBGROUP**
The operand of a USING clause in an ENTRY statement or the first operand of a BASE statement must not be a redefinition of a subordinate (02-49) level.

***** ERROR 233 - ITEM MUST BE 01, 77, FD or MD**
All operands of a USING clause in an ENTRY statement and the first operand of a BASE statement must be 01 or 77 level items, FDs, or MDs, or their redefinition.

***** ERROR 234 - DOUBLY DEFINED SYSTEM ROUTINE**
A user symbol in this compilation has the same name as a Global Cobol system routine called from this compilation. To avoid problems of this nature, it is recommended that the $ character is not used within user-defined symbols.

***** ERROR 235 - UNDEFINED OPERAND**
The indicated operand has not been defined within the compilation unit.

***** ERROR 236 - CANNOT 'CALL' THE PROGRAM NAME**
The target of a CALL statement must be an entry name or external name. The program name may not be used as an entry name.

*** WARNING 237 - FIELD LONGER THAN 8 CHARACTERS**
The target of a LOAD, CHAIN, RUN or EXEC statement is longer than eight characters. Object code will be generated but only the first eight characters of the target will be significant.

***** ERROR 238 - UNSUPPORTED OPCODE**
The opcode specified is not supported for this statement in this version of the compiler.

***** ERROR 239 - 'OPEN NEW' INVALID**
An OPEN NEW operation on an indexed-sequential or data management file has been encountered. Indexed-sequential and data management files must be created separately before use.

***** ERROR 240 - ILLEGAL PIC ON LINE OR COL VAR**
Operands of LINE or COL clauses must be PIC 9(4) COMP variables or positive integer literals.

***** ERROR 241 - MAPNAME EXPECTED**
The first operand of a MAPOUT, MAPCLEAR or MAPIN statement must always be a mapname, or in the case of MAPIN the word ENTRY.

***** ERROR 242 - 'DO' NEST LEVEL NON-ZERO**
The DO nest level was found to be non-zero upon encountering the ENDPROG statement, indicating a missing ENDDO statement within the program.

***** ERROR 243 - 'IF' NEST LEVEL NON-ZERO**
The IF nest level was found to be non-zero upon encountering the ENDPROG statement, indicating a missing END statement within the program.

***** ERROR 244 - ZERO LENGTH ACCEPT NOT ALLOWED**
The variable specified in an ACCEPT statement was an empty group or subgroup item. It must be at least 1 byte long.

***** ERROR 246 - MAXIMUM SIZE IS PIC 9(12,6)**
The first operand of an EDIT statement can have a maximum of twelve digits preceding the decimal point, and six digits following the decimal point.

*** WARNING 247 - MAXIMUM LENGTH IS 30 CHARS**
The second operand of an EDIT statement, the receiving field, cannot be longer than 30 characters. Additional characters will remain unchanged.

***** ERROR 249 - MUST BE PIC [S]9(4) COMP**

If a variable is specified as the field number operand of a MAPOUT, MAPCLEAR or MAPIN statement, its picture clause must be either PIC 9(4) COMP or PIC S9(4) COMP.

*** **ERROR 251 - MULTIPLE LABELS NOT ALLOWED**
More than one label has been found on a line. Only one label per line is allowed. The labels should be recoded on separate lines.

*** **ERROR 252 - LINE MUST NOT BE LABELLED**
A label has been encountered on a SECTION, ENTRY, TO, AND or OR statement. Labelling of these statements is not permitted.

*** **ERROR 253 - INVALID OPERAND**
The indicated operand is of the wrong type. For example attempting to MOVE a data item to a literal will cause this error.

*** **ERROR 254 - 'END OF LINE' EXPECTED**
A carriage-return line-feed sequence or a legal comment were expected but not found. Legal comments begin with an asterisk.

*** **ERROR 255 - RIGHT PARENTHESIS EXPECTED**
The closing round bracket on an index was expected in the specified position.

*** **ERROR 256 - INVALID ARITHMETIC CONSTRUCT**
The word following the first operand of an arithmetic statement is invalid. The correct constructs are:

```
ADD...TO
SUBTRACT...FROM
MULTIPLY...BY
DIVIDE...INTO
```

*** **ERROR 257 - 'GIVING' OR 'ROUNDED' EXPECTED**
An ADD, SUBTRACT, MULTIPLY or DIVIDE statement has continued beyond its second operand. However the continuation is not one of ROUNDED, GIVING or a legal comment. These are the only valid continuations for the statements.

*** **ERROR 258 - 'AT' OR 'ON' EXPECTED**
AT or ON cannot be found after the first operand of a BASE statement.

*** **ERROR 259 - TOO MANY PARAMETERS**
Too many parameters have been encountered in the USING clause of a CALL, ENTRY or SVC statement. The maximum number of parameters permitted is seven.

*** **ERROR 260 - INVALID ACTION**
The action specified at the end of an IF, ACCEPT or ON statement is not one of PERFORM, GO TO, STOP, EXIT or carriage-return line-feed. Other single-line actions should be specified on the next line with ELSE or END coded on the line following.

*** **ERROR 261 - ILLEGAL USE OF A RESERVED WORD**
The programmer has attempted to use a reserved word in a position where it is not permitted.

*** **ERROR 262 - INVALID 'DO' STATEMENT**

DO may only be followed by the language words WHILE, UNTIL or FOR.

***** ERROR 263 - 'GO TO DEPENDING' INVALID**
GO TO DEPENDING ON may not be coded as an action on the end of an IF, ACCEPT or ON statement. It should be coded on the next line followed by the appropriate number of TO statements, followed by either an ELSE or END statement.

***** ERROR 264 - EXCEPTION/OVERFLOW EXPECTED**
ON must always be followed by OVERFLOW or EXCEPTION.

***** ERROR 265 - OLD, NEW OR SHARED EXPECTED**
OPEN must always be followed by one of OLD, NEW or SHARED.

***** ERROR 266 - INTEGER EXPECTED**
The operand in the indicated position must be a valid integer.

***** ERROR 267 - INVALID CONDITION**
Either the condition clause of the indicated DO, IF, AND or OR statement does not conform to any valid format described in Table 4.6.3, or the condition operand of a $JUMP statement is not one of EQ, NE, GT, LT, GE or LE.

***** ERROR 268 - STRING EXPECTED**
The operand in the indicated position must be a valid Global Cobol character string.

***** ERROR 269 - VARIABLE EXPECTED**
The operand involved in a ZERO, POSITIVE, NEGATIVE, HIGH-VALUES, LOW-VALUES, SPACES or NUMERIC test in an IF or DO statement must be a valid Global Cobol variable (possibly indexed).

***** ERROR 270 - BOOLEAN CONDITION EXPECTED**
One of ZERO, POSITIVE, NEGATIVE, NUMERIC, EQUAL, =, LESS, <, GREATER, >, or a figurative constant cannot be found after the first operand (and possibly a NOT) of an IF or DO statement.

***** ERROR 271 - 'FROM' EXPECTED**
FROM cannot be found after the first operand of a WRITE or REWRITE statement.

***** ERROR 272 - 'USING' EXPECTED**
Non-comment characters have been detected after the entry name in an ENTRY statement. However the language word USING has not been found.

***** ERROR 273 - 'INTO' EXPECTED**
INTO cannot be found after the first operand of a READ or EDIT statement.

***** ERROR 274 - 'AT' EXPECTED**
AT cannot be found after the first operand of a POINT statement.

***** ERROR 275 - TOO MANY TARGETS**
Too many targets have been encountered in a multiple-target MOVE statement. The compiler will handle a maximum of seven targets on any one MOVE statement. The statement should be split into as many MOVEs as are required to bring the number of targets within the allowable range.

***** ERROR 276 - INDEXED VARIABLE NOT ALLOWED**
An indexed pointer has been detected as the target of the transfer of control part of a format 2 condition statement. The statement should be recoded as a format 1 condition.

***** ERROR 277 - UNEXPECTED END OF LINE**
A mandatory operand has been omitted.

***** ERROR 279 - 'TO' EXPECTED**
The language word TO was not found in the expected position in a MOVE or GO statement.

***** ERROR 280 - MUST BE UNSIGNED**
The integer in the specified position must not be preceded by + or -.

***** ERROR 282 - NUMBER OUTSIDE VALID RANGE**
The specified integer must be in the range 1 to 99 for the SVC statement and 0 to 31 for the $CC statement.

***** ERROR 283 - MUST BE POSITIVE**
Index values must be strictly positive.

*** WARNING 284 - CONDITION HAS NO VARIABLE PART**
Both operands of a comparison in an IF or DO statement have been found to be literals. Valid code will still be generated.

***** ERROR 285 - 'ON' EXPECTED**
The ON following GO TO DEPENDING cannot be found. The statement will be ignored.

***** ERROR 286 - INVALID MAPPING OPTION**
The option specified in a MAPOUT, MAPIN or MAPCLEAR statement must be one of the words: ALL; FIELD; OUTPUT; PRINT; QUERY; RECORD; or TEXT.

***** ERROR 288 - 'FORMAT' EXPECTED**
The word FORMAT was not found in an EDIT statement.

***** ERROR 289 - INVALID OPTION**
The mapping option specified is not valid for this MAPOUT, MAPCLEAR or MAPIN statement. See the Global Screen Presentation Manual for details of valid options.

***** ERROR 290 - '=' EXPECTED**
An equals sign cannot be found after the first operand in a DO FOR statement.

***** ERROR 291 - 'TO' OR 'STEP' EXPECTED**
Non-comment characters have been detected after the second operand in a DO FOR statement. However the language words TO or STEP have not been found.

***** ERROR 292 - 'STEP' EXPECTED**
Non-comment characters have been detected after the third operand in a DO FOR statement. However the language word STEP has not been found.

***** ERROR 293 - INVALID VALUE**

The VALUE clause for a PIC DATE or PIC FLT item has been given in an incorrect format.

***** ERROR 294 - LOCK TYPE EXPECTED**
Non-comment characters have been found at the end or a READ statement. For data management files the language words LOCK, PROTECT or DELETE-LOCK can optionally appear.