Global 16-bit Development System
System Subroutines Manual
Version 8.1

TABLE OF CONTENTS

Section Description Page Number

APPENDICES

Appendix Description Page Number

1. Introduction

1.1 System Subroutines

Global Cobol provides a number of powerful subroutines which extend the Cobol language with a variety of commonly required functions. These subroutines, known as **system subroutines**, are invoked by the CALL statement. For example:

```
CALL DOWK$ USING DATE DAY
```

might be used to determine which day of the week a given date represents.

1.1.1 System Routine Entry Names

The entry name of a system subroutine, which is of course a global symbol, always ends with a $ character. Therefore, providing programmers do not create symbols containing the $ character, this convention guarantees that the name will not erroneously duplicate any other name appearing in the compilation or linkage edit.

1.1.2 The Global Cobol System Library

System routines are held, together with the routines for the access methods, in compilation file format in the system libraries. Routines from these libraries are incorporated in a program when it is linkage edited.

It is an important feature of Global Cobol that only the system subroutines and access methods that a program actually requires are included in it from the system libraries. The libraries can be as comprehensive as is required without impacting on the size of programs at all. In particular, the libraries can be expanded to contain new services without affecting existing programs in any way.

Important Note: The Program Names of some System Subroutines have been changed between V6.2 and V8.1. Any $LINK jobs that include explicit Program Names to include specific System Subroutines (e.g. BA$A to include CUST$ - see Appendix A) must be checked after the Global Cobol Development System is upgraded from V6.2 to V8.1.

1.1.3 Exceptions Returned by System Subroutines

System subroutines indicate abnormal processing conditions by generating exceptions. The CALL statement that invokes such a subroutine may therefore be followed by an ON EXCEPTION statement introducing the logic which is executed should an exception arise.

Exception condition 1 ($$COND = 1) is often used to indicate that the routine has been terminated by an irrecoverable I/O error, in which case an explanatory message produced by the monitor's I/O error retry routine will appear on the screen.

Other exception conditions may be returned, depending on the subroutine involved. Sometimes the result code ($$RES) will be established to provide further information about the exception. Often an exception should never occur when the user program is properly debugged, and in this case it is unnecessary to code an ON EXCEPTION statement following the CALL because the program will be automatically terminated in error should the unexpected condition actually occur.

1.1.4 File Handling and Screen Handling Subroutines

There are several other system subroutines available besides those collected in this manual. Those concerned with file handling are described in chapter 9 of the Global Development File Management Manual and those involving screen and console handling will be found in chapter 7 of the Global Development Screen Presentation Manual.

1.2 System Variables

System variables are elementary data items which are conceptually declared automatically in the data division of every compilation. The variables do not actually appear as data definitions; neither do they occupy working storage. They are located within a permanently available region known as the System Area, which is used to communicate parameter information between the Global System Manager and an application program. They can be referenced from procedure division statements whenever a level 77 item with the same picture clause would be valid.

2. Date and Time Subroutines

2.1 The Date Conversion Routines, DS-DT$,DL-DT$,DT-DS$ & DT-DL$

The date conversion routines are used to convert a date from internal (computational) to external (character) format, or vice versa. They can also be employed to validate dates. The external format is the form in which dates are printed and displayed, whereas the internal format is used for storing dates compactly in files.

The conversion routines are capable of processing any date occurring in the 1,637 years starting January 1 1063 AD and ending December 31 2699 AD in the Gregorian calendar.

Internal format dates are PIC 9(6) COMP fields (3 bytes) containing the date in the form:

10000*(year - 1900) + 100*(month of year) + day of month

Thus if two internal dates are compared the date which is later will be the greater.

External format dates may be either short dates (DS) for which the century is omitted and assumed to be the 100 year range indicated by the start century year customized using $CUS (see the Global System Manager Manual and system variable $$NCYR), or long dates (DL) which include the full year as four digits. Short dates are PIC X(8) fields, and long dates PIC X(10) fields, of the form:

dd/mm/yy (short) or *dd/mm/ccyy* (long)

when European processing is in force ($$USA = 0), or:

mm/dd/yy (short) or *mm/dd/ccyy* (long)

for American processing ($$USA = 1). Here *dd* and *mm* are one or two digit integers representing the day and month respectively, and *cc* and *yy* are two digit integers representing the century and year within the century. The century may be omitted in the long format, in which case the 100 year range indicated by the start century year (see above) is assumed. As well as "/", the characters "-", "." and "*" are also allowed as separators within the date. Dates may not contain leading or embedded spaces.

For example, suppose European date processing is in force, then the 1st May 1988 could be represented in short external date format as any of the following:

01/05/88 1/5/88*bb* 01-05-88 1.05.88*b*

where *b* represents a single space character. The first of these represents the standard format used by Global System Manager. All these examples would also be valid long format dates if padded out with two spaces on the right. The following, however, are not valid short dates:

*b*1/b5/88 *bb*1-5-88 01:05:88 1/5/1988

although the last of these would be a valid long date if two spaces were added on the right. This date in standard long format would be 01/05/1988.

2.1.1 Validating an External Format Date

You may check whether a field contains a valid external format date by coding a CALL statement of the form:

CALL DS-DT$ USING *short-date*

or:

CALL DL-DT$ USING *long-date*

where *short-date* identifies a PIC X(8) field, and *long-date* identifies a PIC X(10) field, containing the date to be validated. The contents of the field are not altered by the validation. An exception will be generated if the date is not in the valid format, or if it does not represent a legitimate date (leap year checks are performed).

2.1.2 Converting a Date from Internal to External Format

You may convert a date from internal to external format by coding a CALL of the form:

CALL DT-DS$ USING *int-date short-date*

or:

CALL DT-DL$ USING *int-date long-date*

where *int-date* is a PIC 9(6) COMP field containing the internal format date to be converted, which will be unaltered by the call. *Short-date* is a PIC X(8) field in which the resulting short format date is placed, and *long-date* is a PIC X(10) field in which the resulting long format date is placed. The day and month will always be two digits, with leading zeros if required, and the separator will be "/". The century is always included in the resulting long format date.

No exception is generated if *int-date* is invalid. If a wrong value is supplied it will be mechanically converted (for example,75/28/2413).

2.1.3 Converting a Date from External to Internal Format

You may convert a date from external format to internal format by means of a CALL of the form:

CALL DS-DT$ USING *short-date int-date*

or:

CALL DL-DT$ USING *long-date int-date*

where *short-date* is a PIC X(8) literal or variable containing a short format date to be converted, and *long-date* is a PIC X(10) literal or variable containing a long format date to be converted. *int-date* is a PIC 9(6) COMP field in which the resulting internal format date will be placed. The external format date is unaltered by the conversion.

If the external date is not in valid date format, or does not represent a legitimate date, then an exception will be returned and *int-date* will remain unchanged.

2.1.4 Programming Notes

Programmers developing portable applications should take care never to store an external date on a data file which might be transported between American and European installations, since the meaning (and even the validity) of such a date changes as $$USA changes.

2.1.5 Memory Page Versions

Memory Page versions of these subroutines are available. These versions have most of the subroutine code within Global System Manager making the subroutine smaller. Refer to Appendix B for further details of Memory Page subroutines.

2.2 Date to Day Number Conversion, DT-DY$ & DY-DT$

The date to day number system routine allows you to convert an internal format date to a day number, 1 January 1900 counting as day 1. You can also convert day numbers, which are held as PIC 9(6) COMP fields, back to internal date format. The routine is useful when you need to determine the number of days between two dates, often a requirement for interest and other financial calculations.

The routine will handle dates between 1 January 1063 and 31 December 2699, using the Gregorian Calendar. Note that although the Gregorian calendar was first introduced in 1582, it was not adopted until much later in most countries, two of the last being the USSR (1918) and Greece (1923).

2.2.1 Internal Date to Day Number

You can convert an internal date to the corresponding day number by a CALL statement of the form:

CALL DT-DY$ USING *int-date day-number*

where *int-date* is a PIC 9(6) COMP field containing the internal format date, unaltered by DT-DY$, and *day-number* is a PIC 9(6) COMP field in which the day number will be returned if all is well. An exception is returned, and the day number field remains unchanged, if the internal date is not valid.

2.2.2 Day Number to Internal Date

A day number can be converted to internal date format by a CALL statement of the form:

CALL DY-DT$ USING *day-number int-date*

The first parameter, *day-number*, is the name of a PIC 9(6) COMP field containing the day number, and is unaltered by DY-DT$. The second parameter, *int-date*, is the name of another PIC 9(6) COMP field in which the internal date is returned if all goes well. An exception is returned if the day number is not within the valid range, and in this case *int-date* will remain undisturbed.

2.2.3 Exception Conditions

DT-DY$ returns exception condition 1 if its first parameter is not a valid internal format date.

DY-DT$ returns exception condition 1 if its first parameter does not represent a date within the valid range.

2.3 The Time of Day Routine, TIME$

The TIME$ routine is used to obtain the current time of day as an eight character field of the form hh.mm.ss, for example 07.21.03 or 15.08.57.

2.3.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL TIME$ USING time
```

where the parameter *time* is the name of a PIC X(8) field in which the time will be returned.

2.3.2 Exceptions

Exception condition 1 will be returned if time of day processing is not supported by the machine.

2.3.3 Programming Note

If the machine is running at midnight the time and date will not be reset until $D, or the MIDN$ routine described in 2.6, is used to change the date and time. Otherwise the hours will simply continue to be incremented (e.g. 25.17.03). This is done to give the user full control over the date to be used for processing, whilst still retaining unambiguous time/date combinations.

2.4 The Day of the Week Routine, DOWK$

The DOWK$ routine is used to determine the day of the week corresponding to an internal format date.

2.4.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL DOWK$ USING int-date day
```

where *int-date* is a PIC 9(6) COMP field containing the internal format date, and *day* is a PIC 9 COMP field in which the day of the week will be returned, with 1 representing Sunday, 2 representing Monday, and so on. The field *int-date* is unaltered by DOWK$.

2.5 The Time Conversion Routines, T-HMS$ & HMS-T$

The T-HMS$ system routine is used to convert a time from internal to external format, and the HMS-T$ routine to perform the reverse conversion. The HMS-T$ routine can also be employed to check that a PIC X(8) field contains a valid external format time.

External format times are PIC X(8) fields of the form:

hh.mm.ss

where *hh, mm* and *ss* are two digit integers representing the hour, minute and second respectively, counting from midnight. On output the separator is always set to ".", but on input any of the six characters "*", "+", "'", ".", "-" and "/" are acceptable. Times may be greater than 24.00.00: such times can be useful when you wish to run with the previous day's date. For example, 25.15.00 indicates a time of 1.15 am on the day after $$DATE.

Internal format times are PIC 9(9) COMP fields containing the time in seconds, that is:

3600 * hours + 60 * minutes + seconds

2.5.1 Internal to External Time Conversion

You can convert an internal time to the corresponding external time by a CALL statement of the form:

CALL T-HMS$ USING *int-time ext-time*

where *int-time* is a PIC 9(9) COMP field containing the internal time to be converted, and *ext-time* is a PIC X(8) field in which the external time is returned. The field *int-time* is unaltered by T-HMS$. If *int-time* contains an invalid format time then *ext-time* will be set to "********", but no exception is generated.

2.5.2 External to Internal Time Conversion

You can convert an external time to the corresponding internal time by a CALL statement of the form:

CALL HMS-T$ USING *ext-time int-time*

where *ext-time* is a PIC X(8) field containing the external time to be converted, and *int-time* is a PIC 9(9) COMP field in which the internal time will be returned. The field *ext-time* is unaltered by HMS-T$. If *ext-time* contains an invalid time then exception condition 1 is returned, and *int-time* remains unchanged.

2.6 The Elapsed Time and Midnight Routines, SECS$, MIDN$, MIDCH$

The SECS$ routine is used to determine the elapsed time, in seconds, since Global System Manager was initiated. By calling the routine at the beginning and end of an activity, and subtracting one time from the other, you can obtain its duration in seconds.

The MIDN$ routine advances the internal date, $$DATE, by one day, and reduces the time of day by 24 hours if it is after midnight. If midnight has not passed the routine leaves the date and time of day unaltered. In either case the elapsed time supplied by SECS$ will remain unaffected. For example, suppose Global System Manager was initiated at noon on 14/05/88, and has been running exactly 13 hours. Then, assuming MIDN$ has not been called, the date is still 14/05/88, the elapsed time is 46800 seconds, and the time of day is 25.00.00. One second later, following a call on MIDN$, the date is 15/05/88, the elapsed time is 46801, and the time of day is 1.00.01.

Note that the Global System Manager menu system can optionally invoke MIDN$ automatically when a menu entry is selected.

2.6.1 Calculating the Elapsed Time

To calculate the elapsed time, invoke the SECS$ routine with a call of the form:

CALL SECS$ USING *elapsed-time*

where *elapsed-time* is a PIC 9(9) COMP field in which the number of seconds which have passed since Global System Manager was initiated will be returned providing the system supports a timer. If there is no timer, exception condition 1 is returned, and the field is not updated.

2.6.2 Advancing the Date After Midnight

To advance the date after midnight, invoke MIDN$ with a parameter-less call:

```
CALL MIDN$
```

Exception condition 1 will be returned if the system does not support a timer.

2.6.3 Checking to See if it is After Midnight

To check if the time is after midnight, you must invoke MIDCH$ with a parameter-less call of the form:

```
CALL MIDCH$
```

An exception is returned if the time is after midnight; this enables you to warn the operator before calling MIDN$ and/or performing any end of session routines.

If it is before midnight, or if the system does not support a timer, the routine has no effect.

2.7 The Date Formatting Routine, FDAT$

The FDAT$ system routine enables you to convert dates held in internal form into a variety of external formats for display and other purposes.

2.7.1 Invocation

FDAT$ is invoked by a call of the form:

```
CALL FDAT$ USING int-date format area
```

where *int-date* is the PIC 9(6) field holding the date in internal form, *format* defines the way the date will be formatted and *area* is the area where the formatted date will be placed, which must not be more than 1 byte shorter than format.

2.7.2 The Format

The format area contains characters used to determine how the date will be formatted, and must be terminated by a $ character. Characters other than those substituted by FDAT$ (e.g. "/" or ".") will be copied unchanged to the output area.

The following values will be substituted:

DD or dd	receive the day number (e.g. 12, 31);
MMM or mmm	receive the 3 character upper or upper and lower case abbreviated form of the month name (e.g. JUL or Jul);

MM or mm receive the 2 digit month number (e.g. 01 - 12);

YYYY or yyyy receive the 4 digit year number (e.g. 1988);

YY or yy receive the 2 digit year number (e.g. 88).

The longer forms are substituted in preference to the shorter forms; thus 1988 will appear in preference to 88. In addition, there may only be a maximum of one value for each date item; thus either YYYY or YY should be defined, not both.

For example, to produce a standard format external date for European use one would define format as DD/MM/YY$. This would produce dates of the form: 17/07/88.

Alternatively, to produce an unequivocal date for use in the UK or North America you should define the format as "DD mmm YYYY$". This will produce dates of the form: 17 Jul 1988.

2.7.3 Leading Zeros

Zeros will be omitted if the character defined in *format* **preceding** the field to be substituted is a space. In addition, if the first character of the substituted string is the first character of the format, leading zeros will be omitted if the first character after the field is a space. Thus:

DD-	becomes	01-
-DD	becomes	-01
DD*b*	becomes	*b*1*b*
*b*DD	becomes	*bb*1

A stop code is produced if the format defined is not validly terminated with a $ character.

2.7.4 Examples

Given a system date of the seventeenth of May, 1987 the following formats would give the results indicated:

Format	*Result*
DD/MM/YY$	17/05/88
DD-mmm-YYYY$	17-May-1988
MM*b*YY$	5 88
Day DD Month MM Year YYYY$	Day 17 Month 5 Year 1988

3. Data Conversion Subroutines

3.1 The Random Number Generator, RAND$

The RAND$ system routine allows you to generate a sequence of pseudo-random numbers. These are positive integers uniformly distributed between 1 and an upper limit which you supply as a parameter whenever the routine is called.

3.1.1 Setting the Seed

The random numbers are generated using a seed field maintained in the PIC S9(9) COMP system variable $$SEED. If you wish to obtain a repeatable sequence of random numbers, as may be useful in debugging, you should set $$SEED to a specific value between -2147483648 and +2147483647 inclusive before calling RAND$ for the first time. If you do not initialise $$SEED then the sequence of random numbers generated will be truly unpredictable.

3.1.2 Invocation

You invoke the random number generator with a call of the form:

```
CALL RAND$ USING limit number
```

where *limit* must be a PIC 9(4) COMP variable or integer literal containing a value between 1 and 32767 inclusive, and *number* is the name of a PIC 9(4) COMP variable in which the routine returns a random integer between 1 and the limit you have specified. Note that if you mistakenly supply a limit which is not a positive integer your program will be terminated in error.

3.1.3 Examples

The statement:

```
CALL RAND$ USING 6 FACE
```

simulates the throwing of a die, since a random integer between 1 and 6 is returned in the PIC 9(4) COMP variable FACE each time the statement is executed.

The tossing of a coin is simulated by:

```
CALL RAND$ USING 2 SIDE
```

when the value 1 is returned in SIDE this may be taken to represent "heads", the value 2 being "tails".

3.2 The Square Root Routine, SQRT$

The SQRT$ system routine enables you to calculate the square root of a non-negative PIC 9(11,7) COMP field correct to seven decimal places.

3.2.1 Invocation

The square root routine is invoked by a call of the form:

```
CALL SQRT$ USING number root
```

where *number* is the name of the field whose square root will be returned in *root* if all goes well. Both fields must be defined as PIC 9(11,7) COMP.

The first parameter, *number*, is unchanged by SQRT$. If the value it contains is negative, or is greater than or equal to 10^12, an exception will be returned and in this case the root field will not be updated.

3.2.2 Exception Conditions

Exception condition 1 is returned if you attempt to use SQRT$ to find the square root of a negative number or one which is too large.

3.3 The Zero Fill Routine, ZERO$

The ZERO$ system routine is used to replace all leading spaces in an unsigned display numeric variable by zeros.

3.3.1 Invocation

The routine is invoked by a CALL of the form:

CALL ZERO$ USING *variable-name*

The *variable-name* is the name of the display numeric field which is to have its leading spaces replaced by zeros.

3.3.2 Processing

The routine examines the field character by character starting from the left and replaces spaces by zeros until a character is found which is not space.

3.3.3 Programming Note

The display numeric field must contain at least one character which is not a space, otherwise the field following the display numeric field will be corrupted if it starts with a space. The routine can be used on a character variable to replace leading spaces by zeros if required.

Note that if a display numeric field is input from the console by an ACCEPT or ACCEPT...LINE statement, any leading zeros not in the units position are returned as spaces.

3.4 Text/Print Line Conversion Routines, PRIN$ & TEXT$

The PRIN$ system routine takes a text line terminated by a binary zero byte (as returned by the text file access method) and either displays it on the screen or expands it into a print line, the first character of which is a print control byte. In both cases tabs are expanded, assuming standard tab settings at every eighth column (9, 17, 25 etc.).

The TEXT$ system routine takes a print line, prefixed by a print control byte, and converts it to a text line, terminated by a binary zero byte. Spaces are contracted into tabs, assuming standard tab settings at every eighth column (9, 17, 25 etc.).

3.4.1 Displaying a line

A text line is displayed on the screen using a CALL statement of the form:

CALL PRIN$ USING *line*

where *line* is the name of the text line to be displayed, which must be terminated by a binary zero byte.

If the line to be expanded is longer than the screen line width in $$WIDE then it will be truncated to that width, or 80 characters, whichever is the smaller. A form-feed or vertical-tab as the first character will be removed, as will any sequence of carriage-return and line-feed characters at the start of the line. The line will be displayed on a new line, unless it starts with a carriage-return character which is not itself followed by a line-feed or carriage-return character, in which case it is displayed on the current line of the screen. The screen should be in teletype mode when the line is displayed.

3.4.2 Converting a Text Line to a Print Line

A text line is converted to a print line using a CALL statement of the form:

CALL PRIN$ USING *line area* [*area-length*]

where *line* is the name of the text line, which must be terminated by a binary zero byte, *area* is the name of the area in which the print line is to be constructed, and *area-length* is a PIC 9(4) COMP field or integer literal specifying the length of the print line to be created inclusive of the print control byte. If this parameter is omitted the area in which the print line is built is assumed to be 133 bytes long.

The resulting line is prefixed by a print control byte, constructed as follows:

- if the first character of the line is a form-feed then it is removed and the control byte is set to -1 (new page);
- if the first character of the line is a vertical-tab then it is removed and the control byte is set to 3 (3 new lines);
- if the first character of the line is a carriage-return or line-feed then this and any other carriage-returns or line-feeds immediately following are removed and the control byte is set to the number of line-feeds removed;
- otherwise, if there are no control characters, the control byte is set to 1 (new line).

This means that control characters on the front of a text line will be interpreted as one would expect when printing, except that vertical-tabs always advance the paper by three lines. The case where there are no control characters can only arise when printing the first line of a text file.

The remainder of the line is placed in the output area with embedded tabs expanded. If this would result in the line-length being exceeded the extra characters are ignored. A short line is padded with rightmost blanks to fill up the area.

3.4.3 Converting a Print Line to a Text Line

A print line is converted to a text line using a CALL statement of the form:

CALL TEXT$ USING *line area* [*line-length*]

where *line* is the name of an area containing the print line, prefixed by a print control byte, *area* is the name of the area in which the text line will be constructed, and *line-length* is a PIC 9(4) COMP field or integer literal containing the length of the print line, inclusive of the print control byte. If this parameter is omitted a line-length of 133 bytes is assumed.

The size of the area must be at least 32 bytes plus the line-length, otherwise unpredictable errors may occur.

The text line constructed in the area includes prefix characters corresponding to the print control byte, the remainder of the print line information and, finally, a terminating binary zero byte. The table below shows the prefix characters generated for particular control byte settings:

Print Control Byte (*n*)	**Prefix character(s)**
n < -1	None. A null line, consisting of a single binary zero byte is returned (corresponding to a forms control record)
n = 1	Form-feed
0 ≤ *n* ≤ 31	Carriage-return + *n* line-feeds
n > 31	Carriage-return + line-feed

Table 3.4.3 - Prefix Characters in a Text Line

The print line information is converted to text line format by removing any trailing rightmost spaces and replacing multiple contiguous interior spaces by tabs wherever possible, assuming standard settings at columns 9, 17, 25 etc. A completely blank line is represented by just a single space. The line information is always followed by a terminating binary zero byte.

3.4.4 Programming Notes

PRIN$ and TEXT$ are most commonly used when converting data from print file format to text file format, or vice versa. Text and print file handling is fully described in the Global Development File Management Manual.

3.5 The Table Sort Routines, TSRT$ and QSRT$

The TSRT$ system routine is used to sort a table held in memory into the sequence given by a character key consisting of part or all of each table entry. The QSRT$ routine is similar, but uses the "Quicksort" algorithm and as a result is faster for sorting large tables when the sort key starts at the beginning of each table entry.

3.5.1 Invocation

The routines are invoked by means of a call of the form:

CALL TSRT$ USING *ts table*

or:

CALL QSRT$ USING *ts table*

The parameter *ts* is the name of a table sort control area in which you must establish five PIC 9(4) COMP fields in the order listed below. This area is read-only to the sort and contains:

- the length of each table entry in bytes;
- the number of table entries;
- a work field used by the sort;
- the byte number of the start of the sort key within each table entry (the first byte of the table entry is taken as byte number 1);
- the length of the sort key in bytes.

The parameter *table* is the name of the table to be sorted.

3.5.2 Processing

The control block is assumed to be valid: unpredictable errors will occur if it is not. The table entries are sorted into the order given by their embedded keys. If two entries have the same key then their resulting relative sequence is not predictable.

3.5.3 Programming Notes

It is recommended that QSRT$ should be used in preference to TSRT$ for tables containing more than 500 entries when the sort key starts at the beginning of each table entry. If QSRT$ is used in other circumstances it will be no faster than TSRT$, and will occupy more memory.

3.6 Hexadecimal to Binary Conversion, HX-BI$ & BI-HX$

Two system routine entry points are provided to convert a hexadecimal string to binary, or vice versa. By a hexadecimal string we mean a sequence of an even number of ASCII characters, each of which is in the range 0-9 or A-F, and each of which represents 4 bits of the corresponding binary string. For example, the hexadecimal string "7F" corresponds to a binary string 1 byte in length, containing the bits:

0 1 1 1 1 1 1 1

3.6.1 To Convert a Hexadecimal String to Binary

To convert a hexadecimal string of length 2n bytes to the corresponding binary string of length *n* bytes you invoke the HX-BI$ system routine with a call of the form:

CALL HX-BI$ USING *hex binary n*

where *hex* labels the area containing the hexadecimal string, read-only as far as HX-BI$ is concerned; *binary* labels the area in which the binary string will be created; and *n* is the name of a PIC 9(4) COMP field, or integer literal, specifying the non-zero length of the binary string in bytes.

Exception condition 1 will be returned if any byte from the hexadecimal string area contains a value which is not ASCII 0-9 or A-F. If this exception occurs the contents of the binary string area will be unpredictable.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.6.2 To Convert a Binary String to Hexadecimal

To convert a binary string of length *n* bytes to the corresponding hexadecimal string of length 2*n* bytes you invoke the BI-HX$ system routine with a call of the form:

```
CALL BI-HX$ USING binary hex n
```

where *binary* labels the area containing the binary string, read-only as far as BI-HX$ is concerned; *hex* labels the area in which the hexadecimal string will be created; and *n* is the name of a PIC 9(4) COMP field, or integer literal, specifying the non-zero length of the binary string in bytes.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.6.3 Examples

In the examples below Z-PTR is a PIC PTR field and Z-HEX is a PIC X(4) field used to hold the equivalent hexadecimal string.

The following procedure division statements convert Z-PTR to hexadecimal in Z-HEX and display the result:

```
CALL BI-HX$ USING Z-PTR Z-HEX 2
DISPLAY Z-HEX
```

The statements below input and validate a hexadecimal string which is converted to binary in Z-PTR. If the input is invalid, the operator is re-prompted until the correct input is supplied:

```
AA010.
    DISPLAY "INPUT POINTER VALUE IN HEX"
    ACCEPT Z-HEX
    CALL HX-BI$ USING Z-HEX Z-PTR 2
    ON EXCEPTION GO TO AA010
```

3.7 ASCII to EBCDIC Conversion, AS-EB$ & EB-AS$

A routine is provided to convert an ASCII character string to EBCDIC or vice versa. Appendix A of the Global Development Cobol Language Manual contains a table of ASCII-EBCDIC equivalents.

3.7.1 To Convert an ASCII String to an EBCDIC String

To convert an ASCII string of length *n* bytes to the corresponding EBCDIC string, you invoke the AS-EB$ routine with a call of the form:

```
CALL AS-EB$ USING ascii ebcdic n
```

where *ascii* labels the area containing the ASCII string, read-only as far as AS-EB$ is concerned, *ebcdic* labels the area in which the EBCDIC string will be created, and *n* is either the name of a PIC 9(4) COMP field or an integer literal, specifying the non-zero length of each string in bytes. Only the junior seven bits of each ASCII character are used in the conversion, the setting of the senior bit being ignored. If the ASCII character does not have an EBCDIC equivalent, as indicated by the entry (none) in the table of ASCII-EBCDIC equivalents, then the character is converted to an EBCDIC "?" character.

Your program will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.7.2 To Convert an EBCDIC String to an ASCII String

To convert an EBCDIC string of length *n* bytes to the corresponding ASCII string, you invoke the EB-AS$ routine with a call of the form:

```
CALL EB-AS$ USING ebcdic ascii n
```

where *ebcdic* labels the area containing the EBCDIC string, read-only as far as EB-AS$ is concerned, *ascii* labels the area in which the ASCII string will be created, and *n* is either the name of a PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of each string in bytes. If the EBCDIC character does not appear in the table of ASCII-EBCDIC equivalents, then the character is converted to an ASCII "?" character.

Your program will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.7.3 Examples

In the examples below Z-AS and Z-EB are both PIC X(40) variables, Z-AS holding an ASCII string and Z-EB an EBCDIC string.

The following statements accept 40 ASCII characters into the Z-AS field and convert them to EBCDIC in the Z-EB field:

```
ACCEPT Z-AS
CALL AS-EB$ USING Z-AS Z-EB 40
```

The following statements convert the EBCDIC string in Z-EB to ASCII in the Z-AS field and then display the result:

```
CALL EB-AS$ USING Z-EB Z-AS 40
DISPLAY Z-AS
```

3.8 Binary to Bit String Conversion, BI-BS$ & BS-BI$

A routine is provided to convert a binary string to a bit string and vice versa. By a bit string we mean a sequence of a multiple of eight ASCII characters, each of which has the value "0" or "1", and each of which represents a bit of the corresponding binary string. For example, the 8-byte bit string "10110011" corresponds to a binary string 1 byte in length, containing the bits:

1 0 1 1 0 0 1 1

3.8.1 To Convert a Binary String to a Bit String

To convert a binary string of length *n* bytes to the corresponding bit string of length 8*n* bytes, you invoke the BI-BS$ routine with a call of the form:

CALL BI-BS$ USING *binary bitstring n*

where *binary* labels the area containing the binary string, read-only as far as BI-BS$ is concerned, *bitstring* labels the area in which the bit string will be created, and *n* is either the name of a PIC 9 (4) COMP field, or an integer literal, specifying the non-zero length of the binary string in bytes.

Your program will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.8.2 To Convert a Bit String to a Binary String

To convert a bit string of length 8*n* bytes to the corresponding binary string of length *n* bytes, you invoke the BS-BI$ routine with a call of the form:

CALL BS-BI$ USING *bitstring binary n*

where *bitstring* labels the area containing the bit string, read-only as far as BS-BI$ is concerned, *binary* labels the area in which the binary string will be created, and *n* is either the name of a PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of the binary string in bytes.

Exception condition 1 will be returned if any byte from the bit string area contains a value which is not ASCII 0 or ASCII 1. If this exception occurs the contents of the binary string area will be unpredictable.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is not a positive integer.

3.8.3 Examples

In the examples below Z-BI is a PIC 9(4) COMP variable (occupying two bytes) and Z-BS is a PIC X(16) variable used to hold the equivalent bit string.

The following statements accept 16 ASCII characters into the Z-BS field and convert them to binary in the Z-BI field. If the input is invalid, the operator is re-prompted until the correct input is supplied:

AA010.

```
DISPLAY "INPUT 16-CHARACTER BIT STRING VALUE"
ACCEPT Z-BS
CALL BS-BI$ USING Z-BS Z-BI 2
ON EXCEPTION GO TO AA010
```

The following statements convert the binary string in Z-BI to an ASCII bit string in Z-BS and then display the result:

```
CALL BI-BS$ USING Z-BI Z-BS 2
DISPLAY Z-BS
```

3.9 Binary Word String to Octal String Conversion, BI-OC$ & OC-BI$

A routine is provided to convert a string of binary words to an octal string and vice versa. By an octal string we mean a sequence of a multiple of 6 ASCII characters, each of which is in the range "0" to "7". For example, the octal string "177600" corresponds to a binary string two bytes in length, containing the bits:

```
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
```

The format of the words of the binary string (i.e. which of the two bytes of the word is the most significant) can be specified when the routine is called, and thus it can operate on a list of addresses of either format.

3.9.1 To Convert a Binary Word String to an Octal String

To convert a string of *n* binary words to the corresponding octal string of length 6*n* bytes, you invoke the BI-OC$ routine with a call of the form:

```
CALL BI-OC$ USING binary octal n
```

where *binary* labels the area containing the binary string, read-only as far as BI-OC$ is concerned, *octal* labels the area in which the octal string will be created, and *n* is either the name of a PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of the binary string in words. The format of the binary words is specified by the sign of the value n. A positive value means that the junior byte is the most significant (Global Cobol pointer format), while a negative value means that the senior byte is the most significant. Each 6-byte octal value created has leading "0" bytes if necessary.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is zero.

3.9.2 To Convert an Octal String to a Binary Word String

To convert an octal string of length 6*n* bytes to the corresponding string of *n* binary words, you invoke the OC-BI$ routine with a call of the form:

```
CALL OC-BI$ USING octal binary n
```

where *octal* labels the area containing the octal string, read-only as far as OC-BI$ is concerned, *binary* labels the area in which the binary string will be created, and *n* is either the name of a

PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of the binary string in words. The format of the binary words is specified by the sign of the value n. A positive value means that the junior byte is the most significant (Global Cobol pointer format), while a negative value means that the senior byte is the most significant. Each 6-byte octal value can have either leading spaces or zeros if required.

Exception condition 1 will be returned if any byte from the octal string area contains a value which is not ASCII 0-7, or if any set of 6 bytes converts to a value which is greater than 177777 (octal).

If this exception occurs the contents of the binary string area will be unpredictable. Note that this means that any octal string to be converted must **not** contain leading or trailing spaces. Thus short strings must be padded to the correct number of characters by the insertion of the appropriate number of rightmost zeros.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is zero.

3.9.3 Examples

In the following example, Z-OC is a 6-byte area into which an ASCII octal string is accepted and then converted to binary in the 2-byte area Z-BI. Note that the binary value in Z-BI is not in Global Cobol pointer format. If the input is invalid, the operator is re-prompted until the correct input is supplied:

```
AA010.
        DISPLAY "INPUT OCTAL VALUE"
        ACCEPT Z-OC
        CALL OC-BI$ USING Z-OC Z-BI -1
        ON EXCEPTION GO TO AA010
```

In the next example, a binary string held in the two-byte area Z-BI is converted to ASCII octal in the 6-byte area Z-OC. This time the binary value in Z-BI is in Global Cobol pointer format:

```
        CALL BI-OC$ USING Z-BI Z-OC 1
        DISPLAY Z-OC
```

3.10 ASCII to RADIX-50 Conversion, AS-RL$ & RL-AS$

A routine is provided to convert an ASCII character string to a string of RADIX-50 words and vice versa. RADIX-50 describes a compact method of character storage, often used by DEC operating systems, which enables a limited character set (of 40 characters (i.e. octal 50 characters) hence the name) to be held 3 characters per 2-byte word. The table below gives the RADIX-50 values for the characters of the set.

Character	*RADIX-50 value (decimal)*
Space	0
A-Z	1-26
$	27
.	28

?	29
0-9	30-39

For example, the ASCII character string "ABC" would be represented by the 2-byte RADIX-50 value:

1*40*40 + 2*40 + 3 = 1683 (decimal arithmetic)

The format of the RADIX-50 word (i.e. which of the two bytes of the word is the most significant) can be specified when the routine is called.

3.10.1 To Convert an ASCII String to a String of RADIX-50 Words

To convert an ASCII string of length 3*n* bytes to the corresponding RADIX-50 string of length *n* words, you invoke the AS-RL$ routine with a call of the form:

CALL AS-RL$ USING *ascii radix-50 n*

where *ascii* labels the area containing the ASCII string, read-only as far as AS-RL$ is concerned, *radix-50* labels the area in which the RADIX-50 string will be created, and *n* is either the name of a PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of the RADIX-50 string in words. The format of the RADIX-50 words is specified by the sign of the value *n*. A positive value means that the junior byte is the most significant (Global Cobol pointer format), while a negative value means that the senior byte is the most significant.

Exception condition 1 will be returned if any byte from the ASCII string area contains a value which does not have a RADIX-50 equivalent as defined in the table above. If this exception occurs, all undefined ASCII characters will have been converted into RADIX-50 "?" characters.

Your program will be terminated with a stop code if the value of *n* that you supply to the routine is zero.

3.10.2 To Convert a String of RADIX-50 Words to an ASCII String

To convert a RADIX-50 string of *n* words to the corresponding ASCII string of 3*n* bytes, you invoke the RL-AS$ routine with a call of the form:

CALL RL-AS$ USING *radix-50 ascii n*

where *radix-50* labels the area containing the RADIX-50 string, read-only as far as RL-AS$ is concerned, *ascii* labels the area in which the ASCII string will be created, and *n* is either the name of PIC 9(4) COMP field, or an integer literal, specifying the non-zero length of the RADIX-50 string in words. The format of the RADIX-50 words is specified by the sign of the value *n*. A positive value means that the junior byte is the most significant (Global Cobol pointer format), while a negative value means that the senior byte is the most significant.

Exception condition 1 will be returned if any word of the RADIX-50 string has a value greater than 63999 (decimal). If this exception occurs the contents of the ASCII string area will be unpredictable.

Your job will be terminated with a stop code if the value of *n* that you supply to the routine is zero.

3.10.3 Examples

In the following example, Z-AS is a PIC X(6) field into which an ASCII string is accepted and then converted to RADIX-50 form in the PIC X(4) field Z-RL. Note that the two RADIX-50 words created in Z-RL are not in Global Cobol pointer format. If the input is invalid, the operator is re-prompted until the correct input is supplied:

```
AA010.
        DISPLAY "INPUT ASCII STRING"
        ACCEPT Z-AS
        CALL AS-RL$ USING Z-AS Z-RL -2
        ON EXCEPTION GO TO AA010
```

In the following example, a pair of RADIX-50 words held in the PIC X(4) field Z-RL is converted to ASCII in the PIC X(6) field Z-AS. This time the RADIX-50 format of Z-RL is Global Cobol pointer format:

```
        CALL RL-AS$ USING Z-RL Z-AS 2
        DISPLAY Z-AS
```

3.11 Numeric, K, M or G String to Computational Conversion, NKM-C$

The NKM-C$ routine converts a 16-character string in any of the following formats:

number (A)

number K (B)

number M (C)

number G (D)

into a PIC S9(15) COMP signed integer. In format (A) the number is treated as an S9(15) display numeric quantity. In format (B) it is S9(11,3), in format (C) it is S9(8,6) and in format D it is S9(4,6). The suffix K is considered to represent a 1024 multiplier; M a 1048576 (i.e. 1024 x 1024) multiplier; and G a 1073741824 (i.e. 1024 x 1024 x 1024) multiplier. During the calculations a fractional result is truncated towards zero, not rounded. Hence the following strings all convert into the value -256:

```
-256.8
-.25K
-.000245M
```

The routine is used to normalise input when the operator is allowed to specify a quantity, such as a file size, either in bytes, kilobytes, megabytes or gigabytes.

3.11.1 Invocation

To convert a numeric, K, M, or G string to computational format code:

```
CALL NKM-C$ USING string comp
```

where *string* must label a 16-byte area containing the string to be converted, padded with rightmost spaces as necessary. The quantity *comp* is the name of a PIC S9(15) COMP variable to contain the result of the conversion.

3.11.2 Exception Conditions

Exception condition 1 will be returned if the format of the string is invalid, or if the result of the conversion would exceed the capacity of a PIC S9(15) field. In this case the comp field itself remains unchanged.

3.11.3 Example

In the following example, Z-STR is a PIC X(16) field into which a numeric, K, M, or G string is accepted and then converted to computational form in the PIC S9(15) field Z-COMP. If the input is invalid, the operator is re-prompted until the correct input is supplied:

```
AA010.
    DISPLAY "INPUT FILE SIZE"
    ACCEPT Z-STR
    CALL NKM-C$ USING Z-STR Z-COMP
    ON EXCEPTION GO TO AA010
```

3.12 Double Length Multiply and Divide, MULTI$ and DIVID$

The double length Multiply and Divide routines allow the accurate multiplication and division of PIC 9(12,6) COMP numbers. Whereas the normal MULTIPLY and DIVIDE statements can overflow in various circumstances (such as when the result contains more than 9 digits), these subroutines will only overflow if the result exceeds the capacity of a PIC 9(12,6) COMP field.

3.12.1 Double Length Multiplication

The MULTI$ system subroutine is invoked by a CALL statement of the form:

```
CALL MULTI$ USING A B C
```

where *A* and *B* are the numbers you wish to multiply together and *C* is the result. *A*, *B* and *C* must all be defined as PIC 9(12,6) COMP variables.

3.12.2 Double Length Division

The DIVID$ system subroutine is invoked by a CALL statement of the form:

```
CALL DIVID$ USING A B C
```

where *A* is the divisor, *B* is the dividend and *C* is the result.

3.12.3 Exception

Exception condition 1 will be returned if an overflow occurs and C will be set to the largest negative value that can be contained in the PIC 9(12,6) format (i.e. the number represented by the 8 bytes #8000000000000000).

3.12.4 Programming Notes

The quantities *A*, *B* and *C* do not need to be distinct. For example:

```
CALL MULTI$ USING VALUE VALUE VALUE
```

replaces the quantity in VALUE by its square.

3.13 The Password Number Routine, PWNUM$

The password number routine is used to calculate a password number from a user password string. This string is scrambled so that it no longer appears as ASCII, although it still represents the same password as far as the PWNUM$ and PWCHK$ routines are concerned. If you need to retain a copy of the user password then you should retain the scrambled version.

3.13.1 Invocation

The password number routine is invoked with a call of the form:

```
CALL PWNUM$ USING input-string pwnumber
```

input-string is a PIC X(8) variable containing the user password with which the file is to be protected. This is updated by the scrambling process. *pwnumber* is the PIC S9(9) COMP variable in which the encrypted user password is returned.

3.13.2 Programming notes

The password number, **not** the scrambled input string, is saved on the file to protect it. The encryption technique ensures that it is virtually impossible to deduce the user password corresponding to a particular number even if you decode the instructions of PWNUM$. However, only a very simple algorithm is used for scrambling the password string since it is merely intended to prevent temporarily retained passwords standing out in an ASCII dump. The whole point is to rely on encrypted password numbers rather than strings. You should normally call PWNUM$ just as soon as the operator has keyed the password, as in the following example:

```
Ask operator if this file is to be password protected.
If so
        Prompt for password, accepting it into input-string.
        CALL PWNUM$ USING input-string pwnumber
Else (file to be marked unprotected)
        CALL PWNUL$ USING tagword pwnumber       * see 3.14 below
End
```

At the end of this process, *pwnumber* either contains a user password number or the null password number whose value depends on the *tagword*. If the operator did protect the file then the user password number will have been set up and the input string scrambled so that the keyed ASCII is no longer visible.

3.14 The Null Password Number Routine, PWNUL$

The null password number routine is used to set up a special value of the password number taken to mean that the file is unprotected. The null value is a function of the tag-word, a volatile

and critical field preserved on the file together with the password number. The routine is used to avoid there being a single, system-wide null value that can easily be found and patched into files.

3.14.1 Invocation

The null password number routine is invoked with a call of the form:

```
CALL PWNUL$ USING tagword pwnumber
```

tagword is the name of a PIC S9(4) COMP variable containing the tag-word value. This is read only as far as PWNUL$ is concerned. *pwnumber* is the PIC S9(9) COMP variable in which the null password is returned.

3.14.2 Programming note

Section 3.13.2 shows how PWNUL$ might be used in conjunction with PWNUM$ at the time the operator is deciding whether or not to protect the file. You might want to call PWNUL$ before then and only use it at that time if the operator requested protection to be removed from a file currently protected.

You need to use PWNUL$ again when the tag-word is about to change, normally just before the file is updated on disk. Using the expanded subroutine calls, the earlier example becomes:

```
CALL PWNUL$ USING tagword N                    * Save current null pw#
Change tagword.
IF N = pwnumber                                *If file unprotected
        CALL PWNUL$ USING tagword pwnumber     * Set new null pw#
END
Write file with new tag-word (and, if unprotected, new null password number value).
```

3.15 The Password Check Routine, PWCHK$

The password check routine can be used to perform all or certain combinations of the following tests, depending on how its four parameters are set up:

- Does the current password number contain the special null value, indicating that the file is unprotected?
- Does the input string contain the user password?
- Does the input string contain the master password?

The routine completes normally if any selected test succeeds, otherwise it returns an exception.

The master password, mentioned above, is a special password, obtainable from Global System Manager, which can be used to grant access to a password protected file when the original password has been lost. The master password is based on the name of the file, the date on which it is to be used and the installation where it will be used. Normally you would use the master password to go into and then change or remove the original password.

3.15.1 Invocation

(a) To perform all three tests, code:

CALL PWCHK$ USING *file input-string tagword pwnumber*

(b) To check just whether the null password value is set, code:

CALL PWCHK$ USING #80 #80 *tagword pwnumber*

(c) To check just for the master password, code:

CALL PWCHK$ USING *file input-string* #80 #80

file is a PIC X(8) variable (or literal) containing the file-id of the protected file, or #80 if master password checking is not required. It is read-only as far as PWCHK$ is concerned.

The *input-string* is a PIC X(8) variable containing the password originally keyed by the operator (or a scrambled version of that password). PWCHK$ updates the string to the scrambled non-ASCII format if this has not already taken place. The variable may be replaced by #80 if the routine is only being used to check for the null password number. Note that input strings beginning with #80 can never form a valid user or master password and are never scrambled.

The *tagword* is the name of a PIC S9(4) COMP variable containing the tag-word value. It is read-only as far as PWCHK$ is concerned. It and the following *pwnumber* parameter should be replaced by #80 if the routine is used simply to check for the master password.

pwnumber is the PIC S9(9) COMP variable containing the password number as previously established by either the PWNUM$ or PWNUL$ routines. It is read-only as far as PWCHK$ is concerned. Along with the preceding tag-word parameter it should be replaced by #80 if the routine is being used just to check for the master password.

3.15.2 Exception Conditions

Exception condition 1 is returned if no test specified by the parameters succeeds. For a form (a) call this means that the file is protected and the operator has failed to supply either the correct master password or user password. For form (b) an exception indicates that the file is protected and in consequence the operator must be prompted for its password. A form (c) exception means that the operator has failed to supply the master password.

3.15.3 Programming notes

Form (c) is provided for use only by those applications which wish to employ the new master password checking without adopting the rest of the scheme.

A typical authorisation routine uses form (b) to see if the password is null - and if this is so, does not prompt the operator as the file is unprotected. If a password is needed the form (a) is employed to check it once it has been keyed. Expanding the PWCHK$ calls the skeleton authorisation routine and the logic becomes:

```
CALL PWCHK$ USING #80 #80 tagword pwnumber
ON NO EXCEPTION EXIT                    * Out if null password set
Prompt operator for 8 character password.
Accept into input-string
```

```
CALL PWCHK$ USING file input-string tagword pwnumber
ON EXCEPTION EXIT WITH $$CODE          * Reflect wrong password
EXIT                                   * Good exit if master/user pw
```

The quantities *file*, *input-string*, *tagword* and *pwnumber* should be located outside the authorisation routine in order to be made available to the verification process. The *input-string* is scrambled by the time the good exit from the authorisation routine takes place, so that a valid password cannot be seen from another partition. *input-string* should be set to start with a #80 byte before the authorisation routine is called so that if the routine is patched out the password string supplied to the verification routine is guaranteed to be invalid.

3.15.4 Verification processing

Often the entire authorisation process can be disabled by patching the code that prompts for the password to an EXIT statement. This is such a simple process that there is no need to patch the product on disk, so the chance of such tampering being detected is minimal.

The answer to this problem does not lie in making the authorisation routine unnecessarily complicated. Instead, the routine should preserve the 8 character password keyed by the operator together with the filename, tag-word and password number, so that they can be passed to the PWCHK$ routine again by a later verification process. This should be located in a different overlay from the authorisation routine and should always be entered shortly after the authorisation routine has been used.

The verification process itself simply repeats the form (a) call used by the authorisation routine. PWCHK$ will only return an exception if the authorisation routine has not completed properly. Typically, verification involves just two lines of code:

```
CALL PWCHK$ USING file input-string tagword pwnumber
ON EXCEPTION STOP WITH -99
```

4. Program Management Subroutines

4.1 The Customisation Routine, CUST$

The CUST$ system routine is used to write the program last loaded from main memory back to external direct access storage. In this way, if the program has modified itself since it was loaded, any updates will be permanently remembered. Global System Manager uses CUST$ to write back a modified version of the $STARB command program whenever Global System Manager customisation takes place.

4.1.1 Invocation

The customisation routine is invoked by a parameter-less call:

```
CALL CUST$
```

It only returns control when the program has been written successfully to direct access storage. Your job will be terminated with a stop code if an irrecoverable I/O error occurs, or if the program (whose program-id is assumed to be contained in $$PGM, set when it was loaded) cannot be found.

4.1.2 Programming Notes

Although CUST$ is very convenient for parameterising test and other one-off programs, it is best avoided for live programs as it can lead to obscure problems when a variable is accidentally rewritten with an unexpected value in it.

Normally programs which use CUST$ provide a special sequence of prompts, not usually employed in ordinary working, which allow the operator to specify infrequently-varied parameters. Each parameter is verified, and its value is saved in **initialised** working storage. (Any uninitialised working storage at the start of data division does not form part of the program file, and cannot be used for customisation.) Once the special sequence of prompts is complete, the program memory area is written back to direct access storage, overwriting the original program file. The parameters established by the special prompt sequence will therefore be available when the program is subsequently run, and will remain unchanged unless the operator decides to customise the program anew.

You must be careful not to be in the middle of any processing when you call CUST$. In particular, you should not:

- have any open FD's;
- have called JOB$ to pass dialogue;
- be in the middle of processing a MAPIN statement (i.e. in a validation routine).

This is because information about these situations would be "remembered" as part of the customized program and would cause unpredictable errors when it was next run.

4.2 Checking for Online Programs and Libraries, PROG$

You may use the PROG$ system routine to determine whether a particular program or library is available on the current system or program residence device, without displaying a program required prompt if it is not found. If the program is in a library then this library must be attached, otherwise the program will be considered to be unavailable.

4.2.1 Invocation

You determine whether a program or library is online by means of a CALL statement of the form:

CALL PROG$ USING *request* [*volume-id*] [*unit-id*]

where *request* is an 8-character literal or variable containing the program-id or library-id of the program or library to be checked. The optional second parameter, *volume-id*, is only used when a program request is made. It is the name of a PIC X(6) variable in which the volume-id identifying the location of the actual member will be returned if the portion of a dispersed library which is currently online contains only a stub for the requested program. If you include the second parameter you can also include an optional third parameter, *unit-id*. This is the name of a PIC X(3) variable in which the unit-id for the actual member for a stub in the current library will be returned.

Providing PROG$ returns normal completion following a program request, $$PGM is set to the specified program-id. When a library request is similarly honoured the library itself is attached and its library-id is placed in $$LIB.

4.2.2 Processing

If the program or library requested is not currently online an exception is returned. In addition, if the second parameter is supplied and the online library contains a stub for the program requested, the volume-id of the required volume will be returned in the second parameter, but the volume itself will **not** be mounted. If the actual program is online the volume-id will be returned as spaces.

4.2.3 Exceptions

Exception condition 1 will be returned if an irrecoverable I/O error occurs.

Exception condition 2 will be returned if the program or library is not present. $$RES will be "3" if it is genuinely missing or "1" if there is a file with a matching name on the device, but that file is not a program or library.

4.2.4 Programming Notes

If you are using free space management the index area pointer, $$INDE, must be established before calling PROG$. See section 6.29.

If the program name supplied starts with a "$" then the system residence device will be searched, otherwise the program residence device, assigned to $P, will be searched. If the program name starts "*" then "$" will be substituted when searching for the program on $P.

4.3 Quick Overlay Loading Using QINDX$ and QLOAD$

When a program library is attached, Global System Manager keeps it permanently open to reduce overlay time. Nevertheless a program load resulting from an EXEC or LOAD statement

always involves at least two distinct read operations, one for the library index, and one or more for the module itself. This section describes how load time can be reduced by constructing a resident index using the QINDX$ (quick index) routine, and then supplying this index to the QLOAD$ (quick load) routine, so that programs defined in the index can be brought into memory by a single read operation. The technique is used for the Global Cobol compiler and Global Writer, and is recommended for any application involving intensive overlay handling.

4.3.1 Contiguously Initialised Program Overlays

A program can only be quick loaded if it has been contiguously initialised to enable it to be read into memory in a single operation. For an independent program this means that:

- The first byte of the second module, and all subsequent modules, is initialised;
- No module included in the linkage edit defines an FD with a BLOCK CONTAINS statement in working storage.

A dependent program is contiguously initialised if the above rules hold for the link list defining the dependent program itself. Its information overlays need not be taken into account.

The first byte of a routine is initialised if the first item of working storage is an FD or MD, or it has a value clause associated with it. If there are no items in working storage, then the first byte is always initialised.

All Global Cobol system routines except the SAVE$ routine have their first byte initialised and none has a BLOCK CONTAINS statement in working storage, so they need not be considered when designing for contiguous initialisation.

The link map listing output by $LINK indicates the number of load records the resultant program file contains, and this value will be 1 if the program has been contiguously initialised.

(You should note that contiguous initialisation will speed program loading in nearly all cases even when using normal LOAD and EXEC statements. The only time it should be avoided is when the program contains a very large interior area which can be left non-initialised, typically more than 4 Kbytes in length.)

4.3.2 Building the QI block

The system routine QINDX$ is called to build a quick index from the currently attached program library. You code a CALL statement of the form:

CALL QINDX$ USING *QI length*

where *QI* identifies the area where the quick index is to be built and *length* is an integer literal or PIC 9(4) COMP variable containing its length in bytes.

As explained later, a quick index consists of 16 bytes of header/trailer information together with a 17 byte entry for each contiguously initialised program contained in the attached library. An exception is returned if QINDX$ finds that the area length is insufficient for the library index. The maximum index, required if all 100 programs of a maximum library are contiguously initialised and thus eligible for inclusion, is 1716 bytes in length.

Your program will be terminated with a stop code if no library is attached when QINDX$ is called. Therefore, if there is any doubt whether the correct library is attached you should precede the call with the statement:

```
LOAD library-id
```

The QINDX$ routine returns exception condition 1 if an irrecoverable I/O error occurs when accessing the library. Exception 2 is returned if the attached library is not on line. Exception 3 means that the quick index was too large for the specified QI area length.

4.3.3 Quick loading a Program via the Quick Index

To quick load a contiguously initialised program using the previously constructed quick index, set $$PGM to the required program-id and then execute a CALL statement of the form:

```
CALL QLOAD$ USING QI
```

where *QI* identifies the quick index for the currently attached library. The routine brings the program into memory as though a LOAD statement had been issued, returning the entry point in $$EPT. Thus:

```
LOAD "SALES"
```

may be replaced by:

```
MOVE "SALES" TO $$PGM
CALL QLOAD$ USING QI
```

Similarly:

```
EXEC "SA100"
```

becomes:

```
MOVE "SA100" TO $$PGM
CALL QLOAD$ USING QI
CALL $$EPT
```

Your program will be terminated with a stop code if the library attached when QLOAD$ was executed was not the same as the one used when the quick index was built. QLOAD$ returns exception condition 1 if an irrecoverable error occurs when accessing the program library. Exception 2 is returned if the requested program-id is not present in the quick index. Exception condition 3 means the program was too large for the available user area.

```
PROGRAM EXAMPLE
DATA DIVISION
* QUICK INDEX
01      Q1
  03    QILIB  PIC X(8)
  03    QIP OCCURS 100
    05  QIPID
```

```
    07  QIPIDP      PIC X(2)
    07  QIPIDS      PIC X(6)
  05    FILLER      PIC X(9)
 03     QIT         PIC X(8)
*
77      Z-INX1      PIC 9(4) COMP
77      Z-INX2      PIC 9(4) COMP
*
PROCEDURE DIVISION
        MOVE "EXVOL" TO $$PVOL
        LOAD "P.EX"
        CALL QINDX$ USING QI 1716
        MOVE 1 TO Z-INX1 Z-INX2
        DO UNTIL QIPID(Z-INX1) = LOW-VALUES
                IF QIPIDP(Z-INX1) = "FP"
                        MOVE QIPID(Z-INX1) TO QIPID(Z-INX2)
                        ADD 1 TO Z-INX2
                END
                ADD 1 TO Z-INX1
        ENDDO
        MOVE LOW-VALUES TO QIPID(Z-INX2)
AA500.
        DISPLAY "PLEASE KEY PROGRAM-ID"
        ACCEPT $$PGM
        CALL QLOAD$ USING QI
        ON EXCEPTION
                IF $$COND NOT = 2 GO TO AA600
                LOAD PG
                ON EXCEPTION
AA600.
                        GO TO DEPENDING ON $$COND
                        TO AA700
                        TO AA800
                        TO AA900
                END
        END
        CALL $$EPT
        GO TO AA500
AA700.
        BELL
        DISPLAY "I/O ERROR" SAMELINE
        GO TO AA500
AA800.
        BELL
        DISPLAY "NOT FOUND" SAMELINE
        GO TO AA500
AA900.
        BELL
        DISPLAY "TOO LARGE" SAMELINE
        GO TO AA500
```

```
ENDPROG
```

Figure 4.3.5 - Example Program using Quick Load

4.3.4 Quick Index Format

The following QI block defines a maximum size quick index which contains information for 100 contiguously initialised programs:

```
01    QI
  03  QILIB          PIC X(8)          * LIBRARY-ID
  03  QIP OCCURS 100                   * 0 TO 100 ENTRIES
    05  QIPID        PIC X(8)          * PROGRAM-ID
    05  QIPFAD       PIC 9(6) COMP     * FILE ADDRESS OF RECORD
    05  QIPBLO       PIC 9(4) COMP     * LENGTH OF RECORD
    05  QIPMEM       PIC PTR           * MEMORY ADDRESS
    05  QIPSTA       PIC PTR           * ENTRY POINT
  03  QIT            PIC X(8)          * LOW-VALUES TERMINATOR
```

When there are 99 or fewer contiguously initialised programs then the QIPID field of the first unused entry is set to LOW-VALUES and the remainder of the index area is unused. The length of the quick index in bytes is therefore:

```
16 + 17 * Number of programs in index
```

The QI block may be shortened by removing unwanted entries before calling QLOAD$. This can be valuable in memory-critical applications where storage not required by the quick index can be employed for other purposes.

4.3.5 Example - Editing and Using the Quick Index

The example program in Figure 4.3.5 builds a quick index for library P.EX on volume EXVOL. It then edits this index to contain only program-ids beginning with the letters "FP". The example then repeatedly prompts the operator for program-ids. As each one is supplied it is either quick loaded using QLOAD$, or brought into memory using a conventional LOAD statement if the routine returns exception condition 2.

4.3.6 Programming Notes

The program library must be online when an attempt is made to load a program using QLOAD$, since to optimise performance QLOAD$ does not check that the correct library is mounted each time it is called. If the library volume is replaced by another the quick load will cause erroneous data to be loaded into memory in place of the program required.

You should not confuse the library index addressed by $$INDE with the quick index used by QLOAD$. The two indexes may be used in conjunction in applications where the time taken to build the quick index is too long for QINDX$ to be executed every time the system is run. In this case the quick index should be kept on file along with a copy of the library index. Then, to prevent inadvertent use of an out-of-date quick index, compare the file copy of the library index with the current library index supplied in the area of your program addressed by $$INDE following the LOAD library-id statement. Only if the two library indexes are identical can you

guarantee that the file quick index is still valid. If they differ library maintenance has taken place since the time the quick index was built, so you must reconstruct it using QINDX$.

4.4 The Relocatable Loader, LOAD$

The LOAD$ system routine must be used to load a relocatable program created by FCONV or $RELOC, since such a program cannot be loaded by conventional CHAIN, EXEC, RUN or LOAD statements. (LOAD$ cannot be used on ordinary, non-relocatable programs created by the linker.)

4.4.1 Invocation

You load a relocatable program by means of a call of the form:

CALL LOAD$ USING *name type*

where *name* is a PIC X(8) variable or 8-character literal containing the program-id of the relocatable program. If the program-id is less than 8 characters it must be padded with rightmost blanks.

The second parameter, *type,* is a single character variable or literal, which indicates which stack the program is to occupy, and in the case of the user stack, whether it is a temporary program to be automatically unloaded when control returns to the monitor. The type may be:

"S" the program is to be loaded on the system stack;

"U" the program is to be loaded on the user stack;

"T" the program is to be temporarily loaded on the user stack, but is to be automatically unloaded when control returns to the monitor.

Providing the load operation is successful system variable $$EPT will be set to address the entry point of the program once LOAD$ returns control. $$PGM will contain the program-id you specified and $$RUN will be set to 0. If the type specified was "T" or "U" the library index area pointer, $$INDE, will be set to address an area 1134 bytes below the current top of the user stack.

4.4.2 Processing

If the program requested is not already present on the appropriate stack, then providing there is room it will be loaded. If there is a sufficiently large gap in the stack created by a previous unload operation, this will be used, otherwise the program will be added to the top of the stack, which will therefore be extended downwards to begin at a lower memory address.

If a program with a program-id matching the name specified in the LOAD$ call is already on the stack LOAD$ satisfies the request by returning the entry point of this program.

When loading is required, processing is similar to that performed by a LOAD statement. If the name begins with a $-character, firstly P.$CMLB0 and then individual files on the system residence device assigned to $CP are examined. Otherwise the attached library, if any, together with individual files on the device assigned to $P are searched. In the special case when the

first character of the name is an asterisk, the $P device is searched for a program-id beginning with a $-character.

4.4.3 Exceptions

Exception condition 1 will be returned if an irrecoverable I/O error arises when physically loading the requested program.

Exception condition 2 will occur if the program cannot be found, or if it is not a relocatable program. The result code, $$RES, will be "3" if the program is not found, or "1" if it is of the wrong type.

Exception condition 3 will be returned if there is insufficient memory space available on the stack to allow the program to be loaded.

4.4.4 Programming Notes

LOAD$ always accesses the program residence device to read header information, so if you want to avoid this overhead when the program is already on the stack you should only use LOAD$ when a previous ENTRY$ has indicated it is not present.

Because the program-id is used to indicate the presence of a program on a stack, to avoid confusion you should ensure that the program-ids you assign to relocatable programs are unique across your entire system. Since an existing copy of a program is always used if it is loaded, you must take care either to remove the program from the stack by unloading it, so that you obtain a fresh copy, or to code it in such a way that it is re-usable. Indeed, programs on the system stack may be shared by a number of users "simultaneously". However, a particular invocation of such a program cannot be interleaved with a second invocation, so it is only necessary to ensure that the code is serially re-usable between calls. This can be achieved by holding status data in the caller's user area and assuring that any local variables are temporary work fields with no assumed initial values.

To be certain of obtaining a fresh copy of a program on the user stack you should call the UNLO$ system routine to unload it if it is already there, and then reload it using LOAD$. This technique will not, however, guarantee a fresh copy of a system stack program, since the UNLO$ call will not remove it from the stack if it is currently loaded by any other user. If such a program was initially loaded by a type S call on LOAD$ it will only be removed when all its users have explicitly unloaded it using UNLO$. If, however, the program was placed on the system stack by the $CUS Permanently Loaded Modules option it is treated as a logical extension of the Global System Manager nucleus, and cannot be unloaded in any circumstances. Normally, therefore, programs destined for the system stack should be serially reusable between calls, as explained above.

If you load a program on the user stack a number of times, treating it sometimes as type T and sometimes as type U, then it will be temporary or permanent according to the type code you specified in the first call on LOAD$.

Even if a program is loaded as temporary, it is a good practice to unload it explicitly using UNLO$ (although its temporary status will ensure it is unloaded if the program terminates abnormally).

4.5 The Allocate Data on Stack Routine, SDATA$

The SDATA$ system routine will allocate a data area of a requested size on either the user stack or system stack.

4.5.1 Invocation

You allocate a data area on the stack by means of a CALL of the form:

CALL SDATA$ USING *name size type*

where *name* is a PIC X(8) variable or 8-character literal containing the name of the data area to be allocated. The second parameter, *size*, is a PIC 9(4) COMP variable or integer literal containing the size of the area to be allocated in bytes.

The third parameter, *type*, is a single character variable or literal, which indicates which stack the area is to occupy, and in the case of the user stack, whether it is a temporary area to be automatically unloaded when control returns to the monitor. The type may be:

"S" the area is to be allocated on the system stack;

"U" the area is to be allocated on the user stack;

"T" the area is to be temporarily allocated on the user stack, but is to be automatically unloaded when control returns to the monitor.

Providing the area can be allocated successfully system variable $$EPT will be set to address the start of the allocated area when SDATA$ returns control. If the type specified was "T" or "U" the library index area pointer, $$INDE, will be set to address an area 1134 bytes below the current top of the user stack.

4.5.2 Processing

If a stack entry with the supplied name is not already present on the appropriate stack, then providing there is room an area of the appropriate size (rounded up if odd) is allocated, and initialised to binary zeros. If there is a sufficiently large gap in the stack created by a previous unload operation this will be used, otherwise the stack will be extended downwards.

If a data area already exists with the same name, and of the appropriate size, then the start of this area will be returned (but the area is not initialised to binary zeros).

If a stack entry exists which has the same name, but is either the wrong size or is a program rather than a data area, your program will be terminated in error.

4.5.3 Exceptions

Exception condition 3 will be returned if there is insufficient space available to allow the area to be allocated.

4.5.4 Programming Notes

If you allocate a data area on the user stack a number of times, treating it sometimes as type T and sometimes as type U, then it will be temporary or permanent according to the type code specified when it was first allocated.

If you allocate an area on the system stack which has already been allocated by another user, an extra 16 byte data block is created on the stack to represent your usage of the program. The area returned to you will contain any values established by the other user.

4.6 The Entry Point Routine, ENTRY$

You use the ENTRY$ routine to determine whether a named stack entry already exists for the current user, and to return its entry point (if it is a program) or its start (if it is a data area).

4.6.1 Invocation

You determine whether a stack entry already exists by means of a call of the form:

CALL ENTRY$ USING *name type*

where *name* is a PIC X(8) variable or 8-character literal containing the program-id of the program or name of the data area.

The second parameter, *type*, is a single character variable or literal which must have the value "S" for the system stack or "T" or "U" for the user stack.

4.6.2 Processing

If the type parameter is "T" or "U" the routine searches the user stack to see if the requested stack entry has previously been loaded. If it has the entry point or start address is returned in $$EPT, otherwise exception condition 1 is returned.

If the type parameter is "S" the system stack is examined, but this time the search only succeeds if the stack entry is present **and** it has been loaded by the current user, or as a result of the $CUS Permanently Loaded Modules option. (Programs introduced by LOAD are treated as logical extensions of the Global System Manager nucleus and, as such, are considered to belong to every user.)

Note that if the first character of the program-id specified in the name parameter is an asterisk, then it will be treated as though a $-character had been supplied in its place when searching the stack.

4.6.3 Exceptions

Exception condition 1 will be returned if the requested stack entry has not been loaded by the current user or by the $CUS Permanently Loaded Modules option.

4.7 The Unload Routine, UNLO$

You use the UNLO$ system routine to unload a program or de-allocate a data area from the system or user stack once you have finished using it.

4.7.1 Invocation

You unload a relocatable program or de-allocate a data area from the stack by means of a call of the form:

CALL UNLO$ USING *name type*

where *name* is a PIC X(8) variable or 8-character literal containing the name of the stack entry.

The second parameter, *type*, is a single character variable or literal which must have the value "S" for the system stack, or "T" or "U" for the user stack.

If the type specified was "T" or "U" the library index area pointer, $$INDE, will be set to address an area 1134 bytes below the current top of the user stack.

4.7.2 Processing

Providing the named entry is present on the stack selected by the type parameter it will be unloaded or de-allocated, as appropriate. The stack will only be contracted if the entry is at the top, and, in the case of the system stack, is not currently loaded by another user. When the stack cannot be contracted, but there are no other users, the entry is logically deleted to create a gap in the stack which can be used by a subsequent LOAD$ or SDATA$ call.

If the first character of the program-id specified in the name parameter is an asterisk, then it will be treated as though a $-character had been supplied in its place when searching the stack.

4.7.3 Exceptions

Exception condition 1 will be returned if the stack entry specified in the name parameter is not currently loaded by the user who called UNLO$.

4.7.4 Programming Note

A type "S" UNLO$ call which attempts to unload a program which was introduced on the system stack as the result of the $CUS Permanently Loaded Modules option has no effect.

Temporary user stack entries are automatically unloaded at the end of job. However, since this requires access to a special command library overlay, you may wish to avoid the need to mount a system volume when your job completes normally by explicitly unloading any temporary entries using UNLO$.

5. System Management Subroutines

5.1 The Exclusive Control Routine, GETX$, GETXN$ & RELX$

The exclusive control routine is a system routine with three entry points, GETX$, GETXN$ and RELX$. They enable a job operating in a multi-user or networking environment to acquire or relinquish exclusive control of its processor, in which state the job cannot be swapped out. Calls on GETX$, GETXN$ and RELX$ have no effect under a single-user system, or when the job is the only possible user of its processor in a LAN system.

5.1.1 To Acquire Exclusive Control

A job can acquire exclusive control of its processor by executing the statement:

```
CALL GETX$
```

Normal completion will be returned if exclusive control has been granted. The routine also causes the swap file to be closed, so that the volume containing it can be accessed using the Physical Sector Access Method (see chapter 8 of the Global Development File Management Manual). If the swap file cannot be closed, an exception will be returned. If you do not need to close the swap file, you should instead execute the statement:

```
CALL GETXN$
```

which obtains exclusive control without closing the swap file, and hence is faster than a call of GETX$.

5.1.2 To Release Exclusive Control

To release exclusive control and allow other users access to the system, simply code:

```
CALL RELX$
```

This has no effect if your program did not previously possess exclusive control, or if it is operating in a single user environment. The swap file is reopened if necessary.

5.1.3 Programming Notes

Once a program establishes exclusive control of its processor using GETX$ or GETXN$ no swapping takes place and all other jobs running on the same processor are indefinitely suspended until the exclusive user issues a RELX$ call. However, under networked or separated systems any jobs executing on other processors or systems continue unaffected. Thus GETX$ or GETXN$ cannot in general be used to ensure the exclusive updating of files. The LOCK statement, described in the Global Development File Management Manual, should be used to synchronise file updates since it is equally applicable in both multi-user and networking environments.

GETXN$ and RELX$ might typically be employed by a utility program to be run at infrequent intervals during an online session to transmit data collection files to a remote centre. Normally the communication logic would be provided by a special-purpose assembler subroutine incorporated in the utility by the mechanism described in the Global Development Assembler Interface Manual.

When the communication job is running it gets exclusive control to ensure that it will be able to respond immediately to any messages coming in to avoid the other end of the communications link "timing out". Therefore, use of the communication routine must be bracketed by an initial call on GETXN$ to acquire exclusive control to prevent swapping, and a final call on RELX$ to restore normal multi-user working.

GETX$ and RELX$ might be used on a system where the swap file is on a sub-unit of a exchangeable domain when you need to replace the domain volume temporarily with another volume. You should call GETX$ before de-mounting the volume so that the swap file is closed first. If you do not, errors will occur if the system attempts to access the swap file, and the fact that it is open may cause spurious "file in use" errors.

Note that GETX$ and RELX$ are identical in effect to the GET and REL functions of $STATUS.

5.2 The Start Foreground/Background Routine, START$

The START$ routine is used to transfer control from the current partition to another specified partition on the same screen.

5.2.1 Invocation

To start the foreground or reactivate the background job, you invoke the START$ routine with the call:

CALL START$ USING *partition*

The single parameter *partition* is a PIC 9(4) COMP field that allows you to define the partition to which control is to be transferred.

If used on a pre-V6.0 system, START$ will exit with an exception. On a V6.0 or later system it will swap to the specified partition (providing the partition exists, of course).

The special value of 0 returns control to the last partition to call START$ for this purpose (or has no effect if none has done so).

5.2.2 Programming Notes

This routine can be used to run a program across several partitions.

If START$ is called with no parameters, then this is a pre-V6.0 call to activate foreground.

On a pre-V6.0 system, foreground will be entered, just as if this had been requested by the operator keying <CTRL X>. On a V6.0 or later system this process will be mimicked by switching to partition 2.

If the program calling START$ without parameters is running in foreground (or a partition other than partition 1 on a V6.0 or later system) then the call will initially cause any outstanding suspend in partition 1 to be cancelled, reactivating the task running there if it has been waiting for a time interval to expire.

The routine returns exception condition 1 if it is called from an environment where foreground/background working is not supported.

5.3 Establishing an End of Job Routine, EOJ$

Global Cobol allows you to establish an end of job routine which will gain control once the current job is terminated, just before the ready prompt appears. Typically such a routine is used to ensure that files are left in a consistent state if the job is unexpectedly terminated by a program or I/O error, or the operator's keying of <ESCAPE> or <CTRL W>. (Note, however, that if your sole problem is to protect against the operator hitting the ESCAPE key by mistake, this is more simply handled by using the escape key suppress flag, $$ESC, described the Global Development Screen Presentation Manual.)

Global Cobol uses a PERFORM statement to enter your end of job routine, which is therefore most conveniently coded as an independent SECTION of your program.

Because a program or I/O error might occur at any time during execution of your job, you must ensure that whenever an end of job routine address has been established the routine itself remains permanently resident. Furthermore, it should not occupy or access data from the first 1280 bytes of the user area, since this part of memory will be overwritten by the diagnostic or debug routines if a program check occurs.

When it has finished processing, your routine should return to the monitor by issuing an EXIT statement from its highest level of control.

5.3.1 Establishing the End of Job Routine Address

You use the EOJ$ system routine to inform Global System Manager that an end of job routine is present and establish its address. The routine is invoked with a CALL statement of the form:

```
CALL EOJ$ USING name
```

where *name* is the section name of your end of job routine. If you use a second or subsequent call on EOJ$, the end of job routine address information it specifies overrides any information previously supplied. If you wish to suppress the currently established end of job routine, so that your program no longer regains control on job termination, simply call EOJ$ passing no parameter. That is, code:

```
CALL EOJ$
```

5.4 Abnormal Exit Handling, EXIT$

By calling the EXIT$ system routine you can simulate the execution of a GO TO statement to transfer control to a paragraph, section or entry point which then executes at the highest level of control, even though you invoked it from a deeply nested subroutine. By the highest level of control we mean the level at which the program runs when entered from the ready prompt or following a CHAIN statement, before CALL, PERFORM or EXEC statements have established linkage information on the stack. When EXIT$ is used to transfer control, the old stack contents are discarded, so all information concerning the previous linkage is lost.

In a sense a call on EXIT$ is equivalent to a STOP RUN statement. STOP RUN causes control to return immediately to a high level routine within Global Cobol, irrespective of the level of

control at which the statement was executed. Similarly, by using EXIT$ you can transfer control to a specified high level routine within your own program. Typically routines entered in this way are responsible for handling error conditions.

The alternative, if EXIT$ is not used, is for a low level routine detecting an error condition to set a flag and exit to its caller. The caller tests the flag and, seeing it is on, exits to its caller which, in turn, tests the flag and exits. In this way control is laboriously passed back to the highest level which finally invokes the actual routine required to service the error. It is obviously simpler to set the flag where the condition is first detected and then transfer directly to the required routine.

5.4.1 To Code an Abnormal Exit

To code an abnormal exit you must place the address of the routine you wish to enter in $$EPT and then invoke the EXIT$ system routine. For example:

```
POINT $$EPT AT ERRTN
CALL EXIT$
```

causes control to pass to ERRTN which is entered just as though a

```
GO TO ERRTN
```

had been executed at the highest level of control. The address placed in $$EPT should normally correspond to the name of a paragraph, section, or entry statement with no USING clause, appearing in the mainline of your program. Note, however, that if the error routine is not part of the same compilation, you must code:

```
GLOBAL ERRTN
```

both in the program responsible for detecting the error and in the program containing the routine.

5.5 System Request Routine, CMND$

The CMND$ system request routine enables you to force the invocation of a system request command, such as the Calculator or Help, exactly as if the operator had requested it.

5.5.1 Invocation

CMND$ is invoked by a call of the form:

```
CALL CMND$ USING system-request
```

where *system-request* identifies a PIC X field containing the system request identifier. The permitted values are:

A	Invoke the ASCII/Hex conversion table
B	Invoke keyboard translation system request
C	Invoke the calculator
D	Invoke the calendar facility
E	Invoke the user-defined system request
H	Invoke the help system

I Invoke display/change assignment table system request
J Invoke jot telephone messages
K Invoke Global System Manager function key system request
O Invoke the screen picture utility
P Invoke the screen print utility
Q Invoke record/playback system request
S Invoke reset screen sequence system request
T Invoke talk system request
X Invoke the transfer facility
Y Invoke operator-id name list.

These system requests, with the exception of <SYSREQ> E (the user-defined System Request), are explained fully in chapter 5 of the Global Operating Manual. Section 7.7 of this manual explains how to write and implement your own system requests.

5.5.2 Exception
Exception 1 is returned if it is not possible to honour the request, either because the program is running on a pre-V6.0 system or because customisation has removed the ability to use system request commands.

5.5.3 Programming Notes
Not all of these system requests are available on all versions of Global System Manager and the use of <SYSREQ T> has been modified from V8.0 to V8.1. It is advisable to check the version of Global System Manager that the program is running on before attempting to invoke one of the Global System Manager system requests.

5.6 Inactive/Active Program Routines, LOGOF$ & NLOGF$
Under Global System Manager V5.2, and later, you are not allowed to sign off if there are programs active in your other concurrent partitions; this is to prevent data files being left in an inconsistent state. A partition is considered to be active unless it is at the READY prompt, or at a menu selection prompt, or if the program being run has called LOGOF$ to indicate it can be signed off. The NLOGF$ routine cancels the effect of LOGOF$, and marks the program as active again.

In general, you should call LOGOF$ whenever you are at a main menu with no files open, or if a program is in an "enquiry only" mode.

5.6.1 Invocation
LOGOF$ is invoked by a parameter-less call of the form:

```
CALL LOGOF$
```

to indicate that the user may sign off.

5.6.2 NLOGF$
Whenever you want to cancel the effect of a previous LOGOF$ call, to indicate that the program may not be signed off, do this by a call of the form:

```
CALL NLOGF$
```

No exception condition can be returned. Note that neither routine will have any effect under 5.1 or earlier systems.

5.7 Partition Residency Routines, RESID$ & URESI$

You call the RESID$ system routine to force the current partition to remain in memory, and not be swapped to disk when it is inactive. Typically this is necessary when you use an interrupt-driven assembler program loaded in the user area.

5.7.1 Force Residence, RESID$

RESID$ is invoked by a parameter-less call of the form:

```
CALL RESID$
```

Exception condition 1 will be returned if your user area or partition cannot be made resident for one of the following reasons:

- Your program is running under Global System Manager V5.1 (or earlier), and there are more partitions configured than memory banks available;

- Your program is running in the last memory bank on your processor which has not yet been made resident (one spare memory bank is always required to allow other users to execute).

5.7.2 Disable Residence, URESI$

When your partition no longer requires to be protected against being swapped out you should invoke URESI$ by a call of the form:

```
CALL URESI$
```

No exception conditions are returned. If URESI$ is not invoked, the effects of a previous call of RESID$ are not cancelled until you sign off.

Note that if you leave a partition resident unnecessarily this may degrade performance for other users.

5.8 Write to Log-file, LOG$

The Write to Log-file routine, LOG$, is used for logging significant events in the life of the software (such as end of period, backups, etc.). The log-file is a central file written to in common by all processes making use of the event logging system. The log-file may be examined using the $LOG utility.

5.8.1 Invocation

You invoke this routine by means of a CALL of the following form:

```
CALL LOG$ USING message-area
```

where *message-area* is 60 characters in length, the first 10 of which are used to provide information about the source of the event. The message area is laid out as shown below:

```
01    MA
  03  MAAPP     PIC X(2)     * Application id
  03  MADUAD    PIC X(3)     * Data unit
  03  MATYPE    PIC X        * Event type
  03  MAFUNC    PIC X(2)     * Function code
  03  FILLER          PIC X(2)     * Reserved, set to spaces
  03  MAMESS    PIC X(50)    * Event message
```

MAAPP is the application identifier. Values starting with a $ or upper case letter are reserved for use by Global System Manager, Global Cobol and Global software. MADUAD is the unit address of the principle data unit in use by the application (spaces if there is none). MAFUNC is an application specific function code, to which you will assign your own meanings. The codes used by Global System Manager are documented in the Global Utilities Manual. MATYPE is an event type code with the following defined meanings:

B	backup and restore functions
E	entry/exit, used for starting or ending an application
J	journal, identifying an element in an audit trail for a transaction process
M	maintenance, from application parameterisation
R	error
Z	zap, mostly from Global System Manager system zaps

Other alphabetic codes should **not** be used, as they are reserved for future use by Global System Manager.

The message is prefaced by the operator-id, current date, current time, and computer-id so that there is no need to supply the system with this information.

5.8.2 Exceptions

Exception 1 is returned if there is an irrecoverable I/O error on the log-file.

Exception 2 is returned if there are less than 50 records left on the log-file, although the record will still be written. It is a sensible idea to display a message when this occurs so that the user can purge the file.

Exception 3 is returned if the file is full.

5.9 Authorization Routine, AUTH$

The authorization routine must be invoked once, and once only, for each operator processed by your vetting program. You code a CALL of the form:

```
CALL AUTH$ USING code
```

The *code* is the name of a PIC X field, or a character literal. If you have decided to grant access you must supply an alphabetic character, between A and Z inclusive, which will be established in the system variable $$AUTH as a result of the call. If you wish to refuse access, simply provide a code which is less than ASCII A in collating sequence. For example:

```
CALL AUTH$ USING "?"          * REFUSE ACCESS
```

The AUTH$ routine is explained in more detail in section 7.5.

5.10 The Free Space Management Routine, FREE$

The FREE$ system routine provides a variety of storage management functions which enable you to free the occupied user area; establish the size of the program area; obtain work space from the free area; or return previously acquired work space to the area.

5.10.1 To Free the Occupied User Area

To free the occupied user area, setting the program area size to zero and the work space empty, simply code:

```
CALL FREE$
```

This call should normally be followed by a CHAIN, EXEC or LOAD statement to invoke the loader which will re-establish the program area size from the module loaded. Note that if the work space is empty you can achieve the same effect more simply by just setting system variable $$REL to 1 before issuing the CHAIN, EXEC or LOAD statement. This is explained in 8.1.2.

5.10.2 The Free Space Management Request Area

The other free space management functions require that FREE$ is passed a free space management request area of the following format:

```
01    FM
  03  FMFUN      PIC 9 COMP       * FUNCTION REQUIRED
                                  * 0 - FIND PGM AREA SIZE
                                  * 1 - SET PGM AREA SIZE
                                  * 2 - GET WORK SPACE
                                  * 3 - FREE WORK SPACE
  03  FMSIZE     PIC 9(6) COMP    * SIZE IN BYTES
  03  FMALL      PIC 9(6) COMP    * SPACE ALLOCATED
  03  FMPTR      PIC PTR          * POINTER TO SPACE
```

5.10.3 To Find the Size of the Program Area

To find the size of the program area you must set FMFUN to 0, then invoke the routine with the statement:

```
CALL FREE$ USING FM
```

FREE$ will return the size of the program area, in bytes, in FMSIZE.

5.10.4 To Set the Size of the Program Area

To set the size of the program area, providing the work space is **not** in use, you must set FMFUN to 1 and place the size, in bytes, of the area you require in FMSIZE. Then invoke the routine with the statement:

```
CALL FREE$ USING FM
```

FREE$ will return exception condition 1 if it is unable to satisfy your entire request, but in this case it will allocate whatever space is actually available. If your request can be honoured in full the routine returns normal completion.

Whether or not an exception is generated, FMALL will be set to contain the number of bytes **actually** allocated to the program area. Providing no exception is returned the value returned in FMALL will, of course, be the same as that in FMSIZE.

If you attempt to set the size of the program area when the work space is in use, the program calling FREE$ will be terminated in error.

5.10.5 To Obtain Work Space from the Free Area

To obtain work space from the free area, you must set FMFUN to 2 and place the number of bytes you require in FMSIZE. The total size of the work space acquired, including any space acquired by previous FREE$ calls, is limited to 32766 bytes. You invoke the routine by coding:

```
CALL FREE$ USING FM
```

FREE$ will generate exception condition 1 if it is unable to satisfy your entire request, but in this case it will allocate whatever space is actually available up to a maximum of 32766 bytes. If your request can be honoured in full the routine returns normal completion.

Whether or not an exception is generated, FMALL will be set to contain the number of bytes actually allocated and FMPTR will be set to address the new work space. (If **no** bytes were available the value of FMPTR is undefined.) Providing no exception is returned the value returned in FMALL will, of course, be the same as that in FMSIZE.

5.10.6 To Return Work Space to the Free Area

To return work space to the free area, you must set FMFUN to 3 and place the number of bytes you wish to release in FMSIZE. You invoke the routine with the statement:

```
CALL FREE$ USING FM
```

The routine will decrement the free space starting location by the number of bytes specified in FMSIZE. If this results in the work space becoming empty then it is considered to be **not in use**.

If you attempt to return more bytes than your work space actually contains your program will be terminated in error.

5.11 The User Number & Operator-id Checking Routines, USER$ & OPID$

These routines are provided for checking operator-ids and user numbers.

5.11.1 Invocation

The USER$ routine is used to find whether a specified operator is signed on to the system and, if so, return the appropriate user information. It is invoked by a call of the form:

CALL USER$ USING *us area*

where *area* is a PIC X(2000) work area for use by USER$ and *us* is a control block of the following format:

```
01    US
  03  USOPID     PIC X(4)          * operator-id
  03  USUNO      PIC 9(2) COMP     * user number
  03  USSCNN     PIC 9(2) COMP     * screen number
  03  USCID      PIC X             * computer-id
  03  FILLER           PIC X(2)          * reserved
  03  USPART     PIC 9(2) COMP     * partition number
```

The operator-id of the required operator must be established in USOPID before USER$ is called. On successful completion USUNO, USSCNN and USCID will be set to the values and USPART will be set to 1.

The OPID$ routine is used to return operator-id information for a particular user on a particular computer. It is invoked by a call of the form:

CALL OPID$ USING *us area*

where, again, *area* is a PIC X(2000) work area for use by OPID$ and *us* is a control block in the format described above.

The user number must be set up in USUNO before OPID$ is called. If you are running on an XLAN system, the computer-id must also be established in USCID.

On successful completion, OPID$ returns values of USOPID, USSCNN and USPART (plus USCID on non-XLAN systems) for the specified user number.

5.11.2 Exception conditions

Exception condition 1 will be returned if any I/O error occurs on the user file during the call to OPID$ or USER$.

Exception condition 2 is returned by USER$ if the operator is not signed on to the system.

Exception condition 2 is returned by OPID$ if the specified user number is out of range for the system (or computer on an XLAN system).

Exception condition 3 is returned by both routines if used on a pre-V5.1 system.

Exception condition 3 is also returned by OPID$ if it is called with a computer-id of low-values (#00) on an XLAN system.

5.11.3 Programming notes

USER$ can be used to determine screen and computer information for a particular operator, so that MSG$ can be used to send the operator a message.

Repeated calls of OPID$ can be used to find out which operators are signed on to the system:

Firstly, call OPID$ with USUNO set to 1 and USCID set to #00. If exception condition 3 is returned then this is an XLAN system and needs to be treated specially.

For a non-XLAN system, repeated calls with USUNO incremented by 1 each time will return successive operator-ids, but where an operator has multiple partitions you should ignore those in blocks returned with USPART not equal to 1. Exception condition 2 indicates the end of the table.

For an XLAN system, you must perform the above operation for **each computer**. It is conventional to start with USCID set to #41 and to increment it to #FF, then go from #01 to #40. Exception condition 2 indicates the end of users for the specified computer-id.

In both cases, an operator-id of spaces indicates that the operator at that position in the table is not currently signed on.

There is a subtle difference between USUNO and USSCNN. USUNO contains the User Number, a value between 1 and 99. The user number is an "external format" field. For example, the user number appears in the $STATUS report. USSCNN contains the Screen Number, a value between 1 and 255. The screen number is an "internal format" field. For example, the screen number is supplied to the MSG$ subroutine (see section 7.18 of the Global Development Screen Presentation Manual).

5.12 Convert Computer-id for Display Routine, CID-D$ and D-CID$

5.12.1 Converting Computer-id to Display Format

You use the CID-D$ routine to convert a computer-id (e.g. $$LNID) into a form that can be displayed on the screen. This routine is the same as that used by Global System Manager programs.

You invoke this routine with a call of the following form:

CALL CID-D$ USING *comp-id disp-id*

where *comp-id* is a PIC X field containing the hexadecimal computer-id and *disp-id* is the field you wish to contain the converted display-id. The following shows the way the conversion is carried out:

Hardware address (Arcnet)	*$$LNID*	*Display format*
#01 - #1A	"A" - "Z"	A - Z
#1B - #BF	#5B - #FF	1B - BF
#C0	invalid	invalid
#C1 - #FF	#01 - #3F	C1 - FF
#00	invalid	invalid

Note that for other networks than Arcnet different correspondences may exist between hardware address and Global Cobol addresses.

5.12.2 Converting display-id to internal format

To convert from display-id to internal format, the following call is used:

CALL D-CID$ USING *disp-id comp-id*

Exception 1 is returned in the event of an invalid display format id.

5.13 Return Operator Grouping Information, GROUP$

This routine returns the information about a particular operator group, on V8.1 or later Global System Manager, provided that group handling is in use. It can also be used to return the next or previous group in the group list (see the description of $GROUP in the Global Utilities Manual.)

5.13.1 Invocation

The GROUP$ routine is invoked by a call of the form:

CALL GROUP$ USING *gp gpfunc*

where *gp* is a control block of the following format:

```
01   GP
  03 GPNAME                PIC X(10)        * Group name
  03 GPNMEM                PIC 9(2) COMP    * Number of members
  03 GPDESC                PIC X(50)        * Group description
  03 GPOPID OCCURS 99      PIC 9(4) COMP    * Operator-id of members
```

and *gpfunc* is a the name of a PIC 9(4) COMP field or a literal containing the GROUP$ function number.

If the GROUP$ function number is set to 0 then GROUP$ will return the details of the group whose group name is supplied in GPNAME.

If the function number is set to 1 then GROUP$ will return the details of the group whose name is immediately prior to the group whose name is supplied in GPNAME. Therefore if GPNAME is set to HIGH-VALUES the last group will be returned.

If the function number is set to 2 then GROUP$ will return the details of the group whose name is next in collating sequence to the group whose name is supplied in GPNAME. Therefore if GPNAME is set to LOW-VALUES the first group will be returned.

5.13.2 Exception Conditions

Exception condition 1 will be returned if a group file of the correct type is not found or is in use or if an I/O error occurs on the group file.

Exception condition 2 will be returned if there is no current, next or last group to be found depending on the GROUP$ function number.

Exception condition 4 is returned if GROUP$ is run on a pre-V8.1 system.

5.14 Return Operator's Full Name, OPNM$

This routine returns the full name for a given operator on V8.1, or later, Global System Manager provided the operator-id table is being used (see the description of $OPID in the Global Utilities Manual).

5.14.1 Invocation

The OPNM$ routine is invoked by a call of the form:

```
CALL OPNM$ USING operator name area
```

where *operator* is the name of a PIC X(4) field or character literal which contains the operator-id of the operator for whom the full name should be returned, *name* is the name of a PIC X(35) field into which the full name of the operator will be returned, and *area* is a PIC X(1624) work area for use by OPNM$.

5.14.2 Exception Conditions

Exception condition 1 will be returned if the operator list file is not found or is in use or if an I/O occurs on the operator file.

Exception condition 2 will be returned if the operator-id supplied is not found in the operator list file.

Exception condition 3 will be returned if OPNM$ is called on a pre-V8.1 system.

5.15 Extended Overlay Management (SBOVL$ & OVLAY$)

A special overlay loader is available which overlays an existing Speedbase frame with an "overlay frame". On exit from the "overlay frame" the current status is restored. Another, related overlay routine is available to perform a similar function for Global Cobol programs. These new overlay loaders are similar in concept to the system request loader, CMND$ (see section 5.5), which is currently available for use by Global Cobol programs.

Important note: During development, the new Speedbase overlay subroutine was known by its working title of FRAME$. This name was changed to SBOVL$ during the final repackaging.

5.15.1 SBOVL$ Invocation

The SBOVL$ routine is called by the **underlying frame** as follows:

```
CALL SBOVL$ USING frame-initiator table
```

where *frame-initiator* is the name of the **overlay frame initiator**, written in Global Cobol, which overlays the current code in memory. This Overlay Frame Initiator must reside on $P as a stand alone program or within the currently attached library. The *table* consists of both an export and import table, each consisting of a set of 16 pairs of pointers indicating the start and end of data items:

```
01    TABLE
  03  FILLER          PIC X          * RESERVED
  03  EXPTR OCCURS 16                * EXPORT POINTERS
    05 EXPTRS    PIC PTR          * START POINTER
```

```
   05  EXPTRE      PIC PTR          * END POINTER
03     IMPTR OCCURS 16                    * IMPORT POINTERS
   05  IMPTRS      PIC PTR          * START POINTER
   05  IMPTRE      PIC PTR          * END POINTER
```

The export set of data pointers indicate the data that can be passed from the underlying frame to the overlay frame; and the import set of pointers indicate the data returned from the overlay back to the underlying frame. **Important note**: Any unused pointers must be set to #FFFF.

The SBOVL$ routine invokes $MONITOR in a similar way to the current CMND$ system request loader. The monitor opens a work file storing the table of data pointers, the diagnostic log out area, the screen parameters, the current screen, and all the code in the current user area.

Important note: The frame loaded by the frame initiator passed to SBOVL$ must be a root frame and not a dependent frame. Note also that $$AREA is **NOT** saved by SBOVL$. If an application needs to save $$AREA, it must do so itself in an area inside the program. System area fields **cannot** be passed as parameters via the SBOVL$ table.

Note that SBOVL$ calls may be nested to any level (the nesting level is limited to the number of work files that can be opened on the work unit - see below). Note also that a new work-file is opened for each new overlay level.

5.15.2 FPOP$ Invocation

Within the Overlay Frame the data passed via the SBOVL$ table can be retrieved via a call of FPOP$ as follows:

```
CALL FPOP$ USING table
```

where *table* is a table of 16 pointers indicating the start of the areas into which the data passed should be restored:

```
77     TABLE OCCURS 16 PIC PTR          * START POINTER
```

The FPOP$ memory page routine reads the save file according to the SBOVL$ export table and restores the data as required by the FPOP$ table. **Important note**: Any unused pointers must be set to #FFFF.

5.15.3 FPUSH$ Invocation

Before exiting, the Overlay Frame may return date by calling the following routine:

```
CALL FPUSH$ USING table
```

where *table* is an array of 16 pointers indicating the position inthe Overlay Frame of the data which is to be returned to the underlying program as indicated by the SBOVL$ import table:

```
77     TABLE OCCURS 16 PIC PTR          * START POINTER
```

This memory page subroutine writes the data to the data areas in the saved file.

On exit from the frame overlay, the screen, link stack and underlying code (containing any modified data) are restored. **Important note**: Any unused pointers must be set to #FFFF.

5.15.4 SBOVL$ Functionality

The SBOVL$ routine checks the version of Global System Manager and only operates with GSM V8.1, or later. It also opens a work file of the correct size which is the sum of the following:

- Size of the screen image (calculation for the size of the screen image is as for system requests);
- Size of area for saved system area variables;
- Size of the user area;
- Size of the table passed to SBOVL$.

The save file is called $$FOVW*nn* and is opened on unit $OF, if assigned, or $DP otherwise.

Important note: The $OF unit **MUST** be on the same computer node-id or SYSTEM letter as the $DP unit. The channel number of this file is passed to the monitor via the first byte in the SBOVL$ table.

Not that a unique name is produced for each Speedbase work file regardless of the SBOVL$ nesting level.

5.15.5 OVLAY$ Invocation

The SBOVL$ sub-routine is for use by Speedbase applications. The functionally equivalent OVLAY$ sub-routine is for use by Global Cobol applications.

The OVLAY$ routine is called by the **underlying program** as follows:

CALL OVLAY$ USING *program table*

where *program* is the name of the **overlay program** which overlays the current code in memory. This Overlay Program must reside on $P as a stand alone program or within the currently attached library. The *table* consists of both an export and import table, each consisting of a set of 16 pairs of pointers indicating the start and end of data items as described in section 5.15.1.

6. System Variables

System variables are elementary data items which are conceptually declared automatically in the data division of every compilation. The variables do not actually appear as data definitions; neither do they occupy working storage. They are located within a permanently available region known as the **System Area**, which is used to communicate parameter information between Global System Manager and an application program. They can be referenced from procedure division statements whenever a level 77 item with the same picture clause would be valid.

This chapter covers the most commonly used system variables. Additional system variables are defined in the following chapter and in the Global Development Job Management Manual, the Global Development Screen Presentation Manual, the Global Development File Management Manual and the Global Development Assembler Interface Manual.

The following two points should be considered when using system variables:

- Unless it is explicitly stated to the contrary, a system variable should be read-only as far as the application program is concerned;

- System variables may be redefined, but the redefinition must appear in the LINKAGE SECTION.

6.1 The Application Work Area, $$AREA

The application work area consists of 16 bytes within the system area which are available for application use. The bytes are set to binary zero at the start of a session. However, once this area has been erased in this way Global System Manager never refers to it again. Thereafter the 16 bytes can be used for any application purpose whatsoever. The work area allows you to pass a small amount of information between programs very conveniently.

To refer to the application work area you simply redefine $$AREA in the linkage section. For example:

```
LINKAGE SECTION
01    AP REDEFINES $$AREA
  03  APINIT     PIC 9 COMP       * INITIATION FLAG
  03  APPRIV     PIC 9(2) COMP    * OPERATOR PRIVILEGE LEVEL
  03  APNUMB     PIC 9(4) COMP    * OPERATOR NUMBER
```

In practice, of course, you would probably place the entire definition of the application work area in a copy book, so that each programmer need only code, for example:

```
LINKAGE SECTION
COPY AP
```

6.2 The Exception Condition and Result Code, $$COND and $$RES

System variable $$COND is set by any Global Cobol statement which can return an exception. When non-zero it is a positive integer used to identify the reason for an exception when the condition arises due to a number of different circumstances. For certain exceptions system

variable $$RES will also be established to provide further information about the reason for the exception.

$$COND and $$RES are always used in conjunction with an ON EXCEPTION statement, and are explained in more detail as part of the description of that statement in Chapter 4 of the Global Development Cobol Language Manual.

$$RES is also used for special I/O error handling, which is described in the Global Development File Management Manual.

6.3 Today's Date, $$DATE

Today's date is established in $$DATE from information supplied or confirmed by the operator at the start of a session. The date is stored as a computational number equal to:

10000*(year - 1900) + 100*(month of year) + day number

in order that dates can be compared using the standard arithmetic conditional statements. The date conversion routines allow you to convert a date in this PIC 9(6) COMP internal format to an 8 character external form, suitable for printing or displaying, and the DT-DY$ routine enables you to calculate the day number since 1900 from an internal format date field. These routines are described in detail in Chapter 2.

6.4 The day of the week, $$DOWK

This is a PIC 9(2) COMP field containing the day number of the current system date (with 1 representing Sunday, 2 representing Monday etc.). This flag is set to zero on pre-V6.0 Global System Manager.

6.5 The Century Start Year, $$NCYR

This is a PIC 9(2) COMP field containing the year from which the century is said to start for short date purposes on the system. The century start year can be customised using $CUS (see the Global Utilities Manual).For example, if the century start year is set to 50 then the short date, 01/01/11 will be set to refer to 01/01/2011 and 01/01/60 to 101/01/1960 by the date conversion routines.

6.6 The Disable System Request Flag, $$INT

$$INT is a PIC 9 COMP field that can be used to disable system requests during crucial code paths within a program. A value of 1 disables system requests and a value of 0 re-enables them. This flag is used by $CUS to disable system requests, so its value must be preserved by your program. This flag is not available on pre-V6.2 Global System Manager.

6.7 Current Program Information, $$EPT, $$PGM and $$RUN

$$EPT addresses the entry point of the program last loaded, $$PGM contains its program-id, and $$RUN indicates whether the program has been loaded as a result of an operator request ($$RUN = 1) or because of a LOAD, CHAIN or EXEC statement executed by another program ($$RUN = 0). Use of these system variables is explained in more detail in chapter 4.

6.8 The Console Dimensions, $$LINE, $$WIDE and $$RWID

The system variables $$LINE and $$WIDE are two PIC 9(4) COMP fields which indicate, respectively, the depth of the console in lines and its width in characters. Typical values with current technology are $$LINE = 24 and $$WIDE = 80. However, Global System Manager supports some consoles with much larger dimensions, for example $$LINE = 55 and $$WIDE = 132.

Global System Manager will operate successfully on consoles with $$WIDE = 40 or greater, and $$LINE = 20 or greater. However most Global Software requires a console at least 79 characters wide and 24 lines deep. (Note that a number of consoles which are nominally 80 characters wide handle the 80th character position specially, and hence must be treated as 79 characters wide by Global System Manager.)

Portable programs should always check the dimensions of the console to ensure that it is sufficiently large, particularly if they use formatted screens. A program may be able to adjust the depth or width of its displays to suit the console. For example, programs writing long reports to the console in teletype mode should use $$LINE to determine when to output a paging prompt such as:

    Key P to Page, A to Accept and continue:

Normally ($$LINE)-1 lines of report will be output, followed by the prompt, which will then appear on the base line to give the operator the chance to read the information that has just been displayed. Usually, as in the example, you will allow the operator the option of either asking for the next "page" of output (with automatic "wrap around" to the first page following the last), or requesting that the program continue once enough of the report has been read.

Note that even if $$WIDE is greater than 80 the maximum amount of information you can input or output using a single ACCEPT or DISPLAY statement is still restricted to 80 bytes.

$$RWID, the physical terminal's maximum width (i.e. the widest available screen display), is a PIC 9(4) COMP field which indicates the **physical** terminal's maximum width (as opposed to $$WIDE, which indicates the current **logical** width).

6.9 The Standard Printer Page Size, $$PAGE

$$PAGE contains the number of lines per page for the **standard** printers used by a configuration. If printers with two or more different page sizes are supported a decision must be made at installation time as to which size is to be the standard. Normally 66 lines per page is adopted. If any other value is to be established it must be set up by running $CUS, selecting Configuration customisation, and altering Standard Page Size.

Portable application programs writing reports to standard stationery should refer to $$PAGE to determine when it is necessary to advance the stationery to a new page. The page size of special stationery is under program control, as explained in the section on print files in section 1.5 of the Global Development File Management Manual.

6.10 The Standard Printer Line Width, $$PRIN

$$PRIN contains the line width in characters of the **standard** printers used by a configuration. If two or more different line widths are in use a decision must be made at installation time as to which is to be the standard. Normally 132 characters per line is adopted. If any other value is to

be established it must be set up by running $CUS, selecting Printer customisation, and altering Printer Page Size.

By using $$PRIN together with $$PAGE it is possible to develop applications which are able to adapt themselves to a variety of different paper sizes.

6.11 The Seed for the Random Number Generator, $$SEED

System variable $$SEED is a PIC 9(9) COMP item that you may set to cause the RAND$ system routine to generate a repeatable sequence of pseudo-random numbers, as explained in section 3.1.

6.12 The System Name, $$SNAM

This is a PIC X(30) field containing the system name set up using $CUS and is displayed on subsequent Global System Manager menus. This variable is not available on pre-V6.0 Global System Manager

6.13 The Global System Manager operating System Flag, $$SYSM

This PIC 9 COMP flag identifies the host operating system on which Global System Manager is running (0 for Global System Manager (BOS); 1 for Global System Manager (Unix); 2 for Global System Manager (MS-DOS and Windows); 3 for Global System Manager (Novell NetWare); 4 for Global System Manager (Windows) other values are reserved for future use). This flag is not available on pre-V8.0 Global System Manager.

6.14 The Presentation Manager Flag, $$PM

This is a PIC 9(2) COMP field which is set to 1 if the system the program is running on is a Global System Manager - Presentation Manager system and set to 0 if the system is just a Global System Manager system. This flag is not available on pre-V8.1 systems.

6.15 The Maximum Number of Users, $$ULEV

The is a PIC 9(4) COMP field which contains the maximum number of users for which this system can be configured. The maximum number of users depends on the level number of the system ordered. This flag is not available on pre-V8.1 systems

6.16 The Global System Manager Version Number Indicator, $$VERS

System variable $$VERS is a PIC 9 COMP field which indicates the version of Global System Manager being used. It is set to 0 for V5.0; 1 for V5.1; 2 for V5.2; 3 for V6.0; 4 for V6.1; 5 for V6.2; 6 for V7.0; 7 for V8.0; 8 for V8.1 and, potentially, 9 onwards for future releases. It should be used first to determine what further checks on the task environment need to be made by your program.

Note that some of the following variables, together with those that follow are mainly of use under multi-user systems. However, they are allocated default values under single-user systems so that jobs which use them can run with any type of Global System Manager.

6.17 The Global System Manager Level Number Indicator, $$LEVN

$$LEVN is a PIC X field which holds the level number of Global System Manager being used. The field is only available in the V5.2 compiler, and only exists in V5.1, or later, Global System Manager. The value held in the field will be one of the following:

1	Not used at present
2	SBOS
3	CBOS
4	MBOS/PC
5	MBOS
6	BOS/NET (obsolete)
7	BOS/LAN
8	BOS/LAN+
9	BOS/XLAN

6.18 The American Processing Flag, $$USA

$$USA determines the way in which the Global System Manager command programs, together with the DATE$ system routine, construct the external form of the date to be used on listings and displays. When Global System Manager is distributed $$USA is set to 0, indicating that European processing is in force, so the external date appears as the 8 characters:

dd/*mm*/*yy*

where *dd*, *mm* and *yy* are the day number, month number, and year of century, with leading zeros as necessary.

When Global System Manager is installed you can choose either European or American date format, thus setting the value in $$USA to 0 or 1 respectively. If the American format is chosen the external form of the date appears as:

mm/*dd*/*yy*

The format used by an installed system can be altered at any time by running $CUS, selecting Configuration customisation, and altering Date Format.

$$USA can also be used to select other application-dependent American or European options at run-time, thus avoiding the need for two versions of a program.

6.19 The Program Volume-id, $$PVOL

When programs are loaded from exchangeable disks or diskettes you can request Global System Manager to check that the correct volume is online, and to prompt the operator to mount it if it is not. To do this you set $$PVOL to the six character volume-id used to uniquely identity the program volume just before executing the LOAD, EXEC, CHAIN or RUN statement responsible for bringing the program into memory. Once the statement returns control $$PVOL will be reset automatically to its initial state, low-values, to prevent Global System Manager making further checks until you set up another volume-id. Note that $$PVOL has no effect if used on V5.0 Global System Manager.

6.20 The Task Environment Indicator, $$TASK

System variable $$TASK is a PIC S9 COMP field which indicates if your program is currently running as a background job, a normal job, or a foreground job on V5.2 or earlier systems:

$$TASK	*Task environment*
-1	background job
0	normal job
+1	foreground job

A program is said to be running in a normal job environment (with $$TASK = 0) when it is the only job working for a particular operator, as is always the case under SBOS, or if it is running in a concurrent partition. (See $$PAR, used to distinguish different partition numbers.) In the multi-user environment under V5.2 or earlier, however, it is possible for an operator to run a background and a foreground job simultaneously and in this case these jobs will have $$TASK values of -1 and +1 respectively.

You may find it useful to test $$TASK and suppress non-vital screen messages when a job runs in the background, in order to prevent it being unnecessarily suspended. For example, the Global Cobol compiler avoids displaying its first pass error messages when it is executing in the background. Note that you can conveniently suppress all dialogue from a program, without modifying it in any way, by running it under job management and using the SUPPRESS option.

Under V6.0 Global System Manager (and later) $$TASK **always** has a value of 0.

6.21 The Partition Number Indicator, $$PAR

$$PAR is a PIC 9 COMP field in which is returned the partition number currently being used. It is set to 1 or 2 for foreground or background, or 1 - 9 (usually 1 - 4) for concurrent partitions. On single-user processors it is set to 1.

Note that this variable is not available under V5.0. The Global System Manager version number (given by $$VERS, see 6.15) should be checked first, and if this indicates that the program is running under V5.0 $$TASK should be used instead.

6.22 The User Number, $$USER

System variable $$USER is a PIC 9(2) COMP field containing the current user number. A unique user number (between 1 and 99) is allocated for each partition when the system is initiated.

Note that under single-user systems $$USER is always 1. Note also that $$USER should not be relied upon to be unique across a network, since the user number refers to users of a single computer.

6.23 The Multi-user Flag, $$MU

System variable $$MU is a PIC 9 COMP field which is set to zero under SBOS, and set non-zero in all other levels of Global System Manager. Applications can test $$MU in order to bypass special shared file handling procedures when operating in a single-user environment. For example, there is no need for a single-user program to LOCK the files it requires to update.

The following table shows how $$MU indicates the exact type of system under which your program is running:

$$MU	*Type of system*
0	Single-user, single processor (SBOS)
1	Multi-user, single processor (CBOS or MBOS)
2	Single-user processor in a networking environment
3	Multi-user processor in a networking environment

6.24 The Master Node Indicator, $$MNID

$$MNID, the master node indicator, is a PIC X field which identifies the master computer (or system) on a networked or separated system by means of its computer or system identification letter, A - Z. It is set to #00 for other (including V5.0) systems, and can thus be used as a test for whether you are on a network system.

6.25 The Local Node Indicator, $$LNID

$$LNID, the local node indicator, is a PIC X field which gives the computer (or system) identification letter on a networked or separated system. If your computer (or system) is a file server it will be in the range A - Z. For non file server computers (or systems) it can take other values (not necessarily printable). Values in the range #40 - #FF correspond to node numbers 0 - 191, and #01 - #3F correspond to 193 - 255. For non-network systems (including V5.0) $$LNID is set to #00.

6.26 The Screen Number, $$SCNN

$$SCNN is a PIC 9(2) COMP field identifying the partition by a unique screen number (between 1 and 255). This variable is used in various System Routines such as MSG$.

6.27 The Attached Library, $$LIB

System variable $$LIB is a PIC X(8) field containing the library-id of the currently attached user library. If there is no library attached it will contain "P.".

This variable allows a subroutine which needs to attach a library to save the name of the caller's library on entry, and re-attach this library on exit. Its use is explained in section 7.1.

6.28 The Operator-id, $$OPID

The system variable $$OPID is a PIC X(4) field containing the operator-id supplied in response to the prompt:

PLEASE KEY YOUR OPERATOR-ID:

output at the start of a session. Global System Manager checks that the operator-id you supply to the prompt is not the same as any other operator-id currently signed on. The same operator-id is associated with all jobs initiated by an operator during a session, and therefore competing foreground, background and concurrent jobs started at the same screens have the same operator-id.

6.29 The Authorization Code, $$AUTH

The system variable $$AUTH is a PIC X field containing an alphabetic value in the range A to Z, inclusive. It is used when creating an optional authorization vetting program to establish the code for each operation allowed to access your system, as described in section 7.5. of this manual.

6.30 The Supervisor Program Name, $$SVSR

The system variable $$SVSR is a PIC X(8) field in which you can establish the name of a supervisor program. Its use is described in section 7.6.

6.31 The Release Program Area Flag, $$REL

The size of the program area is normally only ever increased by the loader. By setting the PIC 9 COMP system variable $$REL to 1 before executing a LOAD, CHAIN or EXEC statement you can cause the loader to set the program area size to the end of the program it is loading, provided the work space is not in use. If the work space is in use setting $$REL has no effect: you must use FREE$ instead.

6.32 Library Index Pointer, $$INDE

The library index pointer, $$INDE, is a PIC PTR system variable, used in storage management (see chapter 8), which addresses the first byte to be occupied by the 1134-byte library index record when an overlay is next loaded from a program library. At the beginning of each job Global System Manager will have set $$INDE to address the top 1134 bytes of the user area. A job which acquires a work space and then continues to require overlays, or which uses an unconventional overlay scheme, should set $$INDE to address a buffer within the transient area to be used for its overlays in order to prevent the index record corrupting any other part of the occupied user area as explained in 8.1.4.

Note that the LOAD$ and UNLO$ system routines modify $$INDE if they are used to introduce or remove a relocatable program.

6.33 Default Menu Entry, $$MEDF

The default menu entry, $$MEDF, is PIC 9 COMP system variable. Its use is described in section 7.5.6.

7. Special Coding Techniques

7.1 Attaching Program Libraries and Examining their Contents

You may use the LOAD statement to attach (and later detach) a program library under program control. Once a library is attached Global System Manager will search it for programs requested by LOAD, EXEC, CHAIN, or RUN statements, or by operator replies to the ready prompt, before checking whether the program residence device contains an individual file named after the program-id. The library remains attached until either another library is attached in its place, or it is explicitly detached. The attach and detach operations may take place under program control, as described here, or as a consequence of operator responses to the ready prompt, as explained in the Global Operating Manual.

If you set system variable $$INDE to address an 1134-byte data area within your program, the attach operation will cause a copy of the library index record to be moved to the area. You can then examine the index to determine the names of the programs the library contains, and whether any particular member is actually present in the file currently online, or is only represented by a stub.

To attach a library under program control you simply supply its library-id (i.e. its file-id), which must begin with the prefix P., as the operand of a LOAD statement. Similarly to detach the library currently attached, you simply request to "load" the special file named "P." For example, when the statement:

```
LOAD "P.SA"                          * ATTACH SALES LEDGER LIBRARY
```

returns normal completion, program library P.SA will have been attached. To subsequently detach this library you code:

```
LOAD "P."                            * DETACH SALES LEDGER LIBRARY
```

7.1.1 Exceptions

Exception condition 1 is returned if an irrecoverable I/O error occurs when you attempt to attach a library, and exception condition 2 is generated if the library file is not present on the program residence device and you reply N to the library required prompt. In either case, the previously attached library, if any, will remain attached. No exception is returned when a library is detached.

7.1.2 The Attached Library, $$LIB

System variable $$LIB is a PIC X(8) field containing the library-id of the currently attached user library. If there is no library attached it will contain "P." (see section 6.24).

This variable allows a subroutine which needs to attach a library to save the name of the caller's library on entry, and re-attach this library on exit.

For example, such a subroutine might contain the following statements:

```
MOVE $$LIB TO SAVE-LIB               * SAVE CALLER'S LIBRARY
LOAD "P.SA"                          * ATTACH OWN LIBRARY
```

```
.........
......... (other processing)
.........
LOAD SAVE-LIB                          * RESTORE CALLER'S LIBRARY
EXIT
```

Note that if there were no library attached on entry, $$LIB would contain "P.", and hence the final LOAD statement would simply detach the subroutine's library, restoring the initial status.

7.1.3 Examining the Library Index

If you wish to determine which programs are actually resident in a library you should set the PIC PTR system variable $$INDE to address a 1134-byte index area within your program, and then attach the library in question. Providing the attach operation completes normally the library index record will be moved to the area you reserved for it, and you will be able to examine it to determine the library contents. The index record is of the following format:

```
01    LX
  03  LXTTL              PIC X(30)              * LIBRARY TITLE
  03  LXDATE      PIC 9(6)                     * DATE LAST MODIFIED
  03  LXFLAG OCCURS 100 PIC 9(6) COMP          * FLAG FOR I'TH MEMBER
  03  LXID OCCURS 100 PIC X(8)                 * PROGRAM-ID FOR I'TH MEMBER
  03  LXTER              PIC X                  * #00 TERMINATOR WHEN
                                                * THERE ARE 100 MEMBERS
```

The LXID table contains the 8-character program-ids of up to 100 members. Providing the library has been either created or modified using $LIB the member-ids will be in ascending ASCII collating sequence. (However, the order of member-ids in libraries created under Global System Manager 4.2 or earlier, and not subsequently modified, is unpredictable.) The last member-id is followed by a binary zero byte which serves as a terminator so that the LXID table can be examined by SEARCH or SCAN statements.

For each member-id there is a corresponding flag in the LXFLAG table. Usually the flag will be positive indicating that the member is actually present in the library file you have just attached. However, if the library you are examining is a dispersed library, occupying a number of identically named files on different exchangeable volumes, then some of the members will only be represented by stubs (as explained in the documentation of $LIB in the Global Development Cobol User Manual). When there is only a stub present for a member its LXFLAG value is negative.

You should normally preserve and restore the initial value of the $$INDE pointer over the attach operation. For example:

```
MOVE $$INDE TO SAVE-PTR          * SAVE INITIAL VALUE
POINT $$INDE AT LX               * ADDRESS OWN 1134-BYTE LX AREA
LOAD "P.SA"                            * ATTACH LIBRARY
MOVE SAVE-PTR TO $$INDE          * RESTORE INITIAL VALUE
                                 * EXAMINE THE LX RECORD HERE
```

7.1.4 Programming Notes

There are additional uses for system variable $$INDE, when sophisticated storage management schemes are required (see chapter 8).

The system variable $$EPT is corrupted by the LOAD statement's attaching or detaching of a library.

If an exception occurs during an attach operation and you fail to trap it in logic introduced by an ON EXCEPTION statement immediately following the responsible LOAD statement, your program will be terminated.

Only one program library can be attached at a time. However, should you need to attach a different library there is no need to detach the current one: the attaching of the new library automatically takes care of this. A detach request issued when there is no library attached has no effect.

7.2 Returning an Exception Condition

You may code the EXIT WITH condition statement to return control from a PERFORM'ed paragraph or section, or a CALL'ed subroutine, returning an exception condition. Your caller must have coded an ON EXCEPTION statement introducing logic to handle the condition **immediately** following the PERFORM or CALL, otherwise the program will be terminated in error.

The statement is written:

```
EXIT WITH condition
```

where *condition* is an integer literal or PIC 9(4) COMP variable whose value is between 1 and 99 inclusive. Global System Manager places this value in the system variable $$COND, where it is available for your caller to examine.

EXIT WITH *condition* may be coded wherever an ordinary EXIT statement would be valid. Thus the statement can be part of a format 2 conditional. For example:

```
IF RESULT NOT POSITIVE EXIT WITH 1
```

The exception mechanism provides a very convenient and efficient way of processing special conditions.

Note that it is important that the condition values that you use are in the range 1 to 99 since other values are reserved for the use of Global System Manager and Global Cobol.

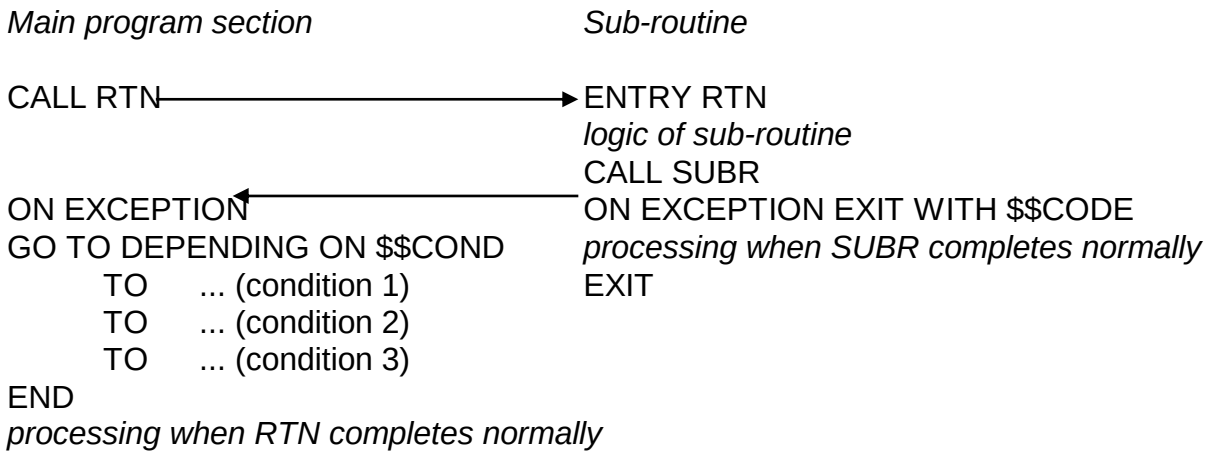
Normally if a routine can detect several different conditions they should be assigned the numbers 1, 2, 3 ... and so on, in order that the calling program can use a:

```
GO TO DEPENDING ON $$COND
```

statement immediately after the ON EXCEPTION statement to route control swiftly to the code responsible for handling a particular condition.

7.2.1 Reflecting an Exception

Sometimes a routine requires to pass an exception condition returned by a lower-level subroutine back to the caller of the original routine for processing. This could be done by moving $$COND to a PIC 9(4) COMP field to be used in a subsequent EXIT WITH condition statement. However, the same effect can be achieved by using the special system variable $$CODE, as shown in the following example:



Program MAIN invokes RTN which in turn calls SUBR, a subroutine capable of generating three possible exception conditions. Within RTN an ON EXCEPTION statement following the call on SUBR detects these exceptions and causes the EXIT WITH $$CODE statement to be executed should one be returned. This statement causes control to be passed back to MAIN and generates a second exception. However, because $$CODE was used, $$COND remains unchanged and still possesses the value in SUBR, which was responsible for generating the first exception.

7.2.2 Programming Notes

Remember that $$COND is set to 0 whenever a normal EXIT statement with no WITH clause is executed. This means that the contents are lost over all subroutine calls, as well as Global Cobol statements that implicitly invoke subroutines. Therefore, if you need to process the condition number, handle $$COND at the very beginning of your exception logic. Either save its value with a MOVE statement; branch on it with a GO TO DEPENDING or a sequence of IF statements, or reflect its handling back to the calling routine using EXIT WITH $$CODE.

Important note: Beware of confusing $$CODE and $$COND. Either of these erroneous statements will prove disastrous:

```
EXIT WITH $$COND

GO TO DEPENDING ON $$CODE
```

The first is wrong because $$COND is PIC 9(2) COMP and EXIT WITH demands a PIC 9(4) COMP variable. The resulting value of $$COND is unpredictable.

The second is wrong because although $$CODE is a computational PIC 9(4) COMP field it is redefined internally by Global System Manager. The condition number, $$COND, is actually held in its junior byte but there is other, non-zero, information in the senior byte. The effect of branching on $$CODE would therefore be to transfer control to a random location within your program.

7.3 Terminating a Program with a Stop Code

You may write the STOP WITH code statement in order to terminate a program with a stop code. The statement is written:

```
STOP WITH code
```

where *code* is an integer literal or PIC 9(4) COMP variable whose value is between 1 and 99 inclusive. The statement causes the Global System Manager error handling logic to gain control and output the error message:

```
$91 TERMINATED - STOP code
```

followed by a diagnostic or debug prompt. The *code* appearing in this message is the value supplied by your STOP WITH *code* statement.

STOP WITH *code* may be written wherever a STOP RUN statement would be valid. Thus, for example, the statement could be part of a format 2 conditional:

```
IF FLAG NOT EQUAL 1 STOP WITH 23
```

Note that it is important that the stop codes you allocate are in the range 1 to 99 because other values are reserved for Global System Manager and Global Cobol itself.

In fairness to the user you should either document the stop codes you assign very thoroughly, or display an explanatory message before terminating the program in error. For example:

```
IF FLAG NOT EQUAL 1
        DISPLAY "CONTROL FILE PROTECTED. UPDATE INVALID"
        STOP WITH 23
END
```

7.4 Coding Routines with a Variable Number of Parameters

The Global Development Cobol Language Manual states that the USING clause of a CALL statement and the ENTRY statement to which it passes control should have the same number of parameters (including zero, when both USING clauses are omitted). However, there is an additional feature of the language, omitted from that manual in the interests of simplicity, which allows you to code routines which take a variable number of parameters. The feature is known as **the entry overflow condition**.

When an ENTRY statement gains control and the number of parameters passed in the calling sequence differs from the number appearing in its USING clause then no parameter passing takes place and an entry overflow condition is generated. This can be detected and processed by an ON OVERFLOW statement immediately following the responsible ENTRY statement.

The overflow logic can itself execute an ENTRY statement with a different number of parameters which may itself either succeed, or generate an entry overflow condition. A further ON OVERFLOW statement can introduce yet another ENTRY statement, and so on. Naturally, if an ENTRY statement generates an overflow condition and is not immediately followed by an

ON OVERFLOW statement the offending program will be terminated in error. Note that it is essential that control is eventually directed to an ENTRY USING statement with the correct number of parameters, or the parameters will remain on the parameter stack which will then not be synchronised with the program.

The entry name which appears as the first parameter of the ENTRY statement is in fact optional, and should be omitted when the statement is only included for the purpose of receiving a different number of parameters.

7.4.1 Example

Subroutine SUBR can be passed zero, one or two parameters by CALL statements of the form:

```
CALL SUBR
CALL SUBR USING AA
CALL SUBR USING AA BB
```

Within SUBR the parameters AA and BB are defined by proper linkage section groups of the same name. The procedure division of SUBR then begins:

```
PROCEDURE DIVISION
*
ENTRY SUBR
        ON OVERFLOW GO TO ENT1
        GO TO PROC0                     * PROCESS NO PARAMETERS
ENT1.
ENTRY USING AA
        ON OVERFLOW GO TO ENT2
        GO TO PROC1                     * PROCESS 1 PARAMETER
ENT2.
ENTRY USING AA BB
        ON OVERFLOW GOTO ERROR          * TOO MANY PARAMETERS
        GOTO TO PROC2                   * PROCESS 2 PARAMETERS
```

7.5 The Operator-id and Authorization Vetting

7.5.1 The Operator-id, $$OPID

The system variable $$OPID is a PIC X(4) field containing the operator-id supplied in response to the prompt:

```
PLEASE KEY YOUR OPERATOR-ID:
```

output at the start of a session. Global System Manager checks that the operator-id you supply to the prompt is not the same as any other operator-id currently signed on. The same operator-id is associated with all jobs initiated by an operator during a session, and therefore competing foreground, background and concurrent jobs started at the same screens have the same operator-id.

On multi-user systems you can use the operator-id to create files which are unique to a particular operator, and which can be accessed by that operator in a different session, possibly

from a different screen. For example, suppose you need to create a set of control files for each operator's private use. You might name them:

CF*oooo*

where *oooo* is the operator-id. Note that on network systems, if the master computer is rebooted without this being done on all networked computers then it is possible for the same operator-id to be signed on twice.

7.5.2 The Authorization Code, $$AUTH

The system variable $$AUTH is a PIC X field containing an alphabetic value in the range A to Z, inclusive. You can supply an optional authorization vetting program to establish the code for each operation allowed to access your system. Alternatively, you may use the standard authorization command, $AUTH, which is supplied as part of Global System Manager and can be customised to associate authorization codes with specific operator-ids. In either case the code assigned remains fixed for the rest of that operator's session and applies to all jobs initiated by him or her, be they background, foreground, concurrent or normal.

Operators allocated authorization codes S, T, U, V, W, X, Y or Z, are treated as "supervisors" by Global System Manager, which means that they are allowed to perform the special functions of the $STATUS command which cancel or restart other users, and that they can key READY to obtain a ready prompt from any Global System Manager menu screen. If you do not supply an authorization program all operators will automatically be allocated authorization code S, and therefore have access to all Global System Manager facilities.

If you use menu customisation (MN) to define the applications available to your users you can use the authorization code to restrict access to menu functions. This is described in your Global Operating Manual.

Whether or not you decide to make use of the authorization code, and how you handle it if you do, is purely dependent on the requirements of your own application. For example, in one very simple scheme only a few very senior operators are allocated code S. The rest are assigned to one of two "grades", these being A, for trainees, still learning how to use the system, and B, for experienced operators allowed access to the full range of transactions. The programs have therefore been written to operate differently according to whether or not:

```
$$AUTH > "A"
```

This means that, from the application viewpoint, the B and S operators are treated identically. It is the following testing within Global System Manager:

```
$$AUTH > "R"
```

which prevents the B operators from performing the special functions only available to "supervisors".

7.5.3 Authorization Vetting Programs

You may optionally provide your own authorization vetting program to grant (or refuse) operators access to the system at the start of each session. It will be entered after the operator has replied to the:

PLEASE KEY YOUR OPERATOR-ID:
[PLEASE KEY TERMINAL CODE (*code*)]

prompt(s), establishing his or her unique operator-id in the PIC X(4) system variable $$OPID.

Your program must validate the operator-id and then call the authorization routine, described below, either to grant the operator access to the system and establish the authorization code to appear in $$AUTH, or alternatively to refuse access to the system. If you have granted access you can terminate the vetting program by either issuing a STOP RUN, in which case control will return to the monitor, or by using the CHAIN statement to invoke another application program directly.

Your program will also be executed for each concurrent partition or when foreground is first entered, to enable you to chain to a particular application program if required. It should test system variable $$PAR to bypass validation of the operator-id when run in foreground (i.e. $$PAR > 1). Under 5.0, where $$PAR is not available, you must use $$TASK to detect foreground/background.

If you decide to refuse access you must end the program with a STOP RUN to return control to Global System Manager which will then write an explanatory message to the screen, and redisplay the operator prompt.

7.5.4 The Authorization Routine, AUTH$

The authorization routine must be invoked once, and once only, for each operator processed by your vetting program. You code a CALL of the form:

```
CALL AUTH$ USING code
```

The *code* is the name of a PIC X field, or a character literal. If you have decided to grant access you must supply an alphabetic character, between A and Z inclusive, which will be established in the system variable $$AUTH as a result of the call. If you wish to refuse access, simply provide a code which is less than ASCII A in collating sequence. For example:

```
CALL AUTH$ USING "?"                    * REFUSE ACCESS
```

7.5.5 Establishing Your Own Authorization Checking Program

Once you have built and tested your authorization vetting program you must introduce it to Global System Manager by running $CUS, selecting Sign-on customisation, and then selecting Authorization Vetting. The name of the current authorization checking program (usually $AUTH) is then displayed, and you are prompted to accept this by keying <CR>, supply the name of another (your own) authorization checking program, or choose not to have any such program by keying <CTRL A>.

7.5.6 Programming Notes

The operator is inhibited from running concurrent jobs until you have called AUTH$ to grant him or her access to the system. Your authorization vetting program is then entered for each partition, so that you can chain to a specified program.

The system variable $$MEDF contains the menu entry selected by the Global System Manager menu if the menu is CHAINed to. It is used by $AUTH and may be useful in other authorization programs.

We recommend that you prevent the program being run, by mistake, from a ready prompt, by introducing the statement:

```
IF $$RUN POSITIVE STOP RUN
```

at the start of the procedure division once you are ready to incorporate it into your system.

Note that your program should always be re-linked with the latest version of AUTH$ where possible.

One common way of designing an authorization vetting program involves using an indexed sequential control file. Each record then typically contains:

- The operator-id of an operator entitled to use the system (this field is used as the key to the file);
- A password associated with the operator. This field may possibly be scrambled so that it cannot easily be interpreted if the file is dumped or printed;
- The authorization code for the operator;
- Possibly additional application-dependent information which may, for example, be used in determining a program to be chained to once access has been granted.

The program first of all validates the operator-id by testing if there is a record on file for the operator: if there is not, the operator is refused access. When a record is present, the routine prompts for the password. If the password matches the one on the file the operator is granted access using the authorization code held in his or her record. Otherwise, after, say, three attempts at supplying the correct password, the operator is refused access to the system.

7.6 Supervisor Programs

A supervisor program is a program executed by Global System Manager instead of displaying the ready prompt. A typical example of such a program is the $MH command, described in the Global Operating Manual, which displays a menu of the available applications and then executes the one selected. At the end of the application the supervisor program is automatically re-invoked, so that another selection can be made.

7.6.1 The Supervisor Program Name, $$SVSR

The system variable $$SVSR is a PIC X(8) field in which you can establish the name of a supervisor program. If the field is blank, as it is initially unless customised as explained below, then there is no supervisor program, and the READY prompt will be displayed. $$SVSR is reset to spaces whenever the supervisor program is invoked, so the supervisor program must re-establish its program name in $$SVSR each time it is executed, unless it wishes to relinquish control, when it should execute a STOP RUN statement with $$SVSR set to spaces, to cause the READY prompt to be displayed.

When the supervisor program is invoked by Global System Manager it is invoked with logging off allowed. If you do not wish the user to be allowed to log off while in the supervisor program, the supervisor program must immediately call the NLOGF$ routine.

7.6.2 The Initial Menu Customisation Instruction

You can specify a supervisor program to be executed automatically following initial sign-on by using the $CUS Sign-on customisation option, and then selecting the Initial Menu entry. The name of the menu program currently entered after sign-on (usually $MH) will be displayed, and you can accept this by keying <CR>, key <CTRL A> to have no such program (if you want to go straight to the ready prompt rather than a menu) or supply the name of another (your own) supervisor program to be entered automatically after sign-on. The supervisor program specified will be executed in each concurrent partition.

7.6.3 Programming Notes

The supervisor program is not automatically invoked by Global System Manager if job management is in control, as the reply to the ready prompt will be supplied from the dialogue table. This allows a job to execute several programs, and then return to the supervisor program when complete.

A subject program returns control to the supervisor program just as it normally returns to Global System Manager, by executing a STOP RUN or an EXIT statement from the highest level of control. If the subject program displays an explanatory message to indicate why it has terminated it should precede this by the BELL statement. For example:

```
IF ZMAX > 1000
        BELL
        DISPLAY "PROGRAM TABLE CAPACITY EXCEEDED"
        STOP RUN
END
```

BELL sounds the console bell and at the same time sets a flag so that a comma prompt is inserted into the dialogue just before the supervisor program regains control. (The flag has no effect when the program runs directly under Global System Manager.) The comma prompt enables the operator to read the terminating message even if the supervisor program itself, like $MH, begins by clearing the screen.

7.7 User-Written System Requests

Under V6.0 and later versions of Global System Manager, you can define your own system request to be called by the use of <SYSREQ> E or by a CMND$ call.

7.7.1 Requirements for writing a system request

A system request is a normal Global Cobol program. It must be linked at location #1300, and the total occupied memory of the program must not exceed address #7000 (23.25K total size). The program may LOAD and EXEC (but not CHAIN) other programs, but these must obey the same requirements on use of memory (subsequent programs may begin after #1300 but must still not extend beyond address #7000). On completion the system request must EXIT back to the calling process from its highest level of control. It must not call EXIT$. It must under no

circumstances issue a STOP RUN, nor may it permit any routines it calls to issue STOP codes (in particular this means that any FDs being used must be coded with OPTION ERROR).

7.7.2 Restrictions on the system request

When the system request is invoked by Global System Manager, certain aspects of the operating environment (specifically the contents of memory locations #1000 to #7000, the state of the various display and accept flags, the contents of the screen image and related information, and those system variables involved with program loading) are saved. When the routine exits such information will be restored.

If the system request makes any other changes to the operating environment then it is responsible for restoring the situation before exiting. In particular it must close any files which were opened and replace any unit assignments which were altered (in general we recommend that unit assignments are not altered by a system request).

The system request must make no assumptions about the state of the system when it is called. In particular it must be passed any unit addresses it will require via the passed data area, and not determine them by calling ASSIG$ or using symbolic unit assignments.

The system request may not modify any user memory outside the area #1300 to #7000, nor change any system variable which has not been preserved by Global System Manager before it is invoked. The only exception to this is that the system request may make normal use of free memory, provided that such memory is returned before the request exits. Great caution should be exercised with this possibility however, as many places from which the request is invoked will already have acquired all of the free memory for their own purposes. We would not recommend use of free memory from within a system request except in unusual circumstances, or as an optional performance improvement.

In pre-V7.0 Global System Manager, when the system request is entered the screen will be in Help mode, with the screen display untouched. Before exiting, the system request should check the $$VERS flag and issue the sequence to end Help and redisplay the screen if appropriate (#1B2C). In pre-V7.0 Global System Manager, Global System Manager will in any case exit from Help mode when control is returned to it, but will leave any information displayed on the screen until it is cleared by the user. It may be sensible to make use of the flag $$HCLR within the system request.

Where a system request is accepting only a small amount of information it will normally be good practice to use CHECK$ to check for terminated replies to prompts, and use CHAR$ to input the information, thereby avoiding unnecessary displays (much as the Global System Manager Calendar system request does.)

Where the system request is passing back information, it must do this as a single call on TYP$ or TYPF$ (multiple calls may be used, but note that each call of the TYP$ routine inserts information at the start of the type-ahead buffer) and is hence limited to the size of the type-ahead buffer. The buffer length can be changed by using Configuration maintenance and any special requirements should be mentioned in the accompanying documentation and the installation process, so that the user may increase the type-ahead buffer size. If the available type-ahead buffer is too small then the request should fail gracefully, by executing a BELL verb if it has minimal dialogue, or providing an explanatory error message.

Information passed back should be in a suitable format for a typical usage of it. In practice this normally means separating each distinct field by a carriage return (#0D). The last item of data returned should not be terminated by a carriage return (this is a standard in other system requests, and allows easy editing of the last or only data item if this is appropriate).

7.7.3 Installing the system request

When installing the system request it is necessary to set up sufficient information both for Global System Manager to recognise the existence of the system request, and for the system request itself to be able to function. For simplicity, and because Global System Manager clearly needs to be able to pass any parameters to the system request when it is loaded, all this information is held in a single centralised file.

Information about system requests is held on a data library called $$UREQ located on unit $M. You may use $URMAIN or alternatively, write a simple program using the Data Library Access Method (DLAM), described in chapter 7 of the Global Development File Management Manual, to create or update this file. The records held in this library are each 100 bytes long and have the following format:

```
01    DLREC
  02  DLNAME          PIC X(10)
  02  DLTTL                 PIC X(30)
  02  DLOPID          PIC X(4)
  02  DLDATE          PIC 9(6) COMP
  02  DLTIME          PIC X(8)
  02  FILLER                PIC X(5)
  02  DLPROG          PIC X(8)
  02  DLUNIT          PIC X(3)
  02  DLLIB           PIC X(8)
  02  DLFMOD          PIC 9 COMP
  02  DLDATA          PIC X(20)
```

The initial fields (DLNAME, DLTTL, DLOPID, DLDATE, DLTIME, and the X(5) FILLER) are those required by the data library access method.

The DLPROG field is used to identify the program to be loaded. Notice that the DLNAME field is used to select the program, hence there is no firm tie between the function name and the program name, and so different system requests may invoke the same program with different parameters. The DLTTL field must be set up so as to clearly specify the function performed by the system request, as it will be listed when the requests are listed and is the only obvious clue the operator will get as to the function of the request. If there are variable data areas which can be set up at installation time it is strongly suggested that the operator be able to provide his own title.

The DLUNIT field identifies the unit where the program resides (it must be a unit address), and the DLLIB field the library in which it is to be found (if it is not in a library set DLLIB to the value "P.").

The DLFMOD field is set up to indicate whether the system request requires formatted mode established before it is invoked. If it is set to 0 no special action takes place. If it is set non-zero then Global System Manager will establish formatted mode (without clearing the screen) before

invoking the system request. Positive values will cause colour to be enabled, and negative values will cause colour not to be enabled (cf CLR-N$).

The DLDATA field is 20 bytes of information which may be set up by your installation process and is passed to the system request program when it is loaded (the 20 bytes are placed at location #1300, and then your request is loaded, so you would access them by having uninitialised data items at the start of your system request program).

Note that when your system request is loaded $$INDE is initially set to #1380.

7.7.4 Invocation of user-written system request

To invoke an installed user-written system request you must first key <SYSREQ> E, to load the system request entry program. This will then prompt you for the name of the system request you wish to use (the prompt will be suppressed if the response has been typed ahead), and if you key <CR> you may list the contents of the data library and make a selection.

Finally, before executing your system request, the entry program will restore the original screen display, enable help mode, and establish colour combination 6 (the standard help colour).

7.7.5 Using a system request as a program overlay

Under some circumstances you may wish to use the system request mechanism to add an extra function to an existing software package. You would normally do this to avoid some of the overheads associated with reorganising an existing overlay structure.

Such a system request overlay must obey the same operating memory constraints as an ordinary system request, except that as you will have complete control over the operating environment there is no difficulty with accessing memory outside the range #1000 to #7000, and the overlay may modify such memory, and system variables, as it sees fit. The principal constraint on such a situation is that the request may not process memory in the range #1000 to #7000, as this area is reserved for the processing of system requests and its contents may be unpredictable.

To identify to Global System Manager which program is to be loaded as a system request overlay, you set $$PGM to be the program name (and $$LIB to identify the program library if necessary). It will be loaded from the currently assigned $P unit, and its load will alter the value contained in $$EPT. To load the program you call the CMND$ routine passing the special value "*" as the request name.

When your overlay is run, Global System Manager saves the System Area data as before, and places the screen into help mode, so as to minimise the possibility of interference between the system request overlay and the existing programs. When your system request overlay exits, Global System Manager will restore the saved system information, and return control to the statement following the call of CMND$.

Important Note: Under Global System Manager V6.0, the space available for system requests only extends up to address #5000 (meaning that there is 15.25K available). A tailoring zap is available for such systems which extends the space available up to 23.25K (address #7000).

7.7.6 Special Handling for DMAM ON FAILURE routines

In the normal course of events DMAM ON FAILURE routines must terminate with a STOP RUN (see the Global Development Data Management Manual). This is not acceptable within a system request so special handling must be included within the ON FAILURE routine when DMAM is included in a system request. This handling is only available on V6.2 or later version of Global System Manager. You must code the following:

```
DATA DIVISION
77      V62PTR      PIC PTR
        VALUE       #1010
77      V70EPT      PIC PTR
        VALUE       #E065
LINKAGE SECTION
77      V62EPT BASED V62PTR PIC PTR
*
* At the end of the ON FAILURE routine:
*
IF $$VERS < 5 STOP RUN                          * No special logic if pre-V6.2
MOVE LOW-VALUES TO $$CLMM
IF $$VERS = 5                                   * If V6.2
        MOVE V62EPT TO $$EPT
ELSE                                            * Else must be V7.0 or greater
        MOVE V70EPT TO $$EPT
END
CALL EXIT$
```

This technique is only available using the V8.1, and later, compiler.

7.8 SVC 14 - Search for Lowest Key

Before using the following SVC instruction you should refer to chapter 6 of the Global Development Cobol Language Manual for a general description of the System Service calls.

The SVC 14 instruction is an assembler routine, used by the SORT verb. It searches through a table and returns an index to the table entry which has the lowest key, where the key field is a character field of any length. It can be useful if you need to write your own special purpose sort routine.

To invoke the service you must code:

```
SVC 14 USING ts table
```

where *ts* is a table sort control block (as used by TSRT$ in section 3.5) and *table* is the table to be searched. The *ts* block consists of five PIC 9(4) COMP fields:

- the length of each table entry in bytes;
- the number of table entries;
- returned index to the entry with the lowest key;
- the start of the key within the entry (first byte = 1);

- the length of the sort key in bytes.

8. Storage Management Facilities

8.1 Concepts and Terminology

The Global Development Cobol Language Manual explains how the Global System Manager memory region is subdivided into two areas, one occupied by the permanently resident Global System Manager nucleus and monitor and the other, the user area, serving as a temporary store for application programs and commands which are loaded as required.

In fact, as Figure 8.1 shows, the actual situation as regards the user area is somewhat more complicated. The user area is itself dynamically partitioned into an occupied area and a free area. Furthermore, the occupied area may contain two types of storage, the program area and the work space.

8.1.1 The Occupied User Area

The occupied user area is that part of the user area which a program has the right to access. It consists of the area in which the program itself resides together with any work space acquired from the free area. In a multi-user system the occupied user area is the only part of the area guaranteed to be preserved when the program is suspended. The implication then is that the program must on no account access the free area directly. The FREE$ system routine, described in section 5.10, provides a variety of different storage management services allowing you to free the occupied user area, establish the size of the program area, obtain work space from the free area and return previously acquired work space to the area.

8.1.2 The Program Area and the Loader

When there is no work space in use the program area and occupied user area are one and the same. The loader (entered by a CHAIN, EXEC or LOAD statement) increases the size of the program area, if necessary, so that it includes the highest location so far loaded by the current job. This means that, left to itself, the loader will tend to increase, but will never decrease, the size of the occupied user area which will therefore reflect the maximum amount of storage in use during the life of a job.

FREE$ provides functions which allow you to find the size of the program area, and to set the size of the program area without using the loader. You can use them to determine the current program size before an EXEC for a program loaded higher in memory, and later to release the space occupied by the executed program by resetting the program area to its original size. Other uses are described later.

In some cases the loader's storage management, quite suitable for simple jobs consisting of only a single program, causes problems. For example, consider a job which starts with a 16K program which, after a short time, chains to a 6K program which is in control until the job ends. Then, for the life of this job the occupied user area will be considered to be 16 Kbytes in length even though only 6K are actually in use.

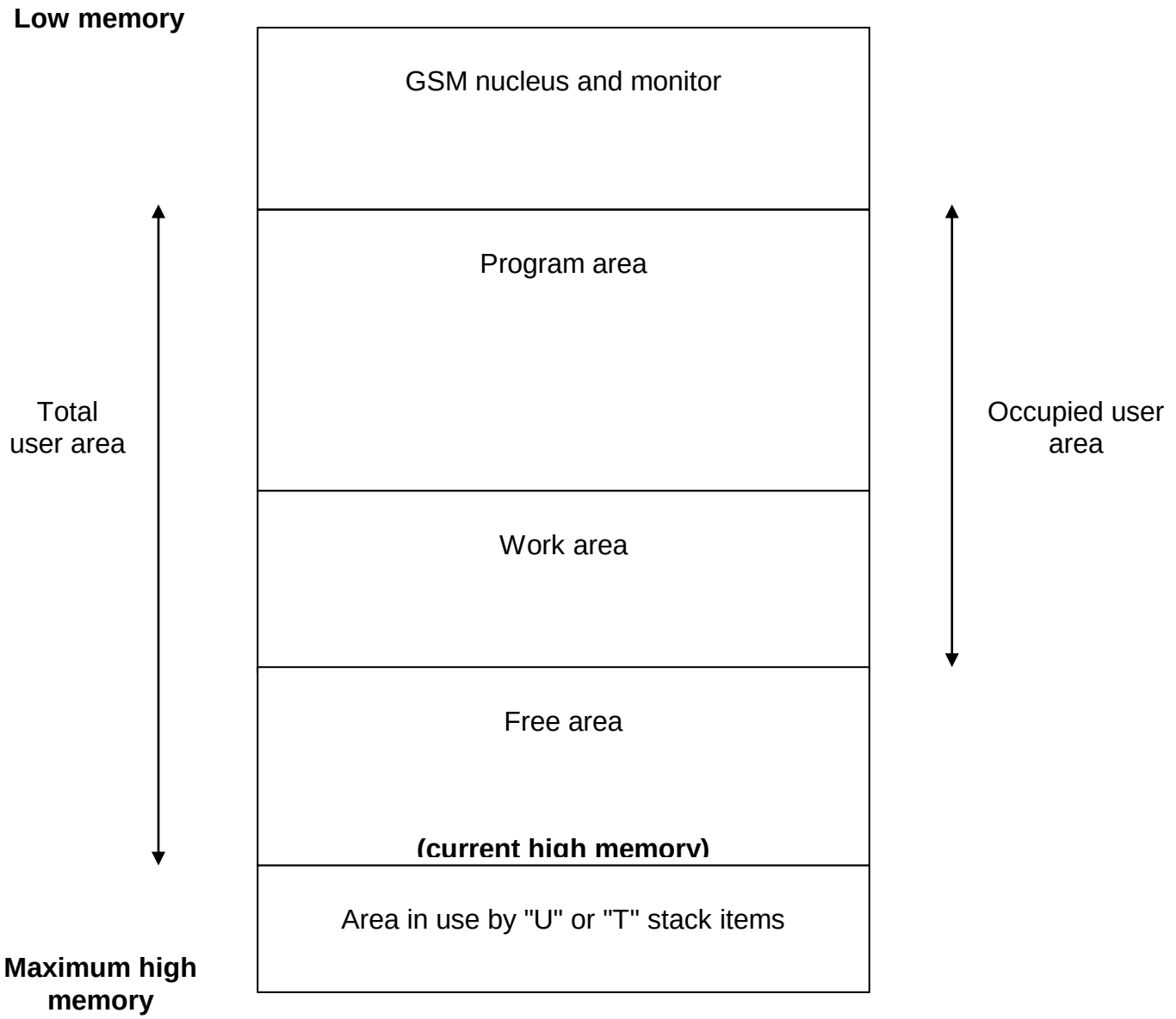


Figure 8.1 - The Global System Manager Memory Region

If the smaller program employs any of the following Global Cobol statements:

CALL COPY$...	to copy a file
CALL REST$...	to restore master data from a backup cycle
CALL SAVE$...	to preserve master data on a backup cycle
SORT...	to invoke the sort

then it will perform less than optimally, because each of these makes temporary use of all currently free storage, and functions better the more memory that is available. However, because of the earlier 16K program, 10K of the free storage actually present is lost as far as these memory-hungry routines are concerned.

There is also a performance implication in certain multi-user environments. Because of the earlier 16K program an unnecessary 10K bytes of the user area may be written to and from the swap file whenever the job is suspended and resumed.

To overcome the problems brought about by the loader's rather simple approach to storage management, a system variable $$REL is provided which, if set, will cause the loader to reset the program area size when a program is next loaded (see section 6.2.8). You should only set it immediately before invoking the loader by a LOAD, CHAIN or EXEC statement. In the example the 16K program would set $$REL immediately before executing the CHAIN statement to invoke the 6K program. This will cause the size of the occupied user area to be correctly re-established (as 6K) so that functions such as the sort perform optimally, and no storage is unnecessarily swapped in the multi-user environment.

8.1.3 Work Space Management

The principal function of the FREE$ routine is to enable programs to obtain and release working storage dynamically. Storage thus acquired is obtained from the beginning of the free area and becomes part of the work space, at the high end of the occupied user area, which is increased in size accordingly. Once a job has obtained storage in this way its work space is said to be **in use**. This effects the job's capability for loading programs.

Once the work space is in use the job can only load programs into the program area, which **precedes** the work space. Should the job attempt to load a program into the area the work space occupies, or into the subsequent free area, it will be terminated in error. You can only increase the size of the program area when the work space is **not** in use. Because of this you may wish to use the function of FREE$ which allows you to increase the size of the program area to establish the maximum area you need **before** acquiring any work space.

You can use FREE$ to return work space to the free area. If, by doing this, you reduce the number of bytes to zero, the work space becomes **empty** and is no longer considered to be in use. The loader will once again determine the size of the occupied user area, increasing it whenever a program is loaded which does not fit within the currently allocated program area.

If you wish to change the size of the current program area you must first release all the allocated work space. This can be done by using a special function of FREE$ immediately before a LOAD, CHAIN or EXEC statement.

Note that work space is managed as a last-in, first-out (LIFO) store. If you acquire three 1000-byte areas, A, B, and C, in that order, A will immediately precede B in memory, and B will

immediately precede C. If you then release 1000 bytes it will be the storage last obtained (i.e. area C) that will be returned to the free area. This is, of course, implied by the geometry of Figure 8.1.

8.1.4 Library Index Record Considerations

Whenever a program library is attached, or a program is loaded when such a library is attached, or a command program is loaded, the loader requires to read the 1134-byte library index record into a buffer located somewhere within the user area. The index record is only required at the very beginning of the loading process, and an incoming program can, if necessary, overwrite the record. It is normally read into the top 1134 bytes of the Global System Manager memory region, at the high end of the user area. There are no problems with multi-user swapping since Global System Manager ensures that the job cannot be suspended during the short time in which the index record is being processed.

The index record handling so far described is satisfactory for programs which do not dynamically acquire work space and which, if they do adopt an overlay structure, conform to the rules described in the section on overlay construction in the Global Development Cobol User Manual. These rules indicate that the deeper the level of an overlay, the higher the address space it should employ. They imply that, even if a large program occupies part of the high memory used by the index record, the latter is completely overwritten by the code of the deepest-level incoming overlay (except in the very unlikely event that the overlay itself begins at a higher address than the index record buffer).

Normal index record handling can interfere with a program which acquires a work space and then continues to load overlays, because the work space will be allocated from the high part of the user area and, depending on the actual amount of storage available, the index record may or may not overwrite the work space itself. This is particularly a problem with the sort, if you decide to handle the input and output processing in two separate overlays. Left to itself the loader will read the index record into the very work area used to hold critical sort information when it comes to acquire the output overlay.

To allow you to overcome this type of problem Global System Manager provides the system variable, $$INDE, a pointer that you can modify in order to control where the library index record is to be placed (see section 6.2.9). You only require to use $$INDE in this way when the normal handling might cause the program area or the work space to be corrupted. You set the pointer to address a buffer within the area to which your overlays are to be loaded. Then the record is simply overwritten by an incoming module and it does not corrupt any part of the remainder of the program. (You may also require to set $$INDE to direct the library index record into a data area within your program, so that you can inspect it to see which programs are actually present in a library: this use of $$INDE is described in 7.1.)

You should note that whenever a program acquires the whole of memory, and then requires to load an overlay, normal index record handling will **always** corrupt part of the work space, and so it is essential to set $$INDE to address the overlay area itself. Programs that do not acquire work space, but adopt unconventional overlay strategies not conforming to the deepest-level, highest-address rule, may also need to manipulate $$INDE. In general, such programs should be avoided.

There is an example of the use of $$INDE, in conjunction with some of the other storage management functions provided by FREE$, in section 8.2. In addition, the programming notes

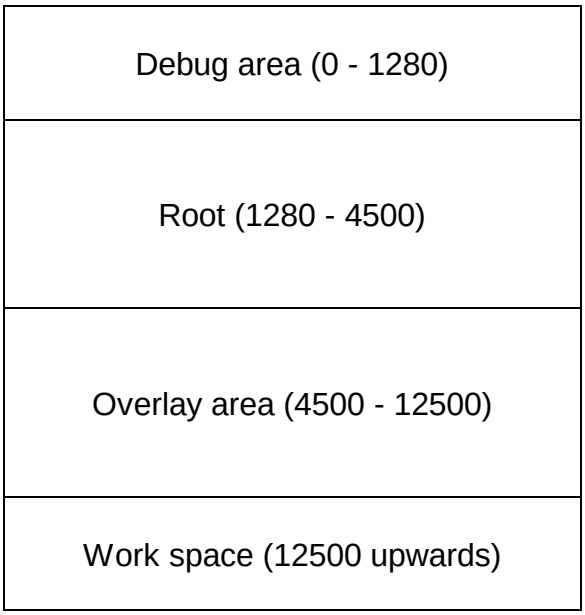
on the use of the sort in the Global Development File Management Manual illustrate how $$INDE should be used when employing input and output processing overlays.

```
PROGRAM OVLAY1
DATA DIVISION
*
*        FREE SPACE REQUEST AREA
*
01      FM
  02    FMFUN       PIC 9(6) COMP          * FUNCTION (1-PGM, 2-GET, ETC.)
  02    FMSIZE      PIC 9(6) COMP
  02    FMALL       PIC 9(6) COMP
  02    FMPTR       PIC PTR
*
LINKAGE SECTION
01      WS
  02    WSPTR       PIC PTR
  02    WSSIZE      PIC 9(6) COMP
*
PROCEDURE DIVISION
ENTRY GETSTO USING WS
*
*        SET PROGRAM AREA TO 12500
*
        MOVE 1 TO FMFUN
        MOVE 12500 TO FMSIZE
        CALL FREE$ USING FM
        ON EXCEPTION STOP WITH 1          * NO STORE FOR PROGRAMS
*
*        GET MAXIMUM AVAILABLE WORK SPACE
*
        MOVE 2 TO FMFUN
        MOVE 32766 TO FMSIZE
        CALL FREE$ USING FM
        ON EXCEPTION
                IF FMALL < 1000 STOP WITH 1  * NOT 1000 BYTES
        END
        MOVE FMALL TO WSSIZE
        MOVE FMPTR TO WSPTR
*
*        SET $$INDE TO ADDRESS THIS OVERLAY'S PROC DIV, THEN EXIT
*
        POINT $$INDE AT GETSTO
EXIT
ENDPROG
```

Figure 8.2 - Example program using storage management facilities

8.2 Example Program using Storage Management

A program consists of a 3000-byte root segment and a number of overlays. None of the overlays are greater than 8000 bytes in length. The job acquires as much work space as there is storage available, up to the limit of 32766 bytes. Suppose the memory map is to appear as shown below:



Then Figure 8.2 shows the coding of an initial overlay, responsible for establishing storage management parameters, which is to be invoked by the statement:

```
EXEC OVLAY1 USING WS
```

WS is a parameter area in which the overlay returns a pointer to the work space, together with its size. The job will be terminated with stop code 1 if there is insufficient space for the program area, or less 1000 bytes of work space available. When all is well $$INDE is set up so that the library index record used in subsequent load operations will occupy the part of the overlay area used by the procedure division of OVLAY1.

9. Scientific Calculation Facilities

9.1 Floating Point Numbers

9.1.1 Internal Format

Global Cobol holds a floating point number to be used in calculations as a 6 byte quantity consisting of a PIC S9(2) COMP exponent followed by a PIC S9(11) COMP representation of the mantissa. There is an implied decimal point immediately to the right of the senior digit of the mantissa (as though it had an S9(1,10) picture). Floating point 0 has an exponent and mantissa both 0. For example:

FP value	*Exponent*	*Mantissa*
0	0	0
1	0	10000000000
-0.01	-2	-10000000000

This representation supports floating point numbers with 11 decimal digits precision. The smallest positive floating point number is 1×10^{-99} and the largest $9.9999999999 \times 10^{99}$. Similarly, the largest negative floating point number is -1×10^{-99} and the smallest $-9.9999999999 \times 10^{99}$.

9.1.2 Display Format

Floating point numbers to be displayed or printed are held as PIC X(17) fields conforming to draft ISO standard 6093.2. Each such **standard numeric** field is an ASCII character string of the form:

*sndnnnnnnnnn*E*tmm*

where *s* is either space, if the number is positive, or "-" if it is negative; the quantity *ndnnnnnnnnn* is a decimal number between 1.0000000000 and 9.9999999999 inclusive; E is that letter; the quantity *t*, the sign of the exponent, is either "+" or "-"; and *mm* is the magnitude of the exponent, in the range 00 to 99 inclusive. The "+" value of t is always used when mm is 00. For example:

FP value	*Standard numeric representation*
0	0.0000000000E+00
1	1.0000000000E+01
-0.01	-1.0000000000E-02

If $CUS Decimal Point customization has been applied to specify that a comma rather than a period be used as the decimal point, then that convention will apply to the standard numeric form; for example, floating point 0 would be appear as 0,0000000000E+00. Note, however, that if you code a decimal point in a VALUE clause or COMPUTE statement a period **must** be used irrespective of the Decimal Point customization.

9.1.3 Defining Floating Point Numbers

Floating point numbers or tables may be defined in the data division of a Global Cobol program by using the PIC FLT picture clause.

Important note: If no VALUE clause is supplied each number will be initialised to spaces if initialisation is required, so you are recommended to give all PIC FLT items a value (LOW-VALUES will set the field to floating point 0).

For a table each occurrence requires 6 bytes, so in the example below FPITEM takes just 6 bytes but FPTAB needs 24:

```
01    FILLER
  03  FPITEM     PIC FLT
  03  FPTAB      OCCURS 4 PIC FLT
```

To initialise a floating point number to a value other than 0 you must use a VALUE clause of the form:

VALUE "*sn...n*E*tmm*"

where the string in quotes is made up as follows:

s is the optional sign of the mantissa, "+" or "-", which may be omitted if the mantissa is positive.

n...n is a decimal integer or fraction defining the absolute mantissa of the floating point number. If there are more than 11 digits (ignoring leading zeros) the 12th will be used to round the preceding 11 digit value and in calculating the eventual exponent. The 13^{th} and subsequent digits may be used in determining the exponent but do not contribute to the mantissa.

E is optional, but must be coded if an explicit exponent is to be specified. When omitted the subsequent *tmm* part must be omitted too.

t is the optional sign of the exponent, "+" or "-" , which may be omitted if the exponent is positive.

mm is 1 or 2 digits specifying the magnitude of the exponent.

You will see that this definition includes the standard numeric form discussed earlier, but is more flexible; you may write VALUE "12" instead of VALUE "1.2000000000E+01", for example. When you require to initialise a table of floating point numbers the various occurrences are set up by separate VALUE clauses coded consecutively in the same way that an ordinary computational table is established. For instance:

```
77    FTAB OCCURS 5 PIC FLT
        VALUE     "1E-6"
        VALUE     ".001"
        VALUE     "1"
        VALUE     "1000"
        VALUE     "1E6"
```

The compiler will reject an invalidly formatted VALUE clause, or one whose magnitude is greater than $9.9999999999 \times 10^{99}$, with an error message. Floating point 0 will be set up for valid values whose magnitudes are less than 1×10^{-99}.

For example:

VALUE	1	invalid, no surrounding quotes
VALUE	" 1"	invalid, blank within quotes
VALUE	"E12"	invalid, no mantissa
VALUE	".1E100"	invalid, exponent too long
VALUE	"10E99"	invalid, too large
VALUE	".9E-99"	floating point 0

9.1.4 Procedure Division Statements

With the exception of the COMPUTE statement used to execute scientific programs (explained later) only the CALL, ENTRY, SEARCH, MOVE and IF statements can be used with PIC FLT numbers. Such numbers may be passed as parameters to subroutines; this is how they are employed by CALL and ENTRY. For example:

CALL *rtn* USING *fp-number*

should transfer control to an ENTRY statement of the form:

ENTRY *rtn* USING *fp-number*

where *fp-number* is defined in the linkage section in the normal way.

A floating point number may be the key of a SEARCH statement; the table control area must then of course define the key length of 6 bytes. Although in theory you could use a SCAN statement in the same way, in practice this would give unpredictable results because a byte by byte comparison of 2 floating point numbers only gives the same result as a floating point comparison in certain cases (e.g. if both are equal or both are greater than 1).

MOVE can transfer one floating point value to another; in addition the form:

MOVE LOW-VALUES TO *fp-number*

sets the number involved to floating point 0. The IF statement should only be used to check whether floating point numbers are equal. Just as with the SCAN statement, using IF to test whether one floating point number is greater or less than another will generally give wrong results; such comparisons must be performed using COMPUTE. However the form:

IF *fp-number* = LOW-VALUES

is very useful and provides a simple means of checking whether a floating point number is 0.

9.2 Floating Point Conversion and I/O Routines

The routines described in this section allow you to convert Global Cobol computational fields to floating point, and vice versa; to convert floating point numbers to standard numeric form; and to display and accept floating point numbers. Throughout this description:

fp-number is any PIC FLT floating point number

comp is any computational field

s-numeric is any PIC X(17) standard numeric field

9.2.1 Convert Accumulator to Floating Point, AC-FP$

This routine may be used to convert a Global Cobol computational field of any format to floating point, providing that the magnitude of the number is less than 99999999999.5. An exception is returned and the floating point number supplied as a parameter remains unchanged if this limit is exceeded. The calling sequence is of the form:

```
$LOAD comp
CALL AC-FP$ USING fp-number
ON EXCEPTION
        Logic to handle the comp too large case
END
```

Note how the intermediate code $LOAD instruction (described in 9.3.1) must be used to set the accumulator to the required value before the routine is called.

9.2.2 Convert Floating Point to Accumulator, FP-AC$

This routine allows you to convert a floating point number into any Global Cobol computational format, providing that the magnitude of the number is less than 1011 and the computational format is of sufficient capacity to store it. The floating point number is read only as far as the routine is concerned. In the typical calling sequence below an exception will be returned and the computational number will remain unchanged in either error condition:

```
CALL FP-AC$ USING fp-number
$STORE comp
ON EXCEPTION
        Logic to handle fp-number too big or comp too small
END
```

Note how the intermediate code $STORE instruction (9.3.1) is used immediately after the call to save the value returned in the accumulator by FP-AC$ in the computational field. If the routine has found that *fp-number* is too big it will return an exception which will cause the $STORE instruction to be suppressed and the immediately following exception handling logic to be entered. In the other error situation FP-AC$ completes normally and the $STORE is attempted but, because *comp* is too small to hold the value, an OVERFLOW condition is raised which causes the exception handling logic to be entered. (Overflow and exception conditions are the same as far as user error handling is concerned; indeed, the ON OVERFLOW and ON EXCEPTION statements generate the same code and may be used interchangeably.)

9.2.3 Convert Floating Point to Standard Numeric, FP-SN$

This routine converts the floating point number supplied as its first parameter into PIC X(17) standard numeric form in its second parameter. The floating point number is read only as far as FP-SN$ is concerned. An exception will be returned if the floating point number contains an **improper** value. (This can only happen if the number has not been initialised, or has become corrupt; for example, the 6 bytes 07FFFFFFFFFF do not represent a proper floating point number since the exponent has the value #7F (i.e. 127) which is out of range.)

The complete typical calling sequence is:

```
CALL FP-SN$ USING fp-number s-numeric
ON EXCEPTION
        Logic to handle an improper floating point number
END
```

In practice the exception handling logic is normally omitted since an exception usually means that the user program itself is in error.

9.2.4 Display Floating Point, DFP$

This routine displays the floating point number supplied as its parameter as a 17 character standard numeric output field. However, if an improper floating point number is supplied an exception is returned and the erroneous output suppressed. The complete typical calling sequence is:

```
CALL DFP$ USING fp-number
ON EXCEPTION
        Logic to handle an improper floating point number
END
```

In practice the exception handling logic is normally omitted since an exception usually means that the user program itself is in error.

In scroll mode DFP$ acts like DISPLAY...SAMELINE, so you must use an ordinary DISPLAY statement before DFP$ is called to position the cursor appropriately. Similarly, when using the routine on formatted screens you must execute DISPLAY...LINE to ensure that the cursor is positioned correctly before it is called, since any previous accept operation will have left the cursor at an indeterminate location.

9.2.5 Accept Floating Point, AFP$

This routine outputs the prompt character (as defined by $$PROM), then accepts information from a 17 character input area and, if all is well, converts it and stores it in the floating point number supplied as its parameter. If the null string is keyed the routine returns an exception and the floating point number remains unchanged. System variable $$EOF defines the actual keystroke used for the null response, as well as the keystroke used to end a non-null reply. The typical calling sequence is:

```
CALL AFP$ USING FP
ON EXCEPTION
        Logic to handle null response
END
```

Non-null input keyed by the operator must be similar in form to the data appearing between the quotes of a valid PIC FLT VALUE clause, except that there may be leading spaces (which will be ignored) and either a period or a comma can be used as the decimal point, depending on the operator's preference. Erroneous input will be rejected with a warning prompt until the operator supplies a correct floating point number.

In scroll mode AFP$ acts like ACCEPT so you must use DISPLAY statements before AFP$ is called to position the cursor appropriately. (Normally you use such a statement to output the text of the prompt of which the AFP$ accept forms part.) Similarly, when using the routine on formatted screens you must execute DISPLAY...LINE to ensure that the cursor is positioned correctly before it is called, since any previous accept operation will have left the cursor at an indeterminate location.

9.2.6 Convert Input to Floating Point, IN-FP$

This routine allows you to obtain floating point information as a PIC X(17) field (e.g. by an ACCEPT or MAPIN) operation, then validate it and, if it is satisfactory, convert it to floating point. The routine can usefully be employed in Global Screen Formatter validation routines to check ordinary PIC X(17) fields keyed by the operator and convert them to floating point numbers.

The PIC X(17) field, *input-string*, say, must be similar in form to the data appearing between the quotes of a valid PIC FLT VALUE clause, except that there may be leading spaces (which will be ignored) and either a period or a comma may appear as the decimal point. If the *input-string*, which is read only as far as IN-FP$ is concerned, does not conform to these rules an exception will be returned. The typical calling sequence is therefore:

```
CALL IN-FP$ USING input-area fp-number
ON EXCEPTION
        Logic for when the input is invalid
END
```

9.3 Scientific Calculations

So far we have seen only how to convert floating point numbers from one format to another and how to input and output them. This section explains how calculations are performed by executing scientific programs using the COMPUTE statement.

9.3.1 The COMPUTE Statement

The COMPUTE statement is coded:

```
COMPUTE "scientific program" [fp-number]
```

The *scientific program* is a character string describing the computation to take place; it must be enclosed in quotes. The optional second parameter is the name of a floating point number within the user program which may feature as an additional variable in the calculations. If this parameter is omitted the program is confined to manipulating the 26 **internal** variables, a to z (or A to Z), so called because their values are held within the subroutine responsible for the COMPUTE statement. When the second parameter - *fp-number* - is present it defines an **external user** variable which can be employed to pass a value to COMPUTE or extract one

from it. The user variable is referred to as $ whenever it appears in a scientific program. For example:

```
COMPUTE "a=1; b=-3; c=2; x= [-b + sqr(b^2-4*a*c)]/2/a"
```

finds the largest positive root of the quadratic equation $x^2 - 3x + 2 = 0$ using the well known formula from elementary algebra. The result is stored in internal variable x. To return this value to FPROOT, a PIC FLT floating point number in the user program, code a second COMPUTE statement:

```
COMPUTE "$=x" FPROOT
```

Alternatively, the calculation can be performed as a single statement not involving x:

```
COMPUTE "a=1; b=-3; c=2; $= [-b +sqr(b^2-4*a*c)]/2/a" FPROOT
```

Note that any COMPUTE statement may be terminated with an exception if the scientific program involved cannot be executed successfully. For instance, suppose (more realistically) that a, b and c have been set up previously. Then the statement:

```
COMPUTE "x= [-b + sqr(b^2-4*a*c)]/2/a"
```

could fail because a was 0, (**divide by 0** error), or because $b^2 - 4ac$ was negative (**invalid function argument**) corresponding, respectively, to the cases when the equation is linear or has no real roots. Alternatively the program might have failed to assign a, b or c previously, causing the COMPUTE to fail with an **undefined variable** exception.

We shall discuss diagnostics more thoroughly later. For the moment, simply bear in mind that there is always a possibility that a COMPUTE statement may return an exception just as an ordinary arithmetic statement may suffer overflow.

Function	Result returned
abs(*x*)	|*x*|, i.e. the absolute value of *x*
acs(*x*)	$\cos^{-1} x$ in radian
act(*x*)	$\cot^{-1} x$ in radians
asn(*x*)	$\sin^{-1} x$ in radians
atn(*x*)	$\tan^{-1} x$ in radians
cos(*r*)	the cosine of *r* radians
cot(*r*)	the cotangent of *r* radians
deg(*r*)	*r* radians as degrees (i.e. 180**r*/ π)
exp(*x*)	e^x, where e is the base of natural logarithms
fra(*x*)	the fractional part of *x* (i.e. *x* - int(*x*))
int(*x*)	the integral part of *x*, NB int(-2.1) = -3
it	the current iteration number (9.3.7)
ln(*x*)	$\log_e x$, i.e. the natural logarithm of *x*
log(*x*)	$\text{Log}_{10} x$, i.e. the common logarithm of *x*
max(*x*,*y*)	*x* if *x* > *y*, otherwise *y*

min(*x*,*y*)	*x* if *x* < *y*, otherwise *y*
opt(*c*,*x*,*y*)	*x* if *c* = 0, otherwise *y*
pi	3.1415926536
rad(*d*)	*d* degrees as radians (i.e. π**d*/180)
rnd	uniform random number between 0 and 1 (9.6.2)
sin(*r*)	the sine of *r* radians
sgn(*x*)	0 if *x* = 0; 1 if *x* > 0; -1 if *x* < 0
sqr(*x*)	the square root of *x*
tan(*r*)	the tangent of *r* radians

Figure 9.3 - Scientific Functions

9.3.2 Scientific Programs - General

Upper or lower case letters can be used interchangeably in scientific programs supplied to COMPUTE. However, in this document we shall use upper case exclusively for vectors and the E starting the exponent of a floating point constant.

A scientific program consists of one or more assignment or condition statements. Multiple statements must be separated from each other by semicolons or colons.

Spaces may be used to improve layout and clarity, but 10 or more in succession are considered to terminate the program so should be avoided. Spaces must not appear within a constant or a function name or between the initial letter of a dimension or vector reference and the subsequent quote or bracket.

9.3.3 Assignment Statements and Expressions

Assignment statements within scientific programs are of the general form:

variable = *expression*

where the *expression* combines variables, constants and operators in the conventional way familiar to users of BASIC or FORTRAN. The sub-expression within the lowest level of parentheses is evaluated first, the operators being processed in priority order as set out below:

functions (highest priority, see figure 9.3)
unary + -
^ or ** (alternate ways of coding power operator)
/ *
dyadic + -
relational operators (lowest priority, see section 9.3.4)

When two or more operators of the same priority are processed the calculation proceeds from left to right. In the interests of clarity curly and square brackets can be used interchangeably with parentheses. Redundant brackets are generally ignored except that the argument list of a multi argument function must be within exactly one level of parentheses, otherwise a syntax error will occur. For example:

min (x,y) **not** min ((x,y))

Constants are set up in the same way as the numbers between quotes in PIC FLT VALUE clauses (9.1.3). For example:

```
COMPUTE "a = 6.02252E23"
```

sets a to Avogadro's number, 6.02252×10^{23}.

9.3.4 Relational Operators and Comparisons

The relational operators are:

=	equality
<>	inequality
>	greater than
>=	greater or equal to
<	less than
<=	less than or equal to
==	approximate equality

Two expressions *x* and *y* are considered to be approximately equal and to satisfy the comparison *x* == *y* if |*x* - *y*| <= *x* *x* 10-10. Approximate equality is used to check whether two quantities are equal to within machine precision, bearing in mind the rounding errors that occur during the evaluation of functions, etc. Note, however, that since the underlying floating point implementation is decimal (rather than binary) simple additions, subtractions, multiplications and divisions will yield precise results when the number can be expressed in 11 digits. So you do not, for example, have to use approximate equality to guard against 1 + 5 not being equal to 6, a feature of some floating point systems.

The result of a relational operator is floating point 1 if the condition it defines is true, or floating point 0 otherwise. COMPUTE statements using such operators are generally needed to perform floating point comparisons, since the Global Cobol IF statement is limited to testing for equality (9.1.4). For instance, suppose in our quadratic equation solving example we required to detect the case when there were no real roots. This occurs when the discriminant of the equation (i.e. b^2 - 4ac) is negative. We might therefore code:

```
77 $ PIC FLT * Floating point work field
.........
.........
COMPUTE "d = b^2 - 4*a*c; $ = d>=0" $
IF $ = LOW-VALUES GO TO logic to handle no real roots etc., etc.
```

Note from this example that $ by itself is a valid Global Cobol data name which we suggest you use for the floating point work field holding the result of a COMPUTE comparison, and any other computed results of a temporary nature. Use of this convention makes the programming involved easier to follow.

9.3.5 Defining Vectors Using VEC$

So far all the scientific programs we have considered have performed calculations on single-valued **scalar** quantities. However, by using the VEC$ system routine you may temporarily replace any internal variable with a multi-valued **vector** defined by the calling program. The routine is invoked by a call statement of the form:

CALL VEC$ USING "V" *start dimension* [*displacement*]

Here V is the name of the variable involved. (Either an upper or a lower case letter may be used, but we recommend you conform to standard mathematical notation and use upper case.) The second parameter, *start*, identifies the beginning of the list of PIC FLT floating point numbers that make up the vector.

The third parameter, the *dimension*, is a PIC 9(4) COMP variable or integer literal indicating how many items are in the list.

The optional *displacement* parameter measures the distance in bytes between the start of each number that makes up the vector. If omitted, the vector will be assumed to be stored contiguously and the *displacement* used will be 6. By specifying a different *displacement* you can, for example define the columns or diagonal of a matrix as vectors (9.3.11).

Once an internal variable has been defined as a vector any statement referencing it is automatically iterated the number of times specified by its *dimension*. During the first iteration, references to the vector are to the first item in its list. Then after the statements involved have been executed once, iteration 2 begins, and all references are to the second item. Eventually the last iteration, defined by the dimension, completes and COMPUTE returns control. An example should make this clear:

Suppose the 10 coefficients of the polynomial:

$$a_9x^9 + a_8x^8 + \ldots a_0 = 0$$

are held in the Global Cobol table COEFF, with a_9 in the first entry, a_8 in the second, and so on. Then COEFF might be defined in working storage as:

```
77 COEFF OCCURS 10 PIC FLT
```

and the internal variable a could be redefined to represent the vector of these coefficients by:

```
CALL VEC$ USING "A" COEFF(1) 10
```

Given that the internal variable x already contains the current value of x, then the following COMPUTE statement returns the value of the polynomial in the PIC FLT number FPP:

```
COMPUTE "$=0: $ = A + $*x" FPP
```

The first statement of the scientific program simply sets $ (i.e. FPP) to zero. The colon separator isolates the vector, A, in the second statement so that statement alone is iterated 10 times to accumulate the required value by the well known method of nested multiplication.

Note the importance of the **colon separator** delimiting the scope of the iteration. Had we mistakenly coded:

```
COMPUTE "$=0; $ = A + $*x" FPP
```

both statements would have been included in the iteration which would have calculated:

.$=0; $=$a_9$ + 0*x; $=0; $=$a_8$ + 0*x; ... ;$=0; $=$a_0$ + 0*x

returning a_0 as the erroneous result.

An iteration may involve more than one vector, providing all concerned have the same dimension. For example, if *a* and *b* have both been defined as vectors of dimension *n* then:

```
COMPUTE "$=0: $ = $ + A*B" $
```

sets the PIC FLT field $ to the vector product of A and B. However, if vectors of different dimensions appear within the same group of statements bounded by colons this will be treated as a **syntax** error and the COMPUTE will be terminated with exception condition 1.

9.3.6 Resetting Variables Using RESET$

When you decide to use an internal variable to represent a vector the variable is no longer available as a scalar. (Recall that it is purely a convention that we code A to represent a vector but a for a scalar; scientific programs are case insensitive and it is only the previous call on VEC$ which has changed the meaning of the variable name as far as COMPUTE is concerned.)

To reuse variables as scalars once they are no longer required as vectors you pass a list of one or more of the variable names involved, terminated by a period, to the RESET$ routine:

```
CALL RESET$ USING "list."
```

For example:

```
CALL RESET$ USING "abx."
```

The letters appearing in the list may be upper or lower case. Here we use lower case to emphasize that the variables are being returned to scalar form.

Variables that have been reset are returned to the initial undefined state that all scalars assume prior to assignment.

9.3.7 The it Function

The it function, which takes no argument, is often employed in calculations involving vectors to obtain the current iteration number. (It always returns 1 when used in a statement not subject to iteration.)

The following example shows how the it function in conjunction with a conversion routine determines the PIC 9(2) COMP field ZI indexing the largest coefficient in the COEFF vector:

```
COMPUTE "n= 0;$= 0 :n= max(n,A);$= opt(n=A,it,$)" $
CALL FP-AC$ $
$STORE ZI
```

9.3.8 Dimensions

The 26 quantities a' ... z' (or A' ... Z') may be used as variables in expressions. Each refers to the dimension of the internal variable with which it is associated. The dimension of a vector is, of course, that established by the VEC$ call which defined it; that of a defined scalar is 0; and that of an undefined scalar is -1. Since a dimension reference is a scalar quality we shall use lower case letters in examples even though upper case are acceptable.

Suppose X has been defined as a vector. Then to calculate the mean of its entries code:

```
COMPUTE "$=0: $ = $ + X: $ = $/x"' $
```

A less obvious use of dimensions is to check that variables have been set up properly. For instance, before evaluating the formula for a quadratic the following test checks that the variables a, b and c are defined scalars:

```
COMPUTE "$ = (a'=0) + (b'=0) + (c'=0) - 3" $
IF $ NOT = LOW-VALUES logic to handle invalid variables
```

Note that when you code a dimension reference the ' character must immediately follow the variable letter; there must be no intervening spaces.

9.3.9 Vector Entries as Scalar Variables

Suppose VEC$ has been used to define A as a vector of dimension *n*. Then you may refer to the *n* different scalars that together make up the vector as a(1), a(2) ... a(*n*). More generally a vector entry reference takes the form:

v(*expression*)

where *v* is the name of a variable defined as a vector and *expression* is any expression. The reference may be coded as a scalar variable on either side of an assignment statement. When a scientific program containing such a reference is executed the expression is evaluated and its integral part (i.e. int(*expression*)) *i* say, calculated. Then providing *i* is between 1 and the vector *dimension*, the floating point number at location *start* + (*i*-1) * *displacement* is used as the variable, where *dimension*, *start* and *displacement* have been defined by the previous VEC$ call. The COMPUTE statement will be terminated with a **syntax** error if you code a reference using a variable which is not a vector. There will be a **dimension** error if the quantity i is not in the range 1 <= *i* <= *dimension.*

Assume A is the 10 entry COEFF vector introduced earlier. Then the quickest way of setting up the coefficients is to code them as PIC FLT numbers. However, they can also be established using COMPUTE:

```
COMPUTE "a(1)=.7; a(2) = 23.1E3; a(3) = 4.8; a(4) = -1"
etc. etc.
```

It is important always to remember that a vector entry is a scalar, not a vector; hence our use of lower case letters even though upper case would be acceptable. Suppose A and B are both vectors of dimension 10, and consider:

```
COMPUTE "A = B"
```

```
COMPUTE "A = b(it)"
COMPUTE "a(it) = b(it)"
```

The first of these involves 2 vectors, A and B, B being copied to A with 10 iterations. In the second statement there is only one vector A, but the it function selects the 10 entries of b one by one, so the effect is the same, even though the more complicated construct is slower and should be avoided. The third COMPUTE does not involve vectors at all, so there is only one iteration with the it function returning the value 1; the statement simply copies b(1) to a(1).

Note that when you code a vector entry reference the (character introducing the expression must immediately follow the variable letter); there must be no intervening spaces.

9.3.10 Condition Statements

There are two condition statements, the if statement and the for statement, of the general form:

if *expression*
for *expression*

A condition statement is always coded to the left of one or more subsequent statements and separated from the next one by a semicolon. When the scientific program containing it is executed the expression is evaluated. Providing the result is not 0 execution continues as though the condition statement were not present. When the condition is 0 evaluation of the statements to the right of the condition up to the next colon separator (or the end of the program) is skipped. Assuming the statements involved do not refer to vectors and are thus not subject to iteration, both condition statements perform identically and if should be coded in preference simply because it is shorter. When, however, the statements controlled by the condition are iterated:

if causes just the current iteration to be skipped;

for causes the iterating process to be terminated and the scientific program to continue with the statement following the next colon separator (or to end if there is no such statement).

The first COMPUTE statement which follows sets *n* to the total number of positive entries in the vector A. The second COMPUTE sets *s* to the number of initial positive entries in the vector, the summation process terminating as soon as a zero or negative entry is detected:

```
COMPUTE "n = 0 : if A>0; n = n + 1"
COMPUTE "s = 0 : for A>0; s = s + 1"
```

9.3.11 Vectors with Special Displacements

So far we have considered only vectors whose entries are stored in successive contiguous PIC FLT numbers (i.e. those for which the displacement is 6). The displacement can however be specified as any integer between -32768 and +32767 and there are important applications for some special values.

For example, earlier we introduced a vector A containing the coefficients of the polynomial $a_9x^9 + a_8x^8 + ... a_0 = 0$. The vector was set up by:

```
CALL VEC$ USING "A" COEFF(1) 10
```

implying, rather awkwardly, COEFF(1) = a_9, ... COEFF(10) = a_0.

To make the Global Cobol programming easier to follow without changing the COMPUTE statements in any way set up the vector 'backwards' using a displacement of -6:

```
CALL VEC$ USING "A" COEFF(10) 10 -6
```

Now we have COEFF(1) = a_0, ... COEFF(10) = a_9, arguably simpler.

Vectors with a displacement of 0 are particularly useful because, although they may have any dimension up to 32767, all the entries are clustered at the same point, a single PIC FLT number. They are in a sense not real vectors at all, so we shall refer to them as **pseudovectors**. They are employed in coding algorithms which require the iterative capability that ordinary vectors provide but which do not need the overhead of individual vector entry storage. To start with a very simple but interesting example, let ZN be a PIC 9(4) COMP field and ZFACT be PIC FLT. Then the following returns the factorial of ZN in ZFACT:

```
CALL VEC$ USING "F" ZFACT ZN 0
COMPUTE "f(1) = 1: F = it*F"
```

Next consider the Taylor series expansion of ln (1+x):

```
-{(1-x)/1 + (1-x)2/2 + (1-x)3/3 ...}
```

The following calculates this in ZLOG, stopping when the new term no longer contributes significantly to the sum. We make the dimension of the pseudovector 32767 (the maximum allowable) in this example, but in practice you would probably want to restrict the number of iterations to a much smaller limit.:

```
CALL VEC$ USING "S" ZLOG 32767 0
COMPUTE "s(1)=0; m=1-x; t=m"
COMPUTE "p=S; p=p-t/it; for 1-(p==S); S=p; t=m*t"
```

Note how a pseudovector is initialised by setting its first entry to 1. Had you, for example, mistakenly coded:

```
COMPUTE "S=0; m=1-x; t=m"
```

the 3 initialisation statements would have been iterated 32767 times! It is the for statement in the next COMPUTE that causes the iteration involving all 5 of its statements to end as soon as the series has converged rather than repeating the calculations 32767 times.

Finally, vectors with special displacements are useful in matrix arithmetic. If an *m* by *n* matrix is stored row by row, then each of its columns can be treated as an *m*-dimensional vector with its entries displaced 6*n* bytes from each other. For example, assume that MATA, MATB and MATC are, respectively, *m* by *n*, *n* by *p*, and *m* by *p* matrices defined in working storage by statements of the form:

```
77 MATA OCCURS mn PIC FLT
77 MATB OCCURS np PIC FLT
77 MATC OCCURS pm PIC FLT
```

and let ZM, ZN, ZP and Z6P be PIC 9(4) COMP fields containing m, n, p and 6p respectively, and ZI, ZK, ZROW, and ZOUT be PIC 9(4) COMP work fields. Then the following code forms the matrix product of MATA and MATB in MATC because the i,k th element of MATC is the vector product of the i'th row of MATA and the k'th column of MATB:

```
MOVE 0 TO ZI
MOVE 1 TO ZROW ZOUT
DO WHILE ZI < ZM
        ADD 1 TO ZI
        CALL VEC$ USING "A" MATA(ZROW) ZN
        ADD ZN TO ZROW
        MOVE 0 TO ZK
        DO WHILE ZK < ZP
                ADD 1 TO ZK
                CALL VEC$ USING "B" MATB(ZK) ZN Z6P
                COMPUTE "$=0: $ = A*B + $" MATC(ZOUT)
                ADD 1 TO ZOUT
        ENDDO
ENDDO
```

9.3.12 Performance Considerations

The floating point divide operation can take up to 4 times as long as a multiplication, so it is always quicker to multiply by a constant than divide by its reciprocal. Thus from the performance point of view:

```
COMPUTE "x= .5*[-b + sqr(b^2-4*a*c)]/a"
```

is better coding than:

```
COMPUTE "x= [-b + sqr(b^2-4*a*c)]/2/a"
```

The evaluation of a trig, log or exponent function typically involves 2 floating point divides and up to 8 additions and multiplications, so it pays to avoid unnecessary function calculations of this type. The most expensive algorithm of all is the one used for the power operator, but this has been specially optimised so that integral powers between 1 and 16 are calculated using repeated multiplication, so for example b^2 and b^3 are calculated just as though you had coded b*b or b*b*b. Note, however, that since negative integral powers are not optimised in this way:

```
COMPUTE "a= (1/x)^4"      is faster than  COMPUTE "a= x^-4"
```

9.4 Diagnostics

This section explains how errors in scientific programs are handled, and describes the debugging facilities available. Before reading it you should be familiar with the V6.1 and later $DEBUG system described in the Global Development Cobol User Manual.

9.4.1 The Diagnostic Report Scientific Extension

In general, if a program which has executed a COMPUTE statement fails, the diagnostic report (obtained by keying D to the debug prompt) contains additional lines to help you debug your scientific calculations, namely:

- The last scientific program executed;
- A line defining the error, if that program failed;
- The current value of the scientific variables.

Figure 9.4a shows the screen as it appears following an error in subroutine QUAD. The routine has attempted:

```
COMPUTE "x = (-b + sqr d)/2/a"
```

when a is 0. The error has been detected and COMPUTE has returned exception condition 2, **divide by 0**. However, there is no ON EXCEPTION statement following the COMPUTE so Global System Manager has terminated the program with EXIT 25402.

The 4 lines before last in the photograph show how the normal diagnostic report is followed by the scientific extension which pinpoints this error. The affected scientific program is listed, then the program error line identifying the failing statement and iteration (both 1 in this simple example) and the reason for the error. Then come the defined variables. Undefined ones are omitted from the list to save space. Scalar values appear in the standard numeric form (e.g. a, b, c and d). The information displayed for the vector, V, is its start address in hex (0560), its dimension (3) and the displacement (i.e. the number of bytes between entries - 6 in this case because the vector is stored in contiguous successive locations).

You may use $DEBUG's P instruction within the SCF window to obtain a colon prompt allowing you to key any scientific program. This enables you to inspect floating point numbers which are not listed, such as vector entries, or to modify existing values. The program you input is executed, the scientific extension is redisplayed, and a new colon prompt is output, the process being repeated until you key <CR>. In the example in Figure 9.4b the operator has first of all examined the vector entries, assigning them to 3 spare variables x, y and z using:

```
x = v(1); y = v(2); z = v(3)
```

Then the operator has changed a from 0 to the value 2.5. Finally, by keying <CR> to the colon prompt he or she has returned to the standard debugging system and, at the end of the screen, is loading symbolic names to examine other aspects of the problem.

[PHOTO HERE]

Figure 9.4a - A Divide by 0 Error

[PHOTO HERE]

Figure 9.4b - Debugging Following the Error

You should note that a scientific extension to the diagnostic report always appears once COMPUTE has been used. The error may have nothing to do with the last scientific program; in this case there will be no program error line and the variable list will immediately follow the line containing the successfully executed program.

9.4.2 Exception Conditions from COMPUTE

The example we have just followed is fairly typical, because generally you will not code an ON EXCEPTION statement following a COMPUTE, so any error will cause your program to crash in a similar way. COMPUTE can suffer 8 different types of error causing an exception to be returned with $$COND in the range 1 to 8. If you fail to trap the condition with an ON EXCEPTION statement immediately after the COMPUTE then Global System Manager will terminate your program with EXIT CODE 25401 to 25408.

The errors fall into 2 distinct categories; the **syntax** errors ($$COND =1) and the **execution** errors ($$COND = 2 to 8). They are therefore discussed in separate sections below.

9.4.3 Syntax Errors

COMPUTE returns exception condition 1 when a syntax error occurs. No attempt is made to execute the faulty scientific program, so the variable values remain unchanged. The error line appearing in the diagnostic report extension takes the form of a "^" character underlining the first part of the program found to be invalid. For example, all misspellings and misconstructions are detected in this way:

```
a = sqw b          (sqr not sqw)
    ^
x = 3x             ( 3*x, not 3x)
     ^
y = 10E99          (constant > 10^99)
       ^
z(3) = 1           (Z not defined as a vector)
^
a = v [3]          (space between v and [)
      ^
x = min (a)        (min takes 2 parameters)
          ^
x = min (a, b, a+b)    (min takes 2 parameters)
              ^
x = min ((a,b))    (argument list must be within single parentheses)
                ^
```

In addition, certain more subtle problems are detected at syntax checking time. If the $ character is flagged in error it means that no user variable was passed to the COMPUTE statement; you cannot refer to $ if you have not supplied a parameter along with the scientific program. If a vector variable is indicated another vector of different dimensions is involved in the same iteration. If you have corrupted the scientific variables area (9.6.2) affected variables will be flagged; the value of a corrupt variable appears as asterisks in the list.

9.4.4 Execution Errors

An execution error occurs when the syntax of a scientific program is correct but the program fails in some way when it is actually run. The program error line is of the form:

STATEMENT *ss* ITERATION *nnnnn reason*

where *ss* is the statement number (counting from 1) and *nnnnn* is the iteration number (also counting from 1, and always 1 for a statement not subject to iteration) at which the error occurred. In general the failing program will have modified the variables since all assignment statements and iterations executed before the one identified will have taken place; however, any assignment implied by the failing statement itself will not have been made.

The 7 different reasons for the error correspond to exception conditions 2 to 8:

DIVIDE BY 0 ($$COND = 2)

The program has attempted to divide by 0.

WRONG POWER ($$COND = 3)

The program has attempted either to take 0 to a negative power or to take a negative number to a non-integral power.

OVERFLOW IN *operator or function* ($$COND = 4)

A result greater in magnitude than $9.9999999999 \times 10^{99}$ has been obtained when processing the indicated operator or function. (The power operator, which may be coded as either ** or ^, always appears as ** in this message.) You should note that evaluation of the relational operators involves an internal subtraction so this error may occur if you try to compare a very large positive number with a negative number of very large magnitude (e.g. 5×10^{99} with -5×10^{99}).

ARGUMENT OF *function* ($$COND = 5)

The indicated function has been supplied with an argument which is out of range. The range for sqr, log, acs and act is determined by mathematical considerations. The trig functions are restricted to arguments of under 10^6 radians in the interests of accuracy. The exp argument range is limited as indicated to prevent overflow when the function is evaluated:

Function	*Valid argument range*
sqr *x*	*x* >= 0
log *x*	*x* > 0
exp *x*	*x* < 230.25850930
sin *x*	|*x*| < 1000000
cos *x*	|*x*| < 1000000 - pi/2 = 999998.4292
tan *x*	|*x*| < 1000000
cot *x*	|*x*| < 1000000
acs *x*	|*x*| > 1
asn *x*	|*x*| > 1

UNDEFINED *scalar* ($$COND = 6)

The indicated scalar variable has been referenced on the right hand side of an assignment statement, or in a condition statement, before it has been defined.

DIMENSION *vector* ($$COND = 7)

A vector entry reference of the form *vector(expression*) has been processed but the integral part of the expression, i say, has not been in the range 1 <= i <= *dimension*, where the *dimension* is that of the indicated vector.

IMPROPER *variable* ($$COND = 8)

The *variable* indicated contained a value which is too large to be a valid floating point variable. If it is a vector entry or the user variable ($) then the program has failed to initialise the vector or variable correctly, or it has become corrupt. If the *variable* is an internal variable, a - z, then probably you have corrupted the scientific variables area by misusing some of the advanced programming techniques (9.6.4).

9.4.5 Obtaining Diagnostics using DEBUG$

The DEBUG$ system routine can be used to provide the diagnostic information appearing in the scientific extension of the diagnostic report under the control of the program using COMPUTE. Thus instead of relying entirely on the debugging system to diagnose unexpected errors, you can call DEBUG$ as a subroutine to display part or all of the extension information at planned trace points. The routine is particularly useful for monitoring the progress of a numerical algorithm, or for displaying diagnostic information returned by a scientific subroutine (9.5.2). It is invoked with a CALL statement of the form:

CALL DEBUG$ USING *report-contents*

where *report-contents* is an integer literal or PIC 9(4) COMP field containing a 4 digit value, *abcd* say, used to specify which parts of the information are to be displayed:

a = 1 displays the last scientific program executed;
b = 1 displays the program error line (if any);
c = 1 displays the variable list;
d = 1 outputs the scientific program prompt.

When the prompt is included the user can input any scientific program and have it executed in the same way as under the debugging system; DEBUG$ will return control to its caller only when <CR> is keyed to the prompt. For instance:

```
CALL DEBUG$ USING 1111      * Full report and prompt
CALL DEBUG$ USING 1110      * Full report but no prompt
CALL DEBUG$ USING 1000      * Last scientific program only
```

9.4.6 Programming Notes

When a COMPUTE is executed Global System Manager establishes a system area pointer addressing the scientific variables, this pointer being reset whenever a ready prompt or main menu reappears at end of job. Symbolic debug checks to see that the pointer is resolved, and only then will the scientific extension be appended to the normal diagnostic report. Near to the variables is stored a special marker field containing the value #DADBDCDD. If this is not

present Global System Manager assumes the scientific variables have been overlaid, and suppresses the extension as if the pointer were missing.

If DEBUG$ is called before a COMPUTE statement has caused the pointer to be set up, or when the scientific variables are overlaid and the marker value is absent, it simply returns control after displaying the message:

NO SCIENTIFIC VARIABLES

9.5 Scientific Subroutines

This section describes the scientific subroutines provided as part of Global Cobol SCF. These are routines which perform commonly required numerical algorithms such as equation solving.

9.5.1 Parameter Passing Conventions

The numerical parameters required by a scientific subroutine are passed as the values of designated scientific variables, and the results of the calculations are returned in other specified variables. In general all variables not explicitly defined as parameters will be preserved as on input to the routine. This applies not only when the routine completes normally, but also if it fails, returning an exception.

Because numerical values are passed using the variables, scientific subroutines require few conventional parameters supplied via CALL...USING statements. However, when a routine makes use of an **ancillary function** provided by the calling program the paragraph or section name beginning the function calculation is supplied as a conventional parameter.

9.5.2 Error handling

A scientific subroutine returns exception condition 1 if it detects a **fatal** error, or exception condition 2 for a **recoverable** error. Fatal errors are normally those that indicate the parameters have been set up wrongly by the calling program; recoverable ones are usually due to failure of the numerical algorithm and thus are more likely to be handled in some way by the caller. In addition, any exception returned by an ancillary function is reflected to the calling program, so such a function should use EXIT WITH 3, EXIT WITH 4 etc. so that the resulting exception conditions, 3, 4 etc., can be distinguished from those used by the scientific subroutine itself.

When a scientific subroutine detects an error it restores the user variables apart from those used as parameters to their input state. Then it executes a sequence of the form:

```
COMPUTE "routinename - explanation"
ON EXCEPTION
END
EXIT WITH condition
```

The COMPUTE (which fails with a syntax error exception) makes the last scientific program processed simply a message identifying the failing routine and the reason for the error. This is so that the scientific extension to the diagnostic report contains meaningful information if the calling program fails to trap the exception resulting from the subsequent EXIT WITH statement.

9.5.3 Typical Calling Sequences

The type of calling sequence employed to invoke a scientific subroutine, in particular the way your program handles its errors, depends very much on the application itself. Probably most typical is the case where your program can deal with recoverable errors but is not expecting fatal ones (since these will be due to a problem within its own code).

In this case the calling sequence is of the form:

```
CALL scientific-subroutine
ON EXCEPTION
        IF $$COND = 1 STOP WITH code        * Debug handles fatal error
        Your own logic to handle recoverable errors
END
Processing when the routine completes normally
```

For the Global System Manager Calculator type of application the variables themselves are set up by previous operator interaction. A fatal error will be due to the user's mistake rather than a problem in the calling program. (Indeed, it is in order to support this type of working that exception condition 1, rather than immediate termination with a stop code, is employed for fatal errors.) In the calculator environment all the error handling normally necessary is to display an explanatory message, ideally the one provided by the failing routine itself. The operator can then take the appropriate action. The program simply calls DEBUG$ to output the message that the subroutine has established as the last scientific program executed:

```
CALL scientific-subroutine
ON EXCEPTION
CALL DEBUG$ USING 1000                      * Display error message
END
Prompt for next operator input, etc. etc.
```

9.5.4 Find a Root of f(x) = 0, ROOT$

Given *a* and *b* such that *a* < *b* and sgn [f(*a*)] = sgn [f(*b*)] this subroutine uses a mixture of interpolation and interval bisection to find x such that *a* <= *x* <= *b* and f(*x*) = 0.

The routine is invoked by a CALL statement of the form

```
CALL ROOT$ USING ancillary-function
```

where *ancillary-function* is the section name or paragraph name of code responsible for calculating f(*x*).

On entry to ROOT$ the calling program must have set scientific variables *a* and *b* to *a* and *b* respectively, so that they delimit the interval over which f(*x*) changes sign. On successful completion the routine will return the root *x* in *x*.

The ancillary function must compute f(*x*) in *f*, where *x* is supplied in *x*. When it is first entered all scientific variables apart from *x* will be as defined immediately prior to calling ROOT$. The function can use a to j for temporary working, since they are reset to their initial values whenever it is called. Variables *k* onwards can be used for results that need to be passed from one invocation to another, but if they are the values set up will be those returned to the caller when ROOT$ completes.

Exception condition 1 is returned if either *a* or *b* is not a defined scalar variable, or if *b* is not greater than *a*.

Exception condition 2 occurs if the interval *a*, *b* is not a sign change of the function (i.e. if sgn [f(*a*)] = sgn [f(*b*)]).

9.5.5 Solve Linear Equations A*X = B, SOLVE$

The SOLVE$ scientific subroutine solves the set of *n* linear equations:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$
.....
.....
.....
$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n$$

using the method of Gauss-Jordan elimination with partial pivoting. The calling program must have defined A, B and X as vectors of dimension n^2, n, and n, respectively. A must be set up to contain the left hand side coefficients stored contiguously in row major order (i.e. the vector is a table of consecutive PIC FLT numbers a_{11}, a_{12}, ... a_{nn}, as listed above). Similarly B must contain the right hand side, b_1 ... b_n, stored contiguously. The routine is invoked with a parameter-less call:

```
CALL SOLVE$
```

When it completes normally it will have calculated the solution vector X, so that $x(i) = x_i$ for i = 1 to n. The contents of A and B however are **destroyed** by the algorithm. The other scientific variables remain undisturbed.

Exception condition 1 will be returned if A, B and X are not defined as vectors of the appropriate dimension, or if either A or B is not stored contiguously (i.e. if the vector has a displacement other than 6). No values will have been changed in this case; all will be as on input to SOLVE$.

Exception condition 2 occurs when the equations are degenerate (e.g. if one of them is a repetition or a multiple of another) so that there are insufficient distinct equations for a solution. In this case A and B will be corrupt but X and other variables will be as on input to SOLVE$.

```
EXTERNAL SECTION SCVAR$
01      FILLER
*
*       26 slots for internal variables or vector attributes
*
  03    S-VARS                                  * 182 bytes, i.e. 26*7 in all
    05  FILLER      OCCURS 26                         * Variables a, b, c ... ,z
     07 S-TYPE      PIC S9 COMP                * -1 undef,0 scalar, 1 vector
     07 S-FP        PIC FLT                    * Fp value of scalar (type=0)
*
  03    FILLER              PIC X(5)
*
```

```
*       Seed for random number generator
*
  03    S-SEED      PIC FLT
*
*       Redefinition of the slots for variables A to Z as vectors
*
01      FILLER      REDEFINES S-VARS OCCURS 26
  03    FILLER            PIC 9 COMP              * Type always 1 for vector
  03    S-BASE      PIC PTR                 * Start address
  03    S-DIM       PIC 9(4) COMP           * Dimension
  03    S-DISP            PIC 9(4) COMP           * Displacement
*
*       The 26 internal variable slots individually named
*
01      FILLER      REDEFINES S-VARS
  03    S-A         PIC X(7)
  03    S-B         PIC X(7)
  03    S-C         PIC X(7)
.........
.........
.........
  03    S-X         PIC X(7)
  03    S-Y         PIC X(7)
  03    S-Z         PIC X(7)
```

Figure 9.6.2 - The Scientific Variables Area

9.6 Advanced Scientific Programming

This section describes how you can compute scientific programs defined at run time (rather than compile time); how you can set the random number generation seed to obtain a repeatable sequence of pseudo random numbers; and how you can create your own scientific subroutines conforming to Global Cobol standards.

9.6.1 Executing Scientific Programs Using COMP$

The COMP$ system routine allows you to execute a scientific program stored as a character variable; this enables you to prompt for a program and then execute it, or to execute a program saved in text form. Suppose program is defined as:

```
77      program     PIC X(n)
```

Then the calling sequence is of the form:

```
$SET program
CALL COMP$ [USING fp-number]
[ON EXCEPTION
        Logic to handle the exceptions described in 9.4
END]
```

The initial $SET statement (9.6.1) sets the source string register to identify the start address and length of the scientific program. A maximum of 78 bytes will be processed; if a longer program is supplied by mistake only the first 78 characters will be translated and executed.

Note that COMP$ is used to handle an ordinary COMPUTE statement; the compiler replaces a statement of the form:

COMPUTE "*scientific program*" [*fp-number*]

with code of the form:

$SET "*scientific program*"
CALL COMP$ [USING *fp-number*]

9.6.2 The Scientific Variables Area

The internal variables and random number seed used by COMPUTE can be accessed directly by compiling your program with copy library S.SCF and coding:

COPY S$

just before the linkage section (or the procedure division if a linkage section is not used). Figure 9.6.2 shows how the copy book defines an external section, SCVAR$, residing within the Global Cobol SCF routines included when the program is linkage edited.

The external section starts with the S-VARS area containing 26 7-byte slots defining the internal variables, arranged in alphabetical order. Thus, for example, S-TYPE(3) is the type field for variable c (or C). The S-TYPE field is the first byte of the slot and indicates how the variable is currently used:

-1 The variable is undefined;
0 The variable is a scalar;
+1 The variable is a vector.

Other values of S-TYPE are improper, and indicate that the area has been corrupted; an **improper** variable is listed as asterisks in the scientific extension of the diagnostic report. The 6 bytes that follow the type are either S-FP, a PIC FLT number containing the value of a scalar; or S-BASE, S-DIM AND S-DISP, the start address, dimension and displacement of a vector. These bytes are not used for an undefined variable.

The main use of the S-VARS area is for saving and restoring the caller's variables within scientific subroutines. To simplify the access to particular slots they are each individually defined as PIC X(7) fields named S-A, S-B etc.

In addition to the internal variables the SCVAR$ section contains the S-SEED floating point number used by the random number generator employed for the rnd function. Each invocation of rnd updates S-SEED, returning the new value as the function result, but remembering it in S-SEED for next time. Therefore, to set up a repeatable pseudo random sequence you need only initialise S-SEED to a specific value between 0.0100000000 and 0.9900000000. (This is similar to setting $$SEED to a particular value to determine the sequence used by the RAND$ system routine described in the Global Development Cobol Language Manual.)

```
PROGRAM TQ$A
*
* This routine finds the largest real root of ax^2 + bx + c = 0 if it
* exists; otherwise exception 2 is returned. The caller must supply
* scalars a, b and c. The routine returns x but leaves the other
* variables undisturbed.
*
DATA DIVISION
*
77    ZDSAVE      PIC X(7)                      * Save area for S-D
      VALUE LOW-VALUES                          * Initialises data division
77    $           PIC FLT                       * Computed fp value
COPY S$ SUPPRESS                                * Scientific variables
*
PROCEDURE DIVISION
ENTRY QUAD$
*
*     Check that a, b and c is each a defined scalar
*
      COMPUTE "$ = (a'=0) + (b'=0) + (c'=0) - 3" $
      IF $ NOT = LOW-VALUES
            COMPUTE "QUAD$ - a, b and c must be scalars"
            ON EXCEPTION                        * Ignore syntax error
            END
            EXIT WITH 1                         * Fatal parameter error
      END
*
      MOVE S-D TO ZDSAVE                        * Save user variables
      CALL RESET$ USING "dx."                   * Reset all working scalars
*
*     Check that d, the discriminant, is non negative
*
      COMPUTE "d = b^2 - 4*a*c; $ = d>=0" $
      IF $ = LOW-VALUES
            COMPUTE "QUAD$ - ax^2 + bx + c = 0 has no real roots"
            ON EXCEPTION                        * Ignore syntax error
            END
            MOVE ZDSAVE TO S-D                  * Restore user variable(s)
            EXIT WITH 2                         * Recoverable numerical error
      END
*
*     Calculate largest root, then exit
*
      COMPUTE "x = (-b + sqr d)/2/a"
      MOVE ZDSAVE TO S-D                        * Restore user variables
      EXIT
ENDPROG
```

Figure 9.6.3 - An Example Subroutine

9.6.3 Coding Scientific Subroutines

Figure 9.6.3 is the listing of the QUAD$ scientific subroutine we have seen failing earlier in the sections on diagnostics; the routine is **not** intended as an example of good numerical software; particularly since it terminates with an untrapped exception following its final COMPUTE statement if a is 0! It is included as a simple illustration of the coding steps necessary when creating a scientific subroutine conforming to the conventions described at the start of section 9.5.

On examining the data division you will see that the routine contains an area ZDSAVE in which those scientific variables which the routine uses internally, but which are not to be passed back to the caller as parameters, are saved. There is only one such variable here, d, but in a more complicated routine you will probably need to save and restore many more variables. In any

case you must compile the program with copy library S.SCF and include book S$ to define the SCVAR$ external section.

The procedure division begins with a COMPUTE statement which uses dimension references to validate that the parameters have been set up correctly (9.3.8). If the test fails QUAD$ executes:

```
COMPUTE "QUAD$ - a, b, c must be scalars"
```

to set up the error message of the form:

routinename - explanation

as the last scientific program executed (7.5.2).

The subsequent ON EXCEPTION and END statements ignore the syntax error returned by COMPUTE when this unacceptable program is translated. At this stage no variables have been altered so the routine simply terminates with exception condition 1 to return a fatal error.

Once parameter checking (which can usually take place without altering any values) is complete a scientific subroutine can proceed. Like QUAD$ it will need to save the variables which it requires for its own working. In our simple example this just involves moving S-D to ZDSAVE; a routine which needed to use many more variables might simply move the 182-byte S-VARS field to a save area in working storage.

After saving the variables the routine must use RESET$ to 'undefine' any of them that are required as scalars. This involves working variable d and the returned parameter x as far as QUAD$ is concerned. Note, however, the dire consequences of forgetting to reset a scalar. Suppose the caller of QUAD$ had established D as a vector and we had not reset it; then the COMPUTE statement which calculates d as the discriminant would overwrite every entry of the D vector rather than using the internal floating point scalar - i.e. S-FP(4) - for the value. Thus it is vital to reset the scalars a routine uses. There is no similar problem with variables that are to be used as vectors since the VEC$ call that defines a vector re-establishes all the information held for the variable in the S-VARS area.

The QUAD$ routine detects a recoverable numerical error when the discriminant of the equation is negative, indicating that there are no real roots. This is handled like the earlier fatal error returned when the parameters are invalid, except that condition 2 rather than condition 1 is returned and the user variables are restored before the EXIT takes place.

When the routine completes normally the results are returned in the specified parameters. For example, QUAD$ simply supplies the root in x; a more elaborate routine might use many more parameters. Then those variables which have been used for internal working which are not employed as parameters are restored to their initial values (only d is involved for QUAD$) and the routine exits returning normal completion.

9.6.4 Additional Uses of the Scientific Variables Area

If you take care you can manipulate the scientific variables directly to improve performance. For example, if x and y are known to be scalars:

```
MOVE S-X TO S-Y
```

has the same effect and is considerably faster than:

```
COMPUTE "y=x"
```

but do note how very careful you must be.

For example, if internal variable y is actually a vector the MOVE statement certainly does not have the same effect as:

```
COMPUTE "Y=x"
```

because, instead of setting each entry of Y to the scalar, the MOVE resets Y as a scalar with value x as if you had coded:

```
CALL RESET$ USING "y."
COMPUTE "y=x"
```

You can overwrite internal variable information to set up a vector directly. For example:

```
POINT S-BASE(1) AT COEFF(1)
MOVE 10 TO S-DIM(1)
MOVE 6 TO S-DISP(1)
```

has the same effect as:

```
CALL VEC$ USING "A" COEFF(1) 10
```

but bypasses the validity checks performed by VEC$ so is slightly faster but, as a consequence, dangerous and rarely to be recommended. It is, however, very useful to be able to find the start address of a user defined vector from within a subroutine by examining the appropriate S-BASE field, or by using the field to base a linkage section table defining the vector. This technique is employed by the SOLVE$ routine (9.5.5) to help it set up vectors for the current row, subdiagonal and so on within the A matrix; to make such coding easy SOLVE$ insists that the A matrix be stored contiguously.

In general only frequently used scientific subroutines written by experienced programmers will benefit from accessing the scientific variables area directly. For one-off calculations you are strongly recommended to employ only COMPUTE statements and VEC$ system routine calls so that your program remains easy to follow and debugging is assisted by the powerful validation checking provided.

9.7 Scientific Calculations using pre-6.1 Systems

If you wish to use the Global Cobol Scientific Calculation Facilities on a pre-6.1 system, then you must use the COMP$ interface as the COMPUTE verb is not supported.

You cannot specify floating point values in the Data Division under pre-6.1 systems, so these must be set up by using a call of COMP$. Define the data items as PIC X(6) variables.

Note that these restrictions only apply to **compiling** programs on a pre-6.1 system. Programs compiled using COMPUTE under V6.1 will execute correctly on all earlier versions of Global System Manager.

Appendix A - Included Routines

System routines referenced by the CALL statement are included in your program when it is linkage edited, as are access methods introduced by FD statements coded in working storage. Table A overleaf shows the program names of the particular subroutines included from the system library when various language constructs described in this manual are coded. The SIZE column indicates the approximate size of each routine in bytes, rounded up to the nearest 0.1K (K = 1024 bytes). System routines described in other manuals are excluded from this table as they are listed in the appendices of the appropriate manuals.

If you require a more accurate estimate you should compile and link a program containing a GLOBAL statement for each of the required routines and file organisations. The link map will then give the total size of the included routines.

A call on ACCE$ normally includes the QS$A routine (0.8K bytes) which is used by ACCEPT...LINE and DISPLAY...LINE. However, if ACCE$ is used only in scroll mode you can avoid including this extra routine, which is no longer required, by defining the global symbol Q$ACCE as a paragraph name within your program. Start the data division with the statement:

```
GLOBAL Q$ACCE
```

and label any one of the paragraphs in the procedure division as Q$ACCE.

Global Cobol statement	Program name of the subroutine included	Size (Kb)
CALL AC-FP$	TF$A	2.6
CALL AFP$	TF$A	2.6
CALL AS-EB$ CALL EB-AS$	IC$A	0.6
CALL AUTH$	QQ$A, EG$A, GH$A, EP$A, IG$A, GC$A, QL$A, EA$A	3.0
CALL BI-BS$ CALL BS-BI$	ID$A	0.4
CALL BI-OC$ CALL OC-BI$	IB$A	0.3
CALL CID-D$ CALL D-CID$	EP$A, IG$A	0.6
CALL CMND$	BC$A	0.1
CALL COMP$ COMPUTE	TF$A, TI$A, TT$A	10.1
CALL CUST$	BA$A	0.3
CALL DEBUG$	TD$A	0.9
CALL DFP$	TF$A	2.6
CALL DIVID$	OB$A	0.9
CALL DOWK$	CN$A	0.5
CALL DL-DT$ CALL DS-DT$ (paged)	GB$A	0.4

CALL DT-DL$ CALL DT-DS$ (paged)	GC$A	0.2
CALL DT-DY$ CALL DY-DT$	QK$A	0.6
CALL ENTRY$	IF$A	0.7
CALL EOJ$	QX$A	0.1
CALL EXIT$	QV$A	0.1
CALL FDAT$	BF$A, BZ$A	0.6
CALL FP-AC$	TF$A	2.6
CALL FP-SN$	TF$A	2.6
CALL FREE$	BN$A	0.5
CALL GETX$ CALL RELX$ CALL GETXN$	QR$A	0.6
CALL GROUP$	CM$A, AM$A, EC$A, ER$A, CA$A	11.1
CALL HMS-T$	CK$A	0.4
CALL LOAD$ CALL SDATA$	IE$A	2.4
CALL HX-BI$ CALL BI-HX$	QU$A	0.4
CALL IN-FP$	TF$A	2.6
CALL LOG$	EG$A, GH$A, EP$A, IG$A, GC$A, QL$A, EA$A	2.1
CALL LOGOF$	IV$A	0.1
CALL MIDN$ CALL MIDCH$	CT$A, QK$A, EA$A, CN$A	1.4
CALL MULTI$	OB$A	0.9
CALL NKM-C$	CW$A	0.3
CALL NLOGF$	IV$A	0.1
CALL OPID$ CALL USER$	BU$A, ER$A	2.3
CALL OPNM$	CO$A	0.3
CALL PRIN$	BP$A	0.5
CALL PROG$	CZ$A, EC$A	0.4
CALL PWCHK$ CALL PWNUL$ CALL PWNUM$	OZ$A	0.5
CALL QINDX$	EX$A	1.8
CALL QLOAD$	EY$A	0.8
CALL QSRT$	EI$A	0.8
CALL RAND$	CG$A	0.2
CALL RESET$	TF$A	2.7
CALL RESID$	IU$A	0.2
CALL RL-AS$ CALL AS-RL$	IA$A	0.7
CALL ROOT$	TR$A, TI$A, TF$A, TT$A	11.2
CALL SECS$	EA$A	0.1
CALL SOLVE$	TS$A, TF$A, TI$A, TT$A	11.2
CALL SQRT$	CE$A	0.3

CALL START$	EF$A	0.2
CALL T-HMS$ CALL TIME$	QL$A, EA$A	0.3
CALL TEXT$	IJ$A	0.3
CALL TSRT$	BT$A	0.1
CALL UNLO$	IF$A	0.7
CALL URESI$	IU$A	0.2
CALL VEC$	TF$A	2.6
CALL ZERO$	BZ$A	0.1

Table A - Included Routines

Appendix B - Memory Page Subroutines

All versions of Global System Manager from V6.2 onwards incorporate a memory paging system. This has been implemented to allow new features to be added to Global System Manager without reducing the memory bank size. Although the memory paging system is mainly used by internal Global System Manager routines, versions of a few of the most commonly used Cobol subroutines have been coded to run in memory pages. These are **Memory Page** subroutines which can be loaded as part of Global System Manager and used by application programs. The memory requirements of an application can be **significantly reduced** by using Memory Page subroutines. Global System Manager can be customised to load the Memory Page subroutines at bootstrap time. Although only one copy of each subroutine is loaded, they are available to all users on the computer. An application must be linked specially to use the Memory Page subroutines.

B.1 Routines available

Programs using the following Cobol functions can benefit from using Memory Page subroutines:

B.1.1 Global System Manager V6.2, V7.0 and V8.0

RSAM	(AR$B)
ISAM	(AI$A)
CLEAR statement and screen clearing subroutines	(GA$A)
Date conversion subroutines	(GB$A/GC$A)
CHECK$ subroutine	(GE$A)

B.1.2 Global System Manager V8.1

DMAM	(AM$A)
SLOCK$	(CA$A)
ASSIG$	(GH$A)
CHAR$	(GF$A)
COLOR$	(GG$A)

The names on the right are the program names of the Cobol subroutines that may have been linked into an application. If they appear on an application program's link map then a Memory Page subroutine can be used instead. The appendices of the appropriate development manuals explain which Cobol function uses which subroutine.

B.2 Linking the application to use Memory Page subroutines

A special subroutine library is installed as part of the development software. It contains small "stub routines" that are replacements for the routines mentioned above. They are merely interfaces to the Memory Page subroutines loaded by Global System Manager. The stub routines may also include data areas associated with the Memory Page subroutines. Include the following line into the dialogue for $LINK:

```
$44 LINK:C.$PAGES  UNIT:$S
```

The new routines will appear on the link map as a program name with the suffix 'Z' e.g. GA$A will be replaced by GA$Z. A program linked in this way will be **incompatible** with those earlier

versions of Global System Manager that do not include the Memory Page versions of the included subroutines (see table above).

B.3 Including the routines into V6.2 Global System Manager

The following technique must be used to include the Memory Page routines in V6.2 Global System Manager. **THIS TECHNIQUE MUST NOT BE ATTEMPTED ON ANY OTHER VERSION OF GLOBAL SYSTEM MANAGER**.

A library called P.$PAGES is distributed with V6.2 Global System Manager on one of the extension diskettes. This should be copied to unit $DP using $F. The $F 'PAM' instruction should then be used to set the number of pages to 13. The pages will then automatically be loaded by Global System Manager when next reloaded. The inclusion of the extra pages will increase the amount of memory required by Global System Manager by 12K bytes but will have no effect on the user memory bank size.

B.4 Including the routines into V7.0 and later Global System Manager

The installation of Global System Manager V7.0 and V8.0 always copies P.$PAGES to the SYSRES unit and sets the number of pages to the correct number. **DO NOT ATTEMPT TO USE THE $F PAM INSTRUCTION TO CHANGE THE NUMBER OF PAGES ON V7.0 OR V8.0 GLOBAL SYSTEM MANAGER.**

B.5 Errors

An attempt to use a Memory Page subroutine which has not been installed will result in a PGM CHECK 8, an illegal jump. This PGM CHECK will also occur if an attempt is made to use a Memory Page subroutine on a pre-V6.2 Global System Manager.