Global 16-bit Development System
Cobol User Manual
Version 8.1

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or
transmitted, in any form or by any means,
electrical, mechanical, photocopying,
recording or otherwise, without
the prior permission of
TIS Software Limited.

Copyright 1994 -2001 Global Software

MS-DOS is a registered trademark of Microsoft, Inc.

Windows NT is a registered trademark of Microsoft, Inc.

Unix is a registered trademark of AT & T.

C-ISAM is a registered trademark of Informix Software Inc.

D-ISAM is a registered trademark of Byte Designs Inc.

Btrieve is a registered trademark of Pervasive Technologies, Inc.

TABLE OF CONTENTS

Section Description Page Number

1. Introduction 5
1.1 Text File Creation and Maintenance 5
1.2 Compiling and Cross-Referencing 7
1.3 Linkage Editing 7
1.4 Compilation and Program Library Maintenance 7
1.5 Debugging Facilities 8
1.6 Print Files and Example Listings 8
1.7 Print Files and Example Listings 8

2. Text file editing using $EDIT 10
2.1 Overview 10
2.2 Editing Instructions 15
2.3 Error and Warning Messages during Editing 28

3. Inspecting, searching and listing 30
3.1 Processing Text and Print Files using $INSPECT 30
3.2 Searching for Strings using $SEARCH 36
3.3 Listing Files of any Format Using $L 38

4. Compiling and cross referencing programs 43
4.1 Compiling Programs using $COBOL 44
4.2 Cross-Referencing Programs Using $XREF 48

5. Library maintenance using $LIB 53
5.1 Introduction 53
5.2 Inspect and Extract Operations 59
5.3 Including and Comparing Members 62
5.4 Rename and Delete Operations 65
5.5 Space Saving Operations 67
5.6 Dispersed Program Library Support 69
5.7 Job Management Support 75

6. Linkage editing using $LINK 78
6.1 Linkage Editing an Independent Program 79
6.2 Linkage Editing Dependent Programs 90

7. Symbolic debugging using $DEBUG 97
7.1 Introduction 97
7.2 Establishing the Symbolic Debug Table for a Compilation 103
7.3 Setting the Program Base 105
7.4 The Diagnostic Report 108
7.5 Inspecting and Modifying Variables 109
7.6 Executing and Resuming Programs from Debug 113
7.7 Traps 119

8. Record and Playback Using $RCP 129
8.1 Recording Keystrokes 129
8.2 Playing Back a Script 131
8.3 Record or Playback Failure 133
8.4 Requirements for Running Record and Playback Software 133
8.5 Structure of the Script File 133

APPENDICES

Appendix Description Pa

A Program Preparation Example 138

B Linkage Editor Error Messages 143

C The Pre-V6.1 $DEBUG Facility 145

1. Introduction

This manual describes the Cobol commands used for preparing, testing and maintaining Global Cobol programs. The principal ones are: the text editor, $EDIT; the Global Cobol compiler, $COBOL; the linkage editor, $LINK; the librarian, $LIB; and the file inspection utility, $INSPECT. Figure 1 shows how they inter-relate. Rectangles represent the operation of the named commands, and the other boxes indicate files. The example file names used in the figure assume that the user system is being used to create or amend program SA013, and that the standard naming conventions recommended in Appendix E of the Global Cobol Language Manual are adopted.

Space prevents a number of other commands described in this manual from being shown in the figure. These are: the string search utility, $SEARCH; the program cross-reference utility, $XREF; the symbolic debugging system, $DEBUG; the record and playback handler, $RCP; and the file list and dump utility, $L. However, all are introduced and explained in the brief overview of the user system which follows.

1.1 Text File Creation and Maintenance

The text editor, $EDIT, uses a father/son updating technique to allow you to create or amend text files. For example, in the figure, S.SA013 might be set up initially by the operator keying its source at the terminal under control of $EDIT. If later a modification is required, $EDIT is used to produce an amended version of S.SA013. The command then renames the original file as B.SA013 so that it remains available as a backup version. The current text file and the previous backup must occupy the same unit.

A text file is held in an industry-standard form, consisting of a string of contiguous bytes interpreted as 8-bit ASCII characters. Appendix A of the Global Cobol Language Manual contains a table relating each possible byte value to its ASCII character equivalent.

A text file provides a particularly compact medium for storage of character information, such as program source and listings, since each line is terminated by a recognisable sequence such as form-feed or carriage-return line-feed and can therefore be stripped of unnecessary rightmost tabs and blanks. Furthermore, within the line, a single tab character may replace up to 8 contiguous blanks. Thus when the compilation listing produced by the Global Cobol compiler is output to direct access storage, rather than to a printer, it is in the format of a text file in order to be as compact as possible.

The $INSPECT command allows you to examine text or print files at the terminal. It can also be used to list selected portions of such files. Furthermore, by "listing" to a direct access device rather than an actual printer you are able to combine two or more fragments from a number of text files to form a new one. You can also use $INSPECT to convert text files to relative sequential print file format and vice versa.

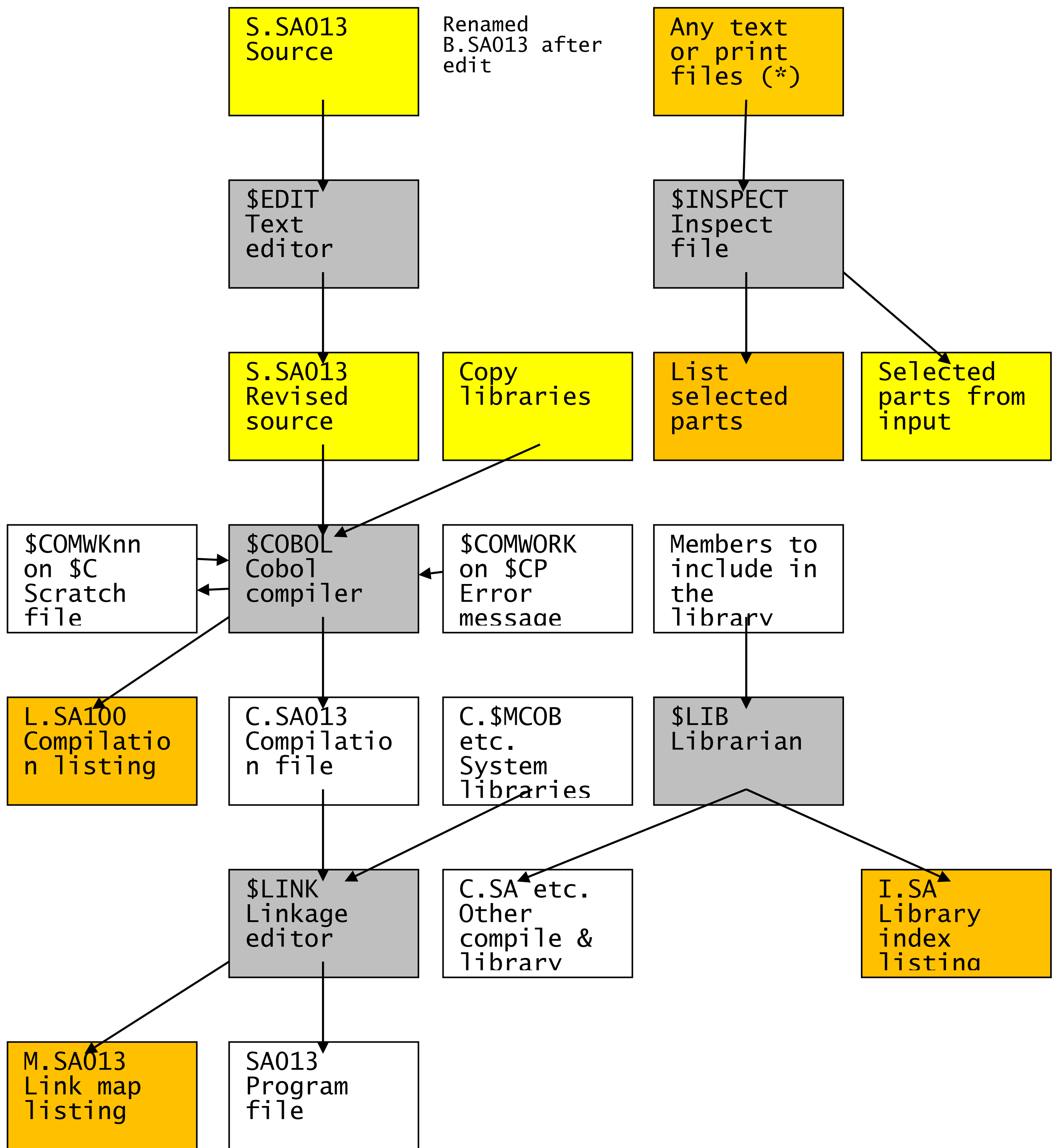


Figure 1 - Principal Commands of the Global Cobol User System

The string search utility, $SEARCH, enables you to examine any number of text files to determine all occurrences of up to 10 different strings. $SEARCH is particularly useful when enhancing or documenting large systems involving many programs.

1.2 Compiling and Cross-Referencing

The compiler, $COBOL, processes a source program, together with up to three copy libraries, to produce a compilation file and an optional compilation listing. The compiler uses a read-only error message file, $COMWORK, which resides on the unit occupied by $COBOL itself. During the course of any compilation a temporary scratch file, named $COMWKnn (where nn is a unique number on the system in question), is developed on the unit assigned to $C. This file will be deleted when $COBOL completes normally.

The source program and the optional copy libraries are all text files, and as such can be maintained by $EDIT, and inspected or searched by $INSPECT or $SEARCH. The rules for creating valid source programs and copy libraries are, of course, described in the Global Cobol Language Manual.

The compilation file output by $COBOL may contain unresolved global references to access methods and system routines. It is not itself an executable program, and must be processed by the linkage editor in order to become so.

When, towards the end of development, you wish to cross-reference a program, you supply its source and any supporting copy libraries to $XREF. This uses the error message and scratch files in the same way as $COBOL, but outputs a cross-reference listing instead of the compilation file and listing.

1.3 Linkage Editing

The linkage editor, $LINK, is used to combine one or more compilation files with subroutines or maps from the system libraries, C.$MCOB and C.$APF and optional user libraries in order to form an executable program file. A link map listing describing the program so created in terms of its constituent modules may be output at the same time. The program file produced by $LINK may either contain an independent stand-alone program, or the root, or a dependent program, of an overlay structure.

Although the system libraries are shown as optional in Figure 1, In practice they are required by all but the most trivial programs. They contain not only system routines explicitly invoked by CALL statements, but also access methods and other routines implicitly called to support Global Cobol statements such as EDIT and SORT.

The system libraries must occupy the unit assigned to $S along with the $$LINK file which controls library selection.

User libraries are optional. When present, they fulfil a role similar to that of the system library, except that they contain common modules coded by the user rather than supplied as part of Global. In any case the linker is able to create a program from up to 25 different input files, each of which may be either a compilation library or an individual compilation file.

1.4 Compilation and Program Library Maintenance

The librarian, $LIB, is responsible for maintaining user compilation and program libraries. Compilation libraries contain your own subroutines or screen formats (created by the Global Screen Presentation's $FORM command), and are used when linking a program,

just like the system library. Program libraries, on the other hand, are employed at run-time, since a program may equally well be executed as a stand-alone file or as a member of the currently attached program library.

The librarian can be used to create or update library files containing up to 100 different members. It can also display or print an index which lists the contents of any particular library. Libraries are updated in place and may be allocated spare space to allow for expansion.

$LIB is not responsible for processing copy libraries, which are conventional text files and are therefore updated using $EDIT.

1.5 Debugging Facilities

The symbolic debugging system, $DEBUG, consists of a collection of overlayed commands which execute in a separate Global partition.

The symbolic debugging system may be invoked following a program check, stop or exit condition, trap, or break request. It is also possible to execute programs from $DEBUG and set traps to cause the system to be re-entered once a particular statement is reached, or a certain overlay is loaded.

Providing the program files involved remain online during $DEBUG's execution, symbolic information is available which allows you to inspect or modify a named data item, or set traps on the first instruction of named sections or paragraphs. You can also work with absolute addresses when symbolic information is not available.

The file list and dump utility, $L, is also a valuable debugging aid since it allows you to display or print files of any type in hexadecimal, with an ASCII conversion where appropriate.

1.6 System Testing Facilities

The record and playback facility using $RCP and <SYSREQ> Q enables you to record keystrokes typed on a keyboard and to subsequently play them back to repeat the original processing. The record script is held in a text file and can be run several times allowing you to repeat program testing dialogue when program amendments have been made. The scripts can also be chained together to enable the testing of large systems so that you can apply the whole of the entire script run or only part.

1.7 Print Files and Example Listings

The files represented by a printer symbol in Figure 1 may either be written to a real printer or to direct access storage, according to information you supply at run-time. You can use $INSPECT to examine a direct access print file at your terminal, or $PRINT (described in the Global Utilities Manual) to output it either to a real printer or a spooling volume.

A sample source program, S.SAMPLE, is provided as part of the Global Cobol Development System. Appendix A of this manual contains the following example listings:

- Source listing, S.SAMPLE, output by $PRINT;

- Compilation listing, L.SAMPLE, output by $COBOL;
- Link Map listing, M.SAMPLE, output by $LINK;
- Cross Reference listing, X.SAMPLE, output by $XREF;
- String Search listing, D.$SRCH, output by $SEARCH.

The appendix explains how you can print, compile, link, cross reference and search S.SAMPLE yourself in order to produce copies of these listings on your own computer. It is recommended that you work through this exercise step by step before using the development system for production work.

2. Text File Editing Using $EDIT

2.1 Overview

The $EDIT command described in this chapter is used either to create a new text file or to modify an existing one. Typically $EDIT is employed to amend programs and copy libraries or to create short programs. The command assumes you are using a display terminal with an adequate line width (usually 79 or 80 characters). The maximum size of line that can be edited is 72 characters, to conform to Global Cobol standards. If $EDIT finds a longer line it will simply truncate it.

$EDIT can only be used to create or modify a single file at a time. Editing is a father/son updating process with both the old input file (if any) and the new output file residing on the same direct access volume. Whenever a "son" is produced by amending a "father", and the edit is successful, the son is given the file-id of the father file, whilst the father has the prefix of its file-id changed to B. so that it remains available as a backup version.

Any number of files may be edited successively during a single execution of $EDIT, and text may be passed from one to another using a hold buffer which $EDIT maintains internally. Thus selected portions of text may be concatenated from a number of files (for example, to merge two or more copy libraries into one).

This chapter is divided into 3 sections. This initial overview describes how to run the $EDIT command and enter its **amendment phase** in which a number of editing instructions are provided to enable you to change or create text. Following this overview, section 2.2 describes the editing instructions in alphabetical order. (Table 2.2 provides a summary of these instructions which you will be able to use as soon as you are familiar with the basic strategy.) Section 2.3, which concludes the chapter, describes the error and warning messages which may appear during the amendment phase.

When you run $EDIT it displays the edit file prompt, requesting the name and unit-id of the file you require to create or update. Unless you explicitly supply a prefix when you key the file-id, $EDIT will assume you are working with a source file and append the S. prefix by default.

For example:

```
$42 EDIT:SAMPLE UNIT:205
```

indicates that the operator wishes to create or modify source file S.SAMPLE on unit 205.

Note that you are not allowed to edit a file with a B. prefix because this prefix is reserved by $EDIT for backup versions of edited files.

2.1.1 To Quit

To quit and return control to the monitor simply key <ESCAPE> in response to the edit file prompt.

Figure 2.1 - Specifying a new file

Figure 2.2 - The empty input file

2.1.2 Creating a New File

If the file you have specified is not present on the unit the editor displays an introductory screen similar to that shown in figure 2.1.

This screen describes the conditions prevailing at the start of the creation of a source file. The size of the data area allocated for the new file is displayed, and you should check that this is adequate to hold the file that you are about to create. The size of $EDIT's internal line buffer, which depends on the size of the Global Cobol user area, is given, as well as the number of lines of this buffer that are currently being used as a hold buffer. If you are not satisfied with this information, you should key N as requested to return to the edit file prompt. Otherwise key <CR> (or any single character apart from N) and the editor enters its amendment phase and displays the screen shown in figure 2.2.

This shows that the input file is empty (as, of course, it will be). The cursor is positioned at the left of the base line to allow you to key an **editing instruction.** You would typically key I now to insert lines of text.

Once you have keyed I you are prompted for the first line of the file by a **text prompt** on the base line. This prompt, which appears as a colon in column 8, is repeated over and over again so that you can supply every line of the file. Figure 2.3 shows the situation when the operator is part of the way through keying the ninth line of a new program.

2.1.3 Updating an Existing File

When the file you specified in response to the edit prompt is present on its volume, $EDIT displays an introductory screen similar to the one below, describing the conditions prevailing at the start of an update to an existing source file as shown in figure 2.4.

The size of the existing old file is displayed, together with the size of the data area allocated for the new file, and the difference which is the amount by which the file may be expanded by this edit. If there is no capacity for expansion then a warning message is displayed at the top of the screen. The size of $EDIT's internal line buffer, which depends on the size of the user area, is given, as well as the number of lines of this buffer that are currently being used as hold buffer. If you are not satisfied with this information, you should key N as requested to return to the edit file prompt. Normally you should key <CR> (or any single character apart from N) to proceed to the amendment phase, but if a warning message appears at the top of the screen then you must key Y if you wish to proceed in spite of the warning. The first page of the file will then be displayed as displayed in figure 2.5.

Figure 2.3 - Keying in a program

Figure 2.4 - Amending an existing file

The cursor is positioned at the left of the base line to allow you to key any appropriate editing instruction. The base line indicates that

the top line of the screen (i.e. the first line of the file) is the current line.

2.1.4 The Amendment Phase

Once the amendment phase has been entered you can create a new file or modify an existing one. The screen remains partitioned into two areas, as shown in figures 2.3 and 2.4. The last line, the **base line**, is used to accept instructions and text, and to display amended lines and error messages. The remainder of the screen, termed the **current page**, is used as a 'window' into the file to show the part that is currently being edited. Except when a sequence of insertions is taking place, the lines that make up the current page are formatted as shown in the following table:

COLUMN	CONTENTS
1 - 4	Blank except for the top line of the screen, which contains a sequence number from the input file if that line number still applies to the current state of the edited file.
5	Blank
6,7	The line number within the current page. The top line is number 1, the second number 2, and so on. If you delete a line and your terminal has cursor control capability, then the entire display line, including the line number, will be blanked out.
8	Blank
9 onwards	The text to be stored on the output file.

Table 2.1.4 - Structure of Lines of the Current Page

At any moment the only text lines that may be changed by editing instructions are those that appear on the current page. However, the current page may be moved forward or backward within the file to make other lines available for modification. Once the file has been modified during an edit there will be a limit beyond which the current page may not be moved backward and if you should encounter this limit you will have to begin a new edit to update the output file itself. (There is a special instruction, B, which allows you to start editing again from the beginning of the output file without having to quit the amendment phase and return to the edit file prompt.).

Figure 2.5 - First page of an existing file

Figure 2.6 - End current file

If you are working at a terminal with cursor control capability, then whenever you amend or delete a line the current page is updated to show the result of your modification. If there is no suitable cursor control feature available then the current page will not be updated. However, in this case the amendment is displayed on the base line to confirm the change, and the current page can be refreshed at any time by keying <CTRL A> in response to either the instruction or the text prompt.

Note that you may also key <CTRL A> (or <ESCAPE>, which is treated identically) if at any time during the amendment phase you "lose your

place" in your dialogue with the editor. This will simply refresh the current page without causing any file updates. In this way you will be able to see exactly which modifications have been effected and still be able to reverse any unwanted change. <ESCAPE> is deliberately treated specially, to prevent you from losing an edit by hitting the key by mistake.

2.1.5 Quitting the Amendment Phase

Two special editing instructions cause $EDIT to quit the amendment phase and redisplay the edit file prompt so that you can either process a new file or return control to the monitor.

The **E (End Current Edit)** instruction is used when you have satisfactorily completed an edit. It causes the remainder of the input file (if any) to be copied to the output file, then the input file (if present) to be renamed with prefix B. as the new backup file, any previous backup file being deleted. The output file is then given the edit file-id. For example, suppose you had keyed SAMPLE in response to the edit file prompt, in order to modify source file S.SAMPLE. Then, as a result of the E instruction:

- If a file named B.SAMPLE already exists on the volume containing S.SAMPLE it is deleted;
- Then the existing S.SAMPLE is renamed B.SAMPLE;
- Next, the new file containing the result of the edit operation is itself named S.SAMPLE;
- Finally, the edit file prompt is redisplayed.

The screen shown in figure 2.6 shows E being used to complete one edit and start another.

The **A (Abandon Current Edit)** instruction can be used if you have made a serious mistake in your editing and wish to leave the file you are working with in its original state. Following your keying of A the editor responds with a second prompt:

```
ARE YOU SURE?:
```

If you key Y to this prompt the abandon instruction will be honoured and the edit file prompt will be redisplayed. A reply of N, <CR> (or any single character apart from Y) will cause the "A" to be ignored and a new instruction prompt to appear.

2.1.6 Edit Jobs

An edit job consists of a fixed sequence of editing instructions which is defined by the J (Job) instruction. The job can then be invoked as often as required at any time during the same execution of $EDIT, using the U (Use Job) or Q (Quiet Job) instruction. A typical use of an edit job might be to replace all occurrences of a given text string within a file by another string.

2.1.7 Operating Notes

If the output file is not large enough the editor will be terminated as soon as it is full with the error message:

```
$42 EDIT OUTPUT FILE EXHAUSTED
```

In this case control will be returned to the monitor and unless you take special pains to recover the editor work file, as explained below, you will have lost the result of your editing session. You should therefore be particularly careful to check the information on the introductory screen to ensure there is enough file space available for your purposes. If there is not you should key N as requested to return to the edit file prompt, <ESCAPE> to quit $EDIT, then use the file utility to delete any unwanted files and possibly condense the volume. For example:

```
GSM READY:$EDIT
$42 EDIT:SAMPLE UNIT:205
.........
......... (introductory screen)
.........
$42 KEY N IF NOT SATISFACTORY:N
$42 EDIT:<ESCAPE>
GSM READY:$F
$66 INPUT DEVICE:205
$66 OUTPUT DEVICE:<CR>
$66 FILE MAINTENANCE
:LIS
......... (directory listing which you examine to see how
......... much spare space is available, which files
......... can be deleted, and so on)
$66 FILE MAINTENANCE
:
(here you are able to DELete files, or CONdense the volume)
```

If the "output file exhausted message" interrupts a long editing session, you may wish to recover the results of your labours from the editor work file, S.$EDInn, where nn is 01, 02, ... etc., your user number, which is 01 on a single-user system. It occupies the same volume as the source file, and contains that part of the file which has been correctly updated. You may use the $INSPECT command to merge this with the remainder of the original source file, so that you can continue working from where you left off. See the example in 3.1.14, at the end of the description of $INSPECT.

EDITING INSTRUCTION	**DESCRIPTION**
A	Abandon current edit.
B	Begin again at the start of the new edit file.
C[line]	Comment part only of line is replaced.
D[line]	Delete line.
E	End current edit.
F[line]	Find a line containing a given string.
H[line]	Hold a line in the hold buffer.
I[line]	Insert lines in front of the specified line.
J	Job definition, i.e. define an edit job.
K[line]	Kill, i.e. delete lines to end of screen.
L	Line with sequence number supplied to text prompt becomes top of the current page.
M[line]	Move a line to the hold buffer.
N	Repeat last instruction until end of screen (D, H, M,

	T, W, X or Y only).
O[line]	Output the contents of the hold buffer before the specified line.
P or -P	Page forward (P) or backward (-P).
Q[line]	Quiet job, i.e. run an edit job without display.
R[line]	Replace line.
S[line]	Statement part only of line is replaced.
T[line]	Tabulate statement and comment.
U[line]	Use job, i.e. run an edit job with display.
V[line]	Position specified line at top of current page.
W[line]	Set the current line from the specified line.
X[line]	Exchange the first occurrence of field-a on the line for field-b. You supply the string: field-a/field-b in response to the text prompt.
Y[line]	Exchange the first occurrence of field-a on the line for field-b. You supply the string: <delimiter>field-a<delimiter>field-b in response to the text prompt.
Z	Zeroise (i.e. clear) the hold buffer.
$pp	Change editor tabulation, as follows: pp=SC Statement column (default starting at 9) pp=CC Comment column (default starting at 41).
$*=x	Change comment start character to x
nnnn +nnn -nnn	Go forward (nnnn or +nnn) or backward (-nnn) the specified number of lines and display a new page.
<CR>	Repeat the previous instruction (D, F, H, M, P, Q, T, U, W, X, Y, -P, nnnn, +nnn or -nnn only).
!	Terminate iteration in Edit Job if exchange fails
?	Terminate iteration in Edit Job if exchange succeeds

Table 2.2 - Summary of Editing Instructions

2.2 Editing Instructions

Table 2.2 summarises the editing instructions which may be keyed in response to the **instruction prompt**, a colon appearing in character position 1 of the base line. An editing instruction typically consists of a letter followed by optional line information represented by [line], where [line] may be:

- an integer, corresponding to the number of a line within the current page, typically between 1 and 23 inclusive;
- omitted, in which case the current line number is assumed;
- N, in which case the current line number plus one is used, any deleted lines being skipped over. If, however, the current line is the last line of the current page this option is invalid, an error message will be displayed, and the instruction ignored. Instructions can only affect lines on the current page.

During its amendment phase $EDIT usually displays the **current line number** in columns 6 and 7 of the base line, so that it appears immediately below the numbers used to identify individual lines of the current page.

When you key certain instructions $EDIT requests additional information, such as the contents of a new line, by means of a text prompt, a colon appearing in character position 8 of the base line. For example, to replace line 12 with new information you key R12 in response to the instruction prompt, and the new information to the subsequent text prompt:

```
R12    12:THIS NEW INFORMATION REPLACES LINE 12
```

If you make an error in keying an instruction, one of the warning messages described in section 2.3 will appear in the area usually reserved for your reply to the text prompt. The message will be blanked out as soon as you key a valid instruction to the instruction prompt. For example, suppose you key:

```
:$     12
```

on the base line (which shows that line 12 is current). This is not a valid editor instruction, so $EDIT displays:

```
:      12 EDITOR INSTRUCTION NOT RECOGNISED
```

If you now key R, the warning message will be blanked out so you can supply information for a replacement line in response to the text prompt:

```
R      12:
```

In the subsections which follow we describe the editing instructions in the same order as Table 2.2, but in more detail.

When editing, updates are reflected on the screen. If you are using a non-cursor control VDU (such as during system installation, for example) this will not occur.

2.2.1 A - Abandon Current Edit

To abandon editing a file, leaving the original source file (if any) unchanged, key the instruction:

```
A
```

The editor responds by asking "ARE YOU SURE?" in case you keyed A by mistake. You must key Y to complete the abandon operation and obtain the edit file prompt so that you can process another file or return control to the monitor. For example:

```
:A ARE YOU SURE?:Y
$42 EDIT:
```

If you reply <CR>, N (or any single character apart from Y) the A instruction you have miskeyed will be ignored, and you will be able to continue the edit.

2.2.2 B - Begin Again at the Start of the New Edit File

To cause the new file you have created to be saved, along with the remaining input, as the source file, and the previous source file to be made the backup file, key the instruction:

```
B
```

The introductory screen will then be displayed so that you can update the file again from the beginning.

The effect of the B instruction if you were, say, editing S.SALES on unit 205, is identical to the longer sequence:

```
:E    (end edit)
$42 EDIT:SALES UNIT:205
```

It is simply more convenient. During a complex edit you might use the B instruction several times. You may wish to pass the file once for each different type of change you are making, or you may simply want to take a periodic checkpoint of any amendments you have made.

2.2.3 C - Comment Replacement

To replace the comment part of a line, leaving the statement part unchanged, key the instruction:

```
C[line]
```

For example:

```
:C12 or :CN or :C
```

Once you have keyed the instruction you will be prompted with the current comment as a field editable default. If you do not begin your comment with an asterisk, then one will be automatically supplied by the editor. If you key spaces the comment part of the line is **deleted.**

The comment part of a line starts with the first asterisk which Is not enclosed within a pair of " characters. The new comment keyed replaces any existing comment in its entirety, and starts in the same column. If there is no existing comment the new comment begins in the comment starting column (explained in 2.2.25) unless the statement part extends into this column, in which case the new comment is positioned so as to leave one space between it and the statement part.

Following the instruction the line modified becomes the current line. The current page will be altered to reflect any change.

2.2.4 D - Delete a Line

To delete a specified line from the current page you key the instruction:

```
D[line]
```

For example:

```
:D12 or :DN or :D
```

Following the instruction the deleted line becomes the current line. However it is not possible to process a deleted line, and if an attempt is made to do so $EDIT will skip to process the next line on the current page, but will not advance past the end of the current page.

The current page will be modified to reflect any change.

If you are not, and you wish to check the effect of any amendments, simply key <CTRL A>.

2.2.5 E - End Current Edit

To finish editing a file, and copy any remaining input unchanged to the new file, key the instruction:

```
E
```

This causes the existing source file, if any, to become the backup file, and the new file to become the current source file. The edit file prompt will be output next so that you can process another file or obtain the monitor's ready prompt. For example:

```
:E
$42 EDIT:
```

2.2.6 F - Find a Line

To find a line containing a given string, and display it at the top of the screen as the first line of the current page, key the instruction:

```
F[line]
```

For example:

```
:F12 or :FN or :F
```

You will then be prompted, by a text prompt, for the characters to be searched for. The search commences at the specified line of the current page and continues until the string is found or the end of the input file is reached. For example, to "find" the linkage section:

```
:F    :LINK
```

When a line containing the characters you specified is found a new page is displayed with that line at the top of the screen. It is the new current line. If the line found is not the one you actually require you can repeat the same search by simply keying <CR> in response to the instruction prompt. You are offered the last F string as a field editable default.

You must take care when searching for a string containing spaces, since the number of spaces actually appearing is significant. There is no convention such as multiple spaces counting as one.

2.2.7 H - Hold a Line

To copy a specified line to the end of the hold buffer, leaving the original line unchanged, key the instruction:

```
H[line]
```

For example:

```
:H12 or :HN or :H
```

Following the instruction the line held becomes the current line.

Once a line has been added to the hold buffer it remains there until either the Z instruction is keyed or control is returned to the monitor. Thus text can be moved between files via the hold buffer.

2.2.8 I - Insert One or More Lines

To insert one or more lines you key the instruction:

```
I[line]
```

For example:

```
:I12 or :IN or :I
```

where [line] gives the number of the line in **front** of which the new line or lines are to be inserted. If the 'end of file' line is specified then any new lines will be added to the end of the file.

Once you have keyed the instruction, the current page will be redisplayed, showing only those lines that precede the proposed insertion. Immediately below these, on the base line, the text prompt will be output as a colon in position 8. You should then key the first new line to be inserted.

As soon as you hit <CR> to signal that the line is complete, the display will scroll, to show your insertion in its proper place, with the character I in column 2 of each line of the current insertion. A new text prompt will be output to allow you to key a second or subsequent line should you so wish. You must key <CR> to indicate that you have finished inserting lines. (If you key <CR> to the first text prompt the I instruction is simply ignored.)

When you terminate the insertion by keying <CR> the screen is refreshed to show the new addition and the instruction prompt is redisplayed. When the screen is refreshed an attempt is made to display all the inserted lines, subject to the restriction that the following line, which becomes the current line, is always displayed.

2.2.9 J - Job Definition

To define an edit job for subsequent execution, key the instruction:

```
J
```

The screen will be cleared except for the title EDIT JOB DEFINITION which appears above an instruction prompt on the base line, indicating that $EDIT is ready to accept your edit job definition. You supply the edit job definition by replying to instruction prompts and text prompts exactly as if you were keying instructions for immediate execution. As each complete editing instruction is entered, the display is scrolled so that, within the limits of the screen size, the job definition is displayed in full to allow you to check it. To terminate the job definition, making it available for execution by a Q or U instruction, key <CTRL C> to an instruction prompt. The job definition remains available for execution until either a subsequent edit job is defined or control is returned to the monitor.

If you make a syntax error in your job definition, your input will either be ignored or rejected with an appropriate error message. If

you realise that you have made a mistake then you must key <CTRL A> or <ESCAPE> to abandon the job definition and return to the instruction prompt so that you can rekey the J instruction.

The following edit job finds the first occurrence of the string FRED on or after the current line, replaces it by the string JIM and updates the current line to become the line edited. When executed repetitively by a Q or U instruction it can be used to replace all occurrences of FRED by JIM within an entire file:

```
      EDIT JOB DEFINITION
:F    :FRED
:X    :FRED/JIM
:<CTRL C>
```

Note that the V instruction can be particularly useful within an edit job as it positions the current line at the top of the screen and thus enables any line within the new screen to be addressed directly. The following edit job, when executed repetitively, will insert a copy of the hold buffer at intervals of ten lines within the input file, and is independent of the number of lines contained in the hold buffer:

```
      EDIT JOB DEFINITION
:O11
:V
:<CTRL C>
```

2.2.10 K - Kill (Delete to the End of the Page)

To delete all lines on the current page, from a specified line number onwards, key the instruction:

```
K[line]
```

For example:

```
:K12 or :KN or :K
```

The next page is then displayed. The top line of this page is the line immediately following the last line of the previous page, and it becomes the current line.

2.2.11 L - Line Number Positioning

Each source line of a Global Cobol listing is assigned a sequence number. To move to a numbered source line you can key the instruction:

```
L
```

and then supply its sequence number, no greater than 9999, in response to the subsequent text prompt. For example:

```
:L    :205
```

Since the line number applies to the input file, you must not select a line which is earlier than any line that has been modified in the current edit.

2.2.12 M - Move a Line to the Hold Buffer

To move a specified line from the current page to the end of the hold buffer, deleting the original line at the same time, key the instruction:

M[line]

For example:

:M12 or :MN or :M

Following the instruction the original line, now deleted, becomes the current line. However it is not possible to process a deleted line, and if an attempt is made to do so $EDIT will skip to process the next line on the current page, but will not advance past the end of the current page.

Once a line has been added to the hold buffer it remains there until either the Z instruction is keyed or control is returned to the monitor. Thus text can be moved between files via the hold buffer.

2.2.13 N - Repeat the Last Instruction until the End of the Page

If the last instruction keyed was D, H, M, T, W, X or Y, then you may key the instruction:

N

to cause the last instruction to act repeatedly on consecutive lines, starting with the line following the current line and ending with the bottom line of the current page, which then becomes the current line.

2.2.14 O - Output the Contents of the Hold Buffer

To insert the entire contents of the hold buffer before a specified line, key the instruction:

O[line]

For example:

:O12 or :ON or :O

If the 'end of file' line is specified then the contents of the hold buffer will be added to the end of the file. The contents of the hold buffer remain unchanged by the instruction.

Following the instruction the screen is refreshed, and an attempt Is made to display all the inserted lines, subject to the restriction that the following line, which becomes the current line, is always displayed.

2.2.15 P - Page Forward or Backward

To page forward within the file you key the instruction:

P

The **last** line of the current page will become the **top** line of the new page displayed, and this line will be the new current line.

To page backward within the file you key the instruction:

    -P

The **top** line of the current page will become the **last** line of the new page displayed, and the top line of the new page will become the new current line.

2.2.16 Q - Execute the Edit Job without Display

To execute the current edit job in "quiet" mode, i.e. without displaying the results of intermediate steps of the edit job, key the instruction:

    Q[line]

For example:

    :Q12 or :QN or :Q

Once you have keyed the instruction you will be prompted for the iteration count (i.e. the number of times that you wish the edit job to be repeated). If you reply <CR>, the iteration count is assumed to be 1.

Before the first iteration of the edit job, the current line is set to the line specified by the instruction. While the edit job is being executed, the display will remain constant, but when the last iteration is complete the screen will be refreshed.

You can key <CTRL W> (or <PR> W) to halt an edit job during execution and return to the instruction prompt. When this keystroke is detected the job is terminated at the end of its current instruction line and the prompt is redisplayed.

2.2.17 R - Replace a Line

To replace a specified line from the current page you key the instruction:

    R[line]

For example:

    :R12 or :RN or :R

Once you have keyed the instruction you will be prompted for the text of the new line by the text prompt, which appears as a colon in column 8. The current line is a field editable default.

Following the instruction the replacement line becomes the new current line. The current page will only be modified to reflect any change.

2.2.18 S - Statement Replacement

To replace the statement part of a line, leaving the comment part unchanged, key the instruction:

    S[line]

For example:

:S12 or :SN or :S

Once you have keyed the instruction you will be prompted for the text of the statement by the text prompt, which appears as a colon in column 8. If you key spaces the statement part of the line is **deleted.**

The statement part of the line consists of all the characters up to but excluding the first asterisk (if any) which is not enclosed within a pair of " characters. The remainder of the line is the comment part. If the new statement keyed starts with a tab or space then it will define its own indentation. If it does not start with a tab or space then it will be given the same indentation as the statement it replaces, unless the statement part of the line is blank, in which case the new statement will begin in the statement starting column (explained in 2.2.26).

Following the instruction the line that has been modified becomes the current line. The current page will only be modified to reflect any change. If you are not, and you wish to check the effect of any amendments, simply key <CTRL A>.

2.2.19 T - Tabulate a Line

To tabulate a line to cause the indentation to correspond to the current defaults you key the instruction:

T[line]

For example:

:T12 or :TN or :T

The line is considered to be in two parts: the statement part consisting of all the characters up to but excluding the first asterisk not enclosed within a pair of " characters and the comment part which begins with that asterisk and includes the rest of the line. Either part may be absent.

The instruction forces the first character of the statement part, which is not itself a space or tab, to begin at the **statement starting column**. The asterisk of the comment is forced to the **comment starting column**, or separated from the statement part by a single space if significant characters of the statement part (not tabs or spaces) extend into the comment area.

Across the page comments (starting in column 1) are not affected by the T instruction.

The statement start column and comment start column are initially 9 and 41 respectively, but can be changed or independently disabled using the $SC and $CC instructions explained in 2.2.26.

Following the instruction the line that has been modified becomes the current line. The current page will only be modified to reflect any change.

2.2.20 U - Use the Edit Job

To execute the current edit job, displaying the results of the intermediate steps of the job, key the instruction:

U[line]

For example:

:U12 or :UN or :U

Once you have keyed the instruction you will be prompted for the iteration count (i.e. the number of times that you wish the edit job to be repeated). If you reply <CR>, the iteration count is assumed to be 1.

Before the first iteration of the edit job, the current line will be set to the line specified by the instruction. As the execution of the edit job proceeds, each individual editing instruction will update the display exactly as if it has been keyed in the normal way.

You can key <CTRL W> (or <SYSREQ> W) to halt an edit job during execution and return to the instruction prompt. When this keystroke is detected the job is terminated at the end of its current instruction line and the prompt is redisplayed.

2.2.21 V - Redisplay with Specified Line as Line 1

To redisplay the current page with the specified line as line 1 you key the instruction:

V[line]

For example:

:V12 or :VN or :V

Additional lines will be read as necessary from the input file to fill up the bottom of the current page. This facility is mainly used in edit job definitions, as it enables the current line to be positioned at the top of the current page, so that an editing instruction can operate on a line whose position relative to the current line is known. There is an example of this use in section 2.2.9.

2.2.22 W - Set the Current Line

To set the current line to a specified line from the current page, key the instruction:

W[line]

For example:

:W12 or :WN or :W

The display of the current page is not affected by the instruction. Keying WN has the effect of incrementing the current line by 1 and keying W has no effect. This instruction would not normally be used directly from the console, but has a use in special edit jobs.

2.2.23 X - Exchange Character Strings

To exchange the first occurrence of a specified character string within a given line for another character string, key the instruction:

```
X[line]
```

For example:

```
:X12 or :XN or :X
```

Once you have keyed the instruction you will be asked for an **exchange string** by the text prompt with the last exchange string used as a field editable default. This may assume any of the following formats, where fff and ggg are character strings which may contain spaces but do not contain a / character:

```
     fff        The first occurrence of fff is removed from the line;
or:  fff/
     fff/ggg        The first occurrence of fff is replaced by ggg.
or:  fff/ggg/

      /ggg          ggg is inserted at the start of the line.
or:  /ggg/
```

The table shows, by way of an example, the exchange strings necessary to change various erroneous lines to the correct form, MOVE A TO B. Note that the final / character is optional.

ERROR LINE	EXCHANGE STRING
MOVEX A TO B	X
MOX VE A TO B	X /
MOXE A TO B	X/V
MOVE A TV B	V /O /

Table 2.2.23 - Typical Exchange Strings

The line you have specified will be modified if an occurrence of fff is found; otherwise it will remain unchanged. In either case it becomes the current line. The current page will only be modified to reflect any change.

2.2.24 Y - Exchange Character Strings with Special Delimiter

This instruction is similar to the X instruction, but may be used to exchange strings containing / characters. You key the instruction:

```
Y[line]
```

For example:

```
:Y12 or :YN or :Y
```

Once you have keyed the instruction you will be asked for an exchange string by the text prompt, with the last exchange string used as a field editable default. This may assume any of the following formats, where fff and ggg are character strings which may contain spaces and d represents any character chosen by you as a delimiter because it is not present in either fff or ggg:

```
     dfff       The first occurrence of fff is removed from the line;
```

```
or:  dfffd

     dfffdggg  The first occurrence of fff is replaced by ggg;
or:  dfffdgggd

     ddgggd    ggg is inserted at the start of the line.
or:  ddggg
```

Thus to correct the line:

```
DISPLAY "//---//"
```

to:

```
DISPLAY "--///--"
```

you might use the delimiter + and thus the exchange string would be:

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

As exchange strings for the X and Y instructions are held independent of each other, it is possible to have two default exchange strings simultaneously available.

2.2.25 Z - Zeroise the Hold Buffer

To clear the hold buffer of all lines of text you key the instruction:

```
Z
```

When $EDIT is loaded the hold buffer is cleared automatically. Lines of text added by the H or M instruction accumulate in the hold buffer and the Z instruction is the only means of removing them.

2.2.26 Change the Editor's Default Tab Columns

To change the statement starting column number (used by the Statement replacement and Tabulate instructions) key the instruction:

```
$SC
```

The current positions of the statement starting column and comment starting column will be indicated by an S and C character respectively appearing on the base line, and you will be prompted for the new statement starting column, a number in the range 1 to 64. This parameter is initially set to 9 and can be reset to this value by keying <CR>. If you reply 0, statement tabulation will be disabled, so that you can use the T instruction to tabulate comments without affecting the indentation of statements. The base line will be refreshed to show the change that has been made to the statement starting column.

To change the **comment starting column** number (used by the Comment replacement and Tabulate instructions) key the instruction:

```
$CC
```

The current positions of the statement starting column and comment starting column will be indicated by an S and C character respectively appearing on the base line, and you will be prompted for the new comment starting column, a number in the range 1 to 64. This parameter is initially set to 41 and can be reset to this value by keying <CR>.

If you reply 0, comment tabulation will be disabled, so that you can use the T instruction to tabulate statements without affecting the indentation of comments. The base line will be refreshed to show the change that has been made to the comment starting column.

Note that the statement starting column must always be to the left of the comment starting column.

2.2.27 Go Forward or Backward a Number of Lines

To go forward a specified number of lines within the file you key the instruction:

nnnn

where nnnn, the number of lines you wish to advance, is an integer in the range 0 to 9999 inclusive. Alternatively you may key:

+*nnn*

where +nnn is an integer in the range +0 to +999, to obtain the same effect.

The instruction causes the specified number of lines to be read from the input file, and the screen is refreshed with the last line read displayed immediately above the base line. The new top line becomes the current line.

If you key <CTRL A> the screen will be refreshed so that the new page displayed shows the effect of any amendments or deletions. If there were deletions extra lines from the input file will be read in order to fill up the screen.

To go backward a specified number of lines within the file you key the instruction:

-*nnn*

where -nnn is an integer in the range -999 to -0.

The instruction causes the last nnn lines of the output file to be reread, and the screen to be refreshed with the last line read displayed at the top. This line becomes the new current line.

If you have not yet modified the file during this edit, you will be able to go right back to the start of the file if required. Otherwise there will be a limit beyond which you will not be able to retreat. This limit will depend on the position of the highest numbered modified or inserted line within the output file and the size of $EDIT's internal line buffer (reduced by the current size of the hold buffer). Thus it is good practice to use the Z instruction to clear the hold buffer of any unrequired lines so that $EDIT's ability to go backward is maximised.

2.2.28 Repeat an Instruction

If you key <CR> to the instruction prompt and the previous instruction was D,F,H,M,Q,T,U,W,X or Y, then the current line will be incremented by 1 and the instruction will be automatically repeated once each time

that <CR> is keyed. If the current line is already set to the bottom line of the current page, a <CR> instruction will be ignored.

If you key <CR> to the instruction prompt and the previous instruction was P, -P, nnnn, +nnn or -nnn then that instruction will be automatically repeated once each time that <CR> is keyed.

At any other time a <CR> instruction will be ignored.

2.2.29 Special Instruction for Edit Job Control

? causes termination of current iteration of edit job if last exchange (X or Y) failed to find match.

! causes termination of current iteration of edit job if last exchange (X or Y) successfully found a match.

These instructions should be used for greater validation. For example, if you wish to change all occurrences of ZWK in MOVE statements to ZWL1, then use:

```
:FN:ZWK
:X:MOVE/MOVE
:?
:X:ZWK/ZWK1
```

2.3 Error and Warning Messages during Editing

The error and warning messages described in this section may be displayed on the base line during the editor's amendment phase. Each is followed by an instruction prompt so that you can take corrective action. If any of these messages appears while an edit job is being executed, the edit job will be terminated and the text "IN EDIT JOB" will be appended to the message.

EDITOR INSTRUCTION NOT RECOGNISED

Table 2.2 lists all the valid replies to the instruction prompt. The optional line information can only be the letter N or an unsigned integer and must directly follow the instruction letter. The unrecognised instruction is ignored and the prompt repeated so that you can correct your keying error.

INVALID LINE NUMBER

The line number keyed as part of a C, D, F, H, I, K, M, O, Q, R, S, T, U, V, W, X or Y instruction is either:

- the letter N, when the current line is already the last line of the page. You cannot change pages using N;
- a number greater than the line number assigned to the last line of the current page;
- not a valid, unsigned positive integer.

The faulty input is ignored and the instruction prompt repeated so you can correct your keying error.

INVALID OR MISSING SEQUENCE NUMBER

This means that the sequence number supplied to the text prompt following the L instruction is either missing, not an unsigned integer, or refers to a line in the input file which precedes a line which has already been modified. In this last case the -nnn instruction may allow you to move back further. Your faulty input will be ignored and the instruction prompt repeated so that you can correct a keying error.

ONLY INSERTS CAN REFER TO END LINE
Only the I and O instructions may select the special *** END OF FILE *** line as their target. This allows you to insert new records at the end of a file or create a new file. You must not select the end line as the target of a C, D, F, H, K, M, R, S, T, V, X or Y instruction. When this message appears your faulty input will be ignored and the instruction prompt repeated so that you can correct a keying error.

INVALID TAB SETTING
The tab setting you have specified in response to the text prompt following the $SC or $CC instruction is not an integer in the valid range, 1-64 inclusive, or you have attempted to set the comment starting column to the left of the statement starting column, or the statement starting column to the right of the comment starting column. The $SC or $CC instruction will be ignored, and the instruction prompt repeated so that you can correct your keying error.

HOLD BUFFER FULL
An H or M instruction has attempted to add a line to the hold buffer but there is no space available. The existing contents of the hold buffer should be output at their destination, so that the hold buffer may be cleared and then reused.

ATTEMPTED TO GO BACK TOO FAR
A -nnn instruction has failed to move the current page as far back as was requested. This may be because you have reached the start of the file, or because you have reached the limit beyond which $EDIT cannot backtrack. If you need to go back further you will have to use the B instruction to terminate the current edit and start again at the start of the output file, or alternatively clear the hold buffer using the Z instruction.

INVALID JOB ITERATION COUNT
The job iteration count must be between 1 and 9999 inclusive.

JOB DEFINITION TABLE OVERFLOW
The edit job being defined by the J instruction is too large to fit in the definition table.

NOT ALLOWED IN EDIT JOB
You may not use an A, B, E, J, Q or U instruction when using the J instruction to define an edit job.

HOLD BUFFER CLEARED
This message confirms that the Z instruction has successfully cleared the hold buffer.

x---x NOT FOUND
The string x---x specified in an F instruction has not been found before the end of file was reached.

3. Inspecting, Searching and Listing

This chapter describes three commands, $INSPECT, $SEARCH and $L, which can be used to examine the contents of files and, in the case of $L, volumes.

$INSPECT will display text and relative sequential print files. It can also be used to extract parts of such files, concatenate them and convert text format to relative sequential print format, or vice versa. $SEARCH will search text files for specified strings. $L will produce a dump of the whole or part of any type of file or volume.

INSPECT INSTRUCTION	DESCRIPTION
E	Exit to process next file (see 3.1.10).
F	Find a line containing a specified string and display a page with the line at the top of the page (see 3.1.4).
G	Get a line starting with a specified string and display a page with the line in the middle of the page (see 3.1.5)
P	Page, i.e. move forward so that the bottom line of the current page becomes the top line of the new page.
-P	Page backwards, i.e. move backwards so that the top line of the current page becomes the bottom line of the new page.
WS WE	Used to print or write out selected parts of the file under inspection (see 3.1.7 on).
$LS	Prompts you for the starting column of the part of the line to be displayed (see 3.1.6)
n	(a number between 0 and 9999). Moves forward n lines and displays the page thus selected.
-n	(a number between -9999 and -1). Moves backward n lines and displays the page thus selected.
<CR>	Following an F, G, P, -P, n or -n instruction it causes the instruction to be repeated, otherwise it is ignored.
W	Enter wide mode (if available) – use compressed characters.
N	Re-enter narrow mode.

Table 3.1 - Summary of Inspect Instructions

3.1 Processing Text and Print Files using $INSPECT

The $INSPECT command allows you to examine any text or print file using a display terminal. It is frequently used to examine compilation and map listings which have been written to direct access storage to avoid the overhead of printing.

$INSPECT can also be employed to print selected parts of any text or print file, convert between text and print format, or produce a file containing selected parts of one or more input files. It can be used to concatenate files or produce a re-arrangement of the text within a file.

3.1.1 The Inspect File Prompt

When you run $INSPECT it signs on by outputting the inspect file prompt, so that you can specify the file-id and unit-id of the file you wish to process. For example:

```
$48 INSPECT:L.SALES     UNIT:205
```

indicates that the operator wishes to inspect the compilation listing file L.SALES on unit 205.

3.1.2 To Quit

To quit and return to the monitor simply key <ESCAPE> in response to the inspect file prompt.

3.1.3 Using Wide Screen Mode

If your screen facilitates wide mode, where the characters are compressed to display 132 characters across the screen, then $INSPECT will ask you:

```
$48 USE WIDE MODE?:
```

If you specify a file which has a line length that is greater than the current screen width. Use Y or <CR> to use the compressed mode characters, N to stay in normal mode.

Note that $INSPECT will always make use of the maximum logical width of screen possible, so that you may, where appropriate, use <SYSREQ> L and R to inspect the listing, rather than needing to use the $LS instruction.

Figure 3.1 - Inspecting a file

3.1.4 File Inspection Instructions

Once you have entered the details of the file to be inspected, the first page of the file is displayed, followed by the inspection instruction prompt; this is a single colon displayed on the base line, in character position 1.

The group of lines displayed by $INSPECT at any time is termed a page. The number of lines in a page is one less than the number available on the screen as the base line is always reserved for the instruction prompt, as shown in figure 3.1.

When lines are displayed print control characters such as form feeds or multiple line feeds are ignored. Each line appears on the screen directly below its predecessor. Normally, if the line length is greater than the screen width, only the start of each line is displayed. However, other parts of longer lines may be examined by using the $LS instruction.

The instructions that you may key in response to the prompt are summarised in Table 3.1 and explained in more detail, when necessary, in the remainder of this section. Note that if you key an instruction such as -9999 which would take you past the beginning of the file then the very first page of the file will be displayed again. If you select a line past the end of the file by, say 9999, or if an F or G

instruction fails to find a line satisfying the specified search criteria then the special message:

```
*** END OF FILE ***
```

will appear as the top line of an otherwise blank page.

3.1.5 F - To Search for a String within a Line

This instruction has a similar function to the F instruction of $EDIT. Once an F instruction has been keyed, an extra colon prompt is displayed on the base line. You must reply with the characters of the string that you are searching for. Spaces in the string are significant. Sufficient characters should be supplied to uniquely identify the required line, but if the search finds an unwanted line containing the search string the search can be resumed again by keying <CR>.

The search always starts at the second line of the screen and whenever a match is found a page is displayed with the line containing the specified string at the top of the page.

3.1.6 G - To Search for a String starting a Line

If a G instruction is keyed then an extra colon prompt is displayed on the base line. You must reply with the starting characters of the line to be searched for. Spaces or tabs at the beginning of the line must not be keyed. Multiple spaces or tabs should not be keyed within the search string. Sufficient characters should be supplied to uniquely identify the line, but if the search finds an unwanted line starting with the characters keyed the search can be resumed again by keying <CR>.

The first line examined is normally the second line of the screen except when one search (G or null resumption) immediately follows another. In this case the first line examined is always the one immediately following the last line found. Whenever a line is found a page is displayed with that line in the middle of the screen.

The G instruction can be used to search for errors in a compilation listing, as all error lines start with an asterisk (see example 2).

Note that the G instruction differs from the F instruction in two respects: it will only find strings located at the start of a line and it displays the line containing the string in the middle of the screen.

3.1.7 $LS - Set Line Start

$INSPECT normally displays each line with its first character in the first column of the screen. Up to 132 characters may be displayed on a line, depending on the width of the screen. If the screen is narrower than the lines being inspected, you can use the $LS instruction to inspect other parts of the line. If you key the $LS instruction you will be prompted, by a single colon on the same line, for the starting column (1 to 150) of the part of the line to be displayed. For example:

```
:$LS  :41
```

This example causes $INSPECT to display columns 41 onwards of the files being inspected, rather than columns 1 onwards. This option remains in force until overridden by a subsequent $LS instruction.

Note that the $LS instruction does not affect the search instructions F and G, which always examine lines starting at column 1. Nor does it affect the operation of the WS and WE instructions.

3.1.8 Supplying the Output File for WS and WE Instructions

The WS and WE instructions are used to print or write to an output file selected parts of the files under inspection. Each time that a WS or WE instruction is keyed you are prompted for the **file-id** and **unit-id** of the output file, unless an output file is already open. The new file will be produced as a text file if the file-id you supply begins with an S. prefix and is on a direct access unit which is not a spool unit. In all other cases a relative sequential print file will be produced. You can use this convention to convert from text to print file format, or vice versa. For example, assuming direct access unit 205 is not a spool unit:

```
OUTPUT FILE:S.SALES1    UNIT:205
```

will result in a text file named S.SALES1 whereas:

```
OUTPUT FILE:SALES2      UNIT:205
```

will result in a relative sequential print file named SALES2.

You may reply <CR> to both the OUTPUT FILE: and the UNIT: prompts. In this case a relative sequential print file will be produced on the unit assigned to $PR, the logical printer. If $PR is assigned to a direct access device the output file will be named D.$INSPE.

3.1.9 WS - Set Write Start Pointer

The WS instruction sets the write start pointer to the line which is at the top of the current page displayed on the screen. The write start pointer is initialised to address the first line in the file being inspected, and is reset whenever an E instruction is keyed.

3.1.10 WE - Set Write End Position

The WE instruction causes lines to be written to the output file from the line addressed by the write start pointer up to and including the line at the top of the page currently displayed on the screen. If the line pointed to by the write start pointer is later in the file than the top line of the screen the message:

```
START LINE PAST TOP OF SCREEN - IGNORED
```

is displayed on the base line and the instruction is ignored.

Once the lines have been written to the output file, the prompt:

```
DO YOU WISH TO CLOSE THE OUTPUT FILE?:
```

appears. You should key Y if you intend to end extraction and this will cause the output file you have constructed to be correctly

closed. If, however, you reply <CR> (or any single character apart from Y) the file remains open so you can continue extending it.

3.1.11 E - End Inspection of Current File

To examine a second (or subsequent) file, or to end inspection, key the E instruction. The inspect file prompt will then appear to allow you to either continue processing files or to return to the monitor by keying <ESCAPE>.

3.1.12 W - Enter Wide Mode

This causes the display to go into the compressed wide mode, if the screen facilitates of this mode.

3.1.13 N - Re-enter Narrow Mode

This causes the display to revert to non-compressed, narrow mode.

3.1.14 Example 1 - Examining a Listing

The following dialogue takes place on a 24-line screen if you examine the first, second and last page of the compilation listing L.SALES on unit 205:

```
GSM READY:$INSPECT
$48 INSPECT:L.SALES UNIT:205
.........
.........                     (first page of 23 lines)
.........
:<CR>
.........
.........                     (second page of 23 lines)
.........
:9999                         (force to end of file)
*** END OF FILE ***
:-23                          (display last page)
.........
.........
.........
:E
$48 INSPECT:<ESCAPE>
GSM READY:
```

3.1.15 Example 2 - Examining Error and Warning Messages

You wish to examine all the errors and warnings in a compilation listing file L.SALES which has been produced on subunit 209. The following dialogue will take place:

```
GSM READY:$INSPECT
$48 INSPECT:L.SALES UNIT:209
.........
.........                     (first page of 23 lines)
.........
:G :*                         (search for lines starting with '*')
.........
.........
*** ERROR.....                (page containing 1st error)
.........
.........
:<CR>
.........
.........
*** ERROR....                 (page containing 2nd error)
.........
.........
```

```
:<CR>
*** END OF FILE ***
                              (end of file - all errors inspected)
:E
$48 INSPECT:<ESCAPE>
GSM READY:
```

3.1.16 Example 3 - Printing Part of a Listing

You wish to obtain a print-out of source listing S.SALES on unit 205 for the program section MAIN only. You know that the only EXIT statement within section MAIN is the final statement of the section. The following dialogue will take place:

```
GSM READY:$INSPECT
$48 INSPECT:S.SALES UNIT:205
.........
.........                     (first page)
.........
:F :SECTION MAIN
SECTION MAIN
.........
.........                     (page with SECTION MAIN as top line)
.........
:WS OUTPUT FILE:<CR> UNIT:<CR>
                              (printer output)
:F :EXIT
EXIT
.........
.........                     (page with EXIT as top line)
.........
:WE DO YOU WISH TO CLOSE THE OUTPUT FILE?:Y
:                             (section is printed)
$48 INSPECT:<ESCAPE>
GSM READY:
```

3.1.17 Example 4 - Converting an RS File to a Text File

You require to convert a relative sequential file, COMMFILE, to text file format as S.COMMS, so that it can be modified using $EDIT. COMMFILE is online on 204 (not a spool unit) and S.COMMS will occupy the same volume. You use WS to convert the whole file:

```
GSM READY:$INSPECT
$48 INSPECT:COMMFILE UNIT:204
.........
.........                     (first page)
.........
:WS OUTPUT FILE:S.COMMS UNIT:204
:9999
*** END OF FILE ***

:WE DO YOU WISH TO CLOSE THE OUTPUT FILE:Y
:E
$48 INSPECT:<ESCAPE>
GSM READY:
```

3.1.18 Example 5 - Concatenating Text

While editing the file S.PROG on subunit 201, $EDIT has exhausted its output file. The editor work file, which is named S.$EDI01 in a single-user system, will contain the first part of the source of S.PROG, containing most, if not all, of the updates applied in this editing session, and S.PROG itself will still contain the original source. This example shows how $INSPECT can be used to concatenate the editor work file with the latter part of S.PROG, producing the source

file S.RECOV, which, after careful checking, is renamed as S.PROG, completing the recovery process. The following dialogue will take place:

```
GSM READY:$INSPECT
$48 INSPECT:S.$EDI01 UNIT:201
.........
.........                     (first page of editor work file)
.........
:9999                         (to end of editor work file file)
*** END OF FILE ***
:-P
.........
.........                     (last page of edit work file -
.........                     make a note of the last line)
:P                            (return to end of editor work file)
:WE OUTPUT FILE:S.RECOV UNIT:201
DO YOU WISH TO CLOSE THE OUTPUT FILE?:N
:E                            (end inspect of S.$EDI01)
$48 INSPECT:S.PROG UNIT:201
.........
.........                     (first screen of original source file)
.........
:     (supply inspect instructions to cause the first line
      that is to be concatenated from the original source
      file to be positioned at the top of the page)
.........
.........                     (first page that is to be concatenated)
.........
:WS
:9999                         (advance to end of original source file)
*** END OF FILE ***
:WE DO YOU WISH TO CLOSE THE OUTPUT FILE?:Y
:E
$48 INSPECT:S.RECOV UNIT:201
.........
.........                     (first page of recovered source file)
.........
:     (supply inspect instructions to check carefully that
      the file has been successfully recovered)
:E
$48 INSPECT:<ESCAPE>
GSM READY:
```

Now that S.RECOV has been checked to be satisfactory, $F can be used to rename it as S.PROG, replacing the original source. Note that the editor work file will be re-used the very next time that $EDIT is run, so it is essential that this recovery procedure is carried out (or the editor work file is renamed) before $EDIT is run again.

3.2 Searching for Strings using $SEARCH

You run the $SEARCH command to produce, from one or more Global Cobol text files, a listing of all the lines which contain one of the specified search strings. Up to 10 different search strings may be specified at one time. Appendix A contains a typical search listing.

3.2.1 The Search Prompt

$SEARCH begins by prompting you for the strings for which it is to search. When a string has been specified the prompt is repeated to allow another string to be specified. Up to 10 strings may be specified, and each may be of up to 50 characters in length. You may reply <CR> to indicate that you have finished specifying search strings before ten have been supplied. For example, the following

dialogue would occur if you wished to search for the two strings DO WHILE and DO UNTIL:

```
GSM READY:$SEARCH
$68 SEARCH FOR:DO WHILE
$68 SEARCH FOR:DO UNTIL
$68 SEARCH FOR:<CR>
$68 LISTING UNIT:
```

Spaces are significant in search strings, and thus:

```
DO WHILE
```

and:

```
DO WHILE
```

are different strings.

3.2.2 The Listing Unit Prompt

Once the search strings have been specified, the listing unit prompt is displayed:

```
$68 LISTING UNIT:209
```

In the above example, the output will be written to file D.$SRCH on subunit 209. If this file already exists it will be deleted, and the new file will replace it. If you reply <CR> to the prompt the file will be written to the unit assigned to $PR. The special reply of <CTRL A> will cause it to be listed on the screen.

3.2.3 The File Prompt

Once the output unit has been specified, the file prompt is displayed:

```
$68 FILE:
```

To search an individual file, you must supply the file-id and unit-id of the file. If you do not explicitly key a prefix $SEARCH will assume that the file is a conventionally named source file, and will append the S. prefix. For example, the dialogue:

```
$68 FILE:SA100 UNIT:209
```

will cause file S.SA100 on subunit 209 to be searched.

If you reply <CR> to the file prompt then all the text files on that unit will be offered for processing. This option is particularly useful if you wish to find all the references to a variable in a suite of programs. For example the dialogue:

```
$68 FILE:<CR> UNIT:209
```

will cause the first text file on subunit 209 to be offered for processing. For example:

```
$68 SEARCH filename?:
```

You must key Y to search the file, <CR> or N (or any other single character except Y) to avoid searching it.

You can use comparable selection codes to those in $F to select files to be searched, e.g. <CR> to be prompted for each file in turn, <CTRL A> to avoid processing any, and <CTRL B> to process all the files on the unit.

Once you have specified the file or unit the message:

```
SEARCHING filename UNIT uuu
```

will be displayed, and then the file or files specified will be searched and a listing printed or displayed. The message:

```
STRING(S) NOT FOUND
```

will be displayed when appropriate.

When the search is complete the file prompt will be redisplayed to permit you to search further files for the same strings.

3.2.4 To Quit

To quit and return control to the monitor key <ESCAPE> to either the file or the unit prompt.

3.3 Listing Files of any Format using $L

The $L command allows you to produce a hexadecimal dump of all or part of a file or volume. Files of any organisation and format can be processed. The dump appears as fixed-length records and each line contains hexadecimal and ASCII translations side by side.

3.3.1 The Input File Name Prompt

When you run $L it displays an identifying message followed by the input file name prompt:

```
GSM READY:$L
$63 SELECTIVE FILE LIST/DUMP PROGRAM
$63 INPUT FILE NAME:
```

You must specify the file-id and unit-id of the file you require to list. For example:

```
$63 INPUT FILE NAME:SADATA UNIT:209
```

The special reply <CTRL B> to the input file name prompt informs $L that you wish to dump the sectors of a volume, and you are then prompted for the unit containing that volume. The unit supplied must address a discrete direct access volume or domain, not a subvolume of a domain. Each sector of the volume is treated as though it were a record of an RS file, where the physical sector number corresponds to the record number and the record size is the sector size. For example:

```
$63 INPUT FILE NAME:<CTRL B> UNIT:101
```

If you have replied previously to the input file name and unit prompts during this execution of $L, a reply of <CR> to either prompt is acceptable and causes input to default to the previously supplied value.

3.3.2 To Quit

You may quit and return to the monitor by keying <ESCAPE> to any prompt.

3.3.3 The Listing Option Prompt

Once the input file has been defined the listing option prompt:

```
$63 LISTING OPTION:
```

is output repeatedly to enable you to specify zero or more options to control the listing. The valid options and their meanings are as follows:

An — This option is only valid when dumping the contents of a volume. The value n is a one or two digit access option specifying the interleaving of sectors within a track;

D — Direct all subsequent dumps to the display. This option remains in force for the current file until cancelled by a P option;

En — Set the ending record number to n, a decimal integer. The first record number is 1;

On — Set a byte offset n which is to be added to all file or volume addresses. This permits a correctly-aligned dump in cases where fixed-length records do not start at the first byte of the file, e.g. the overflow records of an IS file; key OVERFLOW to align in the overflow area;

P — Redirect all subsequent dumps to unit $PR, the logical printer;

P=uuu — Redirect all subsequent dumps to unit uuu.

Rn — Set the record length to n, a decimal integer giving the size of the record in bytes;

Sn — Set the starting record number to n, a decimal integer;

n — Select single-record mode, with n, a decimal integer, being the number of the first record to be dumped.

The special replies of <CTRL C> and ? will list on the screen the current options in force, including details of the input file type, record length and size, data size and total file size.

Each time a new input file or volume is selected, the listing options are restored to default values. The dump will be directed to the console, the starting record number and ending record number will be set to select every record/sector of the file/volume, and the byte offset will be set to 0. For a file, the record length will be set to that established when an RS or IS file was created, or to 256 if the file is of another organisation, without a fixed record length. For a volume, the record length will be set to the sector size and the access option will be set to 1 (no interleaving).

To proceed when all the required options have been specified, key <CR> in response to the final listing option prompt. If you are dumping a range of records, the starting record number will be incremented as each complete record is listed. When the listing is complete, the listing option prompt is repeated. You may now change the settings of one or more options to obtain another listing of the same file.

If you have selected single-record mode by supplying a record number to the listing option prompt, then after each record has been dumped the prompt:

```
$63 NEXT RECORD?:
```

will appear to allow you to select the next record to be dumped. You key Y or <CR> to dump the next higher record, or a decimal integer to select a specific record. If you reply N the output process will be terminated and the listing option prompt will be redisplayed.

You may return to the input file prompt by keying <CTRL A> to the listing option prompt.

The error message:

```
RECORD SIZE TOO LONG
```

indicates that the dump cannot be produced because $L has insufficient work space to hold a single record of the file. In this case use the listing option prompt to specify a smaller record size (ideally a divisor of the true record size) and try again.

3.3.4 Output to the Console

When the dump is output to the console, each page of output is followed by the prompt:

```
$63 NEXT PAGE?:
```

which appears on the base line immediately below the dumped data of the file. You key Y or <CR> to proceed to obtain the display of the next page. If you reply with a decimal integer, the starting record number will be set to that value and the next page displayed will start with the dump of that record.

If you reply N the output process will be terminated and the listing option prompt will be redisplayed. If you now key <CR> to this prompt, the listing will resume from the beginning of the last record to be output, if this were only partially displayed, or from the next record otherwise. Thus <CR> has the effect of aligning the dump so that the start of the current record coincides with the start of the display.

3.3.5 Output to Direct Access Storage

If your computer does not possess a printer, you may decide to output the dump to a direct access volume (e.g. diskette) in order to have it printed on another machine later. In this case the dump will occupy the file D.$L. If this file already exists on the output volume it will be deleted and the new dump file will replace it.

3.3.6 Operating Notes

You can use the E listing option to access records beyond the end of the file (though not beyond the end of the extent allocated to the file). This allows you to examine the records written to a new file by a program which has failed before closing that file.

If you use $L on an indexed sequential file and allow the start, end and record length to be defaulted, the records of the prime data area will be output satisfactorily, then will come the index, which may not be meaningful to you, and this will be followed in turn by fragmented records of the overflow area. This is because the program is not sensitive to file organization and can, indeed, be used to dump every type of file. It simply treats any file as a string of contiguous fixed length records.

Figure 3.2 - Displaying records from a file

It is possible to use the O option to offset all file addresses by the byte address of the first overflow record, and thus obtain a correctly-aligned dump of the overflow area; a more effective approach, however, is to run the conversion command program, $CONV, to extract a relative sequential file from the indexed sequential file and then use $L on the new file. $CONV is described in the Global Utilities Manual. Example 3 shows it being used in conjunction with $L to dump an indexed sequential file.

3.3.7 Example 1 - Displaying Records from a File

This example shows how you would use $L to display selected records of an RS file on the console, in hexadecimal format. Firstly, to display the first few records:

```
GSM READY:$L
$63 SELECTIVE FILE LIST/DUMP PROGRAM
$63 INPUT FILE:SADATA UNIT:205
$63 LISTING OPTION:<CR>
.........
......... (first page of information)
.........
$63 NEXT PAGE?:<CR>
.........
......... (second page of information)
.........
$63 NEXT PAGE?:
```

If a page finishes with a partial record, then the next page can be easily displayed starting with a complete dump of this record as follows:

```
$63 NEXT PAGE?:N
$63 LISTING OPTION:<CR>
.........
......... (next page aligned with start of record)
.........
$63 NEXT PAGE?:
```

To continue displaying at a specific record, without returning to the listing option prompt, proceed as follows:

```
$63 NEXT PAGE?:1234
.........
......... (page starting with record 1234)
```

```
.........
$63 NEXT PAGE?:
```

Figure 3.2 shows a typical page of information. Note how the display is subdivided into numbered records. Each line holds 16 bytes of information which appears both in hexadecimal form and in an ASCII translation.

3.3.8 Example 2 - Comparing Two Files

The file utility's CFI instruction (described under $F in the Global Utilities Manual) has been used to compare two files, TESTA and TESTB and has discovered a discrepancy at byte 11406. You decide to print bytes 11000 to 11999 of each of the files to see the problem in context:

```
GSM READY:$L
$63 SELECTIVE FILE LIST/DUMP PROGRAM
$63 OUTPUT UNIT:$PR
$63 WIDE FORMAT?:Y
$63 INPUT FILE NAME:TESTA UNIT:205
```

3.3.9 Example 3 - Dumping an IS File

Following a program test you wish to dump the indexed sequential file SAINDEX. You first of all use $CONV (described in detail in the Global Utilities Manual) to convert the file to relative sequential form as SAWORK and then you print all the records of this file:

```
GSM READY:$CONV
$65 CONVERSION INPUT:SAINDEX UNIT:205
$65 CONVERSION OUTPUT:SAWORK UNIT:<CR> SIZE:<CR> TYPE:R
$65 CONVERTING
(the conversion process, which may take some time, depending
on the size of SAINDEX, takes place)
$65 CONVERSION SUCCESSFUL
$65 CONVERSION INPUT:<ESCAPE>
GSM READY:$L
$63 SELECTIVE FILE LIST/DUMP PROGRAM
$63 INPUT FILE NAME:SAWORK UNIT:205
$63 LISTING OPTION:P
$63 LISTING OPTION:<CR>
(the whole work file is now printed)
$63 LISTING OPTION:<ESCAPE>
GSM READY:
```

3.3.10 Example 4 - Dumping the Directory Track of a Volume

You wish to obtain a printed dump of the directory track of a diskette on unit 100, (which need not necessarily contain a Global volume). You already know that the directory track is the first track on the volume and contains 26 sectors, i.e. sectors 1 to 26 are to be dumped. The sectors are interleaved such that every other sector is skipped (Global System Manager access option 52). Proceed as follows:

```
GSM READY:$L
$63 SELECTIVE FILE LIST/DUMP PROGRAM
$63 INPUT FILE NAME:<CTRL B> UNIT:100
$63 LISTING OPTION:P
$63 LISTING OPTION:A52
$63 LISTING OPTION:E26
$63 LISTING OPTION:<CR>
(the dump is now produced in wide format on the printer)
$63 LISTING OPTION:<ESCAPE>
GSM READY:
```

4. Compiling, and Cross-Referencing Programs

This chapter describes two commands, $COBOL and $XREF, which allow you to compile and cross-reference Global Cobol source files.

The Global Cobol compiler, $COBOL, compiles a source file, together with up to four copy libraries to produce a compilation file and a compilation listing. The compilation file contains relocation information and, in most cases, unresolved references to subroutines. It therefore cannot be immediately executed, but must be linkage edited using $LINK, as described in Chapter 6.

The cross-reference utility, $XREF, produces, from a Global Cobol source file and any number of copy libraries, a listing in alphabetical order of all the symbols used in the program together with the number of the line on which each symbol is defined and the numbers of the lines from which each is referenced. This utility is particularly useful when amending a program, as it enables a programmer to examine every reference to a particular variable before changing the usage of that variable. A cross-reference listing would not normally be produced each time a program is compiled, but it is recommended that one is produced for every completed program used for live running.

Both commands use two work files, the error message file, $COMWORK, and a scratch file, $COMWKxx where xx is a number unique within the unit concerned. The error message file must be on the unit from which the command was run. The scratch file is allocated on the unit assigned to $C, and will be deleted at the end of the run.

If $COMWORK is not present on the unit from which $COBOL or $XREF is run, the command immediately terminates with the message:

```
$43 MESSAGE FILE NOT FOUND
```

Similarly, if there is less than the minimum amount of space required for the scratch file (approximately 2u - 31K, where u is the size of the user area), or there are no free directory entries in the directory, the command terminates with the message:

```
$43 INSUFFICIENT SPACE TO CREATE WORKFILE ON uuu
```

where uuu is the unit address involved, determined from the assignment of $C.

Note that if you terminate either $COBOL or $XREF prematurely by keying <CTRL W> the scratch file will still be allocated. You should use $F's DEL instruction to delete it.

All the files used by the compiler and cross-reference, including the command library containing $COBOL or $XREF, must remain online throughout the processing. In systems where storage is limited it is usual to place the central copy library used by a project on the volume occupied by $COBOL.

Appendix A contains the listings resulting from compiling and cross-referencing an example program. These listings are annotated to explain the meanings of the various fields.

4.1 Compiling Programs using $COBOL

You run the $COBOL command to compile a Global Cobol source file named S.xxxxxx to produce a compilation file C.xxxxxx and a listing file L.xxxxxx. Both the compilation file and the listing file are optional and may be individually suppressed if they are not required.

4.1.1 The Source File Prompt

$COBOL signs on by prompting you for the file-id of the source file to be compiled and its unit-id. If you do not explicitly key a prefix when you input the file-id $COBOL will assume that the file is a conventionally named source file and will append the S. prefix by default. For example:

```
GSM READY:$COBOL
$43 SOURCE:SA100 UNIT:205
$43 COMPILATION UNIT:
```

indicates that the operator wishes to compile source file S.SA100 on unit 101.

4.1.2 The Compilation Unit Prompt

When the source file has been specified you will be prompted for the unit and size of the compilation file to be produced:

```
$43 COMPILATION UNIT:205 SIZE:8K
$43 COPY LIBRARY:
```

The suffix of the file is always the same as that of the source file. In this example, if the source file is S.SALES the compilation file would be C.SALES on unit 205, with size 8K bytes.

If you reply <CR> to the unit prompt then the compilation file will be placed on the same unit as the source file.

If you reply <CTRL A> to the unit prompt this means that you do not wish to produce a compilation file. In this case the size prompt is not output.

You must reply to the SIZE: prompt with one of the following:

- a decimal number, the number of bytes to be allocated;
- a decimal number, between 1 and 99 inclusive, immediately followed by the letter K, indicating the number of kilobytes to be allocated. (K = 1024);
- <CR>, when 20K bytes will be allocated.

At the end of the compilation any unused space possessed by the compilation file will be released for re-allocation.

4.1.3 The Copy Library Prompt

Once the compilation file has been specified the copy library prompt is displayed. It requests the library's file-id and unit-id. If you do not explicitly key a prefix when you input the file-id, $COBOL will

assume that the library is a conventionally named source file and append the S. prefix by default. For example:

```
$43 COPY LIBRARY:SL UNIT:205
```

causes file S.SL on unit 205 to be used as the copy library.

When you have specified a copy library, the prompt is redisplayed, so that you can specify further copy libraries. Note that a book is always copied from the first library specified in which it exists, so that, for example, if the same book were in both the first and second copy libraries, it would be copied from the first library.

The reply <CR> to the copy library prompt indicates that you have finished specifying copy libraries. If you reply <CR> to the first copy library prompt this means that no copy library is needed, and COPY statements found in the compilation will be flagged as errors.

If you reply <CR> to the unit prompt then the file is assumed to be on the same unit as the previous copy library specified. However, if you reply <CR> to the very first unit prompt, the first copy library is considered to be on the same unit as the source file itself.

4.1.4 The Listing Unit Prompt

Once any copy libraries required have been specified, the listing unit prompt is displayed:

```
$43 LISTING UNIT:205
$43 COMPILER OPTION:
```

If the suffix of the source file is SA100 then the above example will cause the listing to be written to file L.SA100 on unit 205. If the listing unit is a direct access device rather than a printer then the file will be allocated the largest available space on the volume and will be written in text file format (which is more compact than print file format). Any unused space will be released at the end of the compilation.

If <CR> is keyed in response to the prompt the file will be placed on unit $PR. You should note that when the listing is output to a real printer, the print file is only opened when the compiler is ready to produce the listing at the beginning of its second pass. If you have forgotten to turn the printer on and it is a very simple device unable to provide Global System Manager with status information, then this may cause the compilation to mysteriously "hang" after a few minutes. The remedy is to follow the instructions laid down in your Global Operating Manual, and always switch any printers on when you initiate Global System Manager.

If you reply <CTRL A> to the prompt no listing will be produced.

4.1.5 The Option Prompt

When the listing unit has been specified the option prompt is displayed:

```
$43 COMPILER OPTION:
```

You may reply with any single valid compiler option or <CR>. If you specify a compiler option the same prompt is redisplayed so that you can input further options. The reply <CR> indicates that you have finished specifying options, and that compilation is to commence. For example, the following dialogue indicates that the printing of statements included from copy books is to be suppressed, and a symbol table is to be produced at the end of the listing:

```
$43 COMPILER OPTION:NCX
$43 COMPILER OPTION:ST
$43 COMPILER OPTION:<CR>
$43 COMPILING
```

The available compiler options are described in detail in Appendix B of the Global Cobol Language Manual, and summarised in table 4.1:

OPTION	**DESCRIPTION**
CX/NCX	List contents of copy books.
CG/NCG	Display no. of line being processed on <CTRL G>
ED/**NED**	Force even byte data alignment.
LN/**NLN**	Use long (31 character) names.
PL=nnn	Page length (default $$PAGE).
SD/NSD	Generate symbolic debug information.
SL/NSL	Print all source lines.
ST/**NST**	Print symbol table.
TC/**NTC**	Print table of contents.
TE/NTE	Terminate job management if errors are detected.
TR/NTR	Generate trace information.
TW/**NTW**	Terminate job management if warnings are detected.

Table 4.1 - Summary of Compiler Options (Defaults Underlined)

If an option has been specified in an OPT statement appearing in the source file it can be overridden by respecifying it in response to the option prompt. If, in response to the option prompt, an option is specified twice, or conflicting options are specified, then the last one to be specified is the one which is used.

4.1.6 Source Program Validation

Once any options required have been input the message:

```
$43 COMPILING
```

is displayed and the initial stage of the compilation, known as the first pass, begins.

The source program is input and validated. Should a line giving rise to an error or warning condition be encountered it is displayed on the terminal, along with the appropriate explanatory message. Once this first pass validation is complete, the compiler outputs the message:

```
$43 END OF FIRST PASS
```

If your program contains some trivial error you will be able to cancel the compilation and return to the monitor by keying <CTRL W>, so that you can run $EDIT to make any amendments required. In this way you save time by avoiding the compiler's second pass, which normally

accounts for about 60 percent of the time spent in processing a program.

Note that although the majority of error and warning conditions can be listed on the terminal during the first pass validation process, there are a few conditions which cannot be displayed in this way, since they are only detected on the second pass. Normally when compiling a new program, or one subjected to a major amendment, you should allow it to complete, even if first pass errors are detected, so that you obtain a compilation listing for thorough checking.

The error or warning messages appear on the terminal or listing along with the source statement to which they apply. Each has a message number which allows you to refer to the appropriate part of Appendix G of the Global Cobol Language Manual where the condition is described in detail, and a recovery action suggested.

4.1.7 Summary and Completion Messages

When the compilation completes normally $COBOL closes all its files and then outputs the following summary messages:

```
$43 NUMBER OF ERRORS eeee
$43 NUMBER OF WARNINGS wwww
```

The quantities eeee and wwww are decimal numbers which will appear as 0 if no errors or warnings have been detected: these totals are accurate and include conditions detected on both passes.

Some errors that $COBOL detects are fatal inasmuch as they destroy the integrity of the compilation and mean that not only the statement flagged in error, but other correct statements, fail to compile good code. If such an error occurs during a run where a compilation file was being produced, that file is not created, and a further message is displayed:

```
$43 *** COMPILATION FILE NOT CREATED BECAUSE OF SERIOUS ERRORS
```

Once the two (or three) summary messages have been produced, the compiler displays its final message:

```
$43 COMPILATION COMPLETED
```

and then returns control to the monitor.

4.1.8 Processing if the Compilation File Becomes Full

If the space allocated to the compilation file is insufficient, once $COBOL is unable to write a record it outputs the message and prompt:

```
$43 INSUFFICIENT SPACE ON COMPILATION FILE
$43 SPECIFY NEW COMPILATION UNIT:
```

You should reply with the unit-id of the new device to which the compilation is to be written. If you key <CR> the file will be reallocated on its current unit. The size of the new file allocated will be 10K bytes larger than the original file. If you key <CTRL A> the compilation will continue but will not produce a compilation file, only a listing file.

If you do not wish to continue once the compilation file has become full, key <ESCAPE> in response to the new compilation unit prompt.

4.1.9 Respecifying the Listing File Unit

There are two cases in which you may be asked to change the listing file unit, indicated by the message:

```
$43 PRINTER IN USE
```

or:

```
$43 INSUFFICIENT SPACE ON LISTING FILE
```

preceding the prompt:

```
$43 SPECIFY NEW LISTING UNIT:
```

The first message can only occur if you are attempting to write to a real printer in a multi-user environment. (Not recommended - you should use a spool unit.) The second message means that the listing file has become full and there is no room to write any more output.

You should reply to the prompt by specifying the unit-id of the new device to which the listing file is to be written. You can specify the same device if you believe the printer is no longer in use, or, in the case of the second message, you are willing to let the later part of the listing overwrite the earlier part.

The following special replies are also valid:

<CR> write the remainder of the listing to the logical printer assigned to $PR;
<CTRL A> suppress further printing of the listing and continue;
<ESCAPE> abandon the job and return to the monitor.

4.1.10 Example

You require to compile source program S.SA100 on unit 205 to produce compilation file C.SA100 and listing L.SA100. L.SA100 is to be written to the logical printer assigned to $PR. Two copy libraries are to be used, S.SL on unit $C and a special copy library S.SL100 on unit 205 which contains new versions of some of the books in library S.SL, and hence must be specified first so that the new versions are used. The following dialogue takes place:

```
GSM READY:$COBOL
$43 SOURCE:SA100 UNIT:205
$43 COMPILATION UNIT:<CR> SIZE:<CR>
$43 COPY LIBRARY:SL100 UNIT:<CR>
$43 COPY LIBRARY:SL UNIT:$C
$43 COPY LIBRARY:<CR>
$43 LISTING UNIT:<CR>
$43 COMPILER OPTION:<CR>
$43 COMPILING
(the compilation now begins)
$43 END OF FIRST PASS
(the second pass now takes place)
$43 NUMBER OF ERRORS 0
$43 NUMBER OF WARNINGS 0
$43 COMPILATION COMPLETED
GSM READY:
```

4.2 Cross-Referencing Programs using $XREF

You run the $XREF command to produce, from a Global Cobol source file named S.xxxxxx, an alphabetic listing X.xxxxxx of all the symbols used in the program together with the numbers of all the lines on which each symbol is defined or referenced. Undefined symbols are indicated by an asterisk in the definition column. Appendix A shows some typical cross-reference listings.

4.2.1 The Source File Prompt

$XREF begins by prompting you for the file-id of the source file to be referenced and its unit-id. If you do not explicitly key a prefix when you input the file-id $XREF will assume that the file is a conventionally named source file and will append the S. prefix by default. For example:

```
GSM READY:$XREF
$54 SOURCE:SA100 UNIT:205
$54 COPY LIBRARY:
```

indicates that the operator wishes to produce a cross-reference of source file S.SA100 on unit 205.

4.2.2 The Copy Library Prompt

Once the source file has been specified the copy library prompt is displayed. It requests the library's file-id and unit-id. If you do not explicitly key a prefix when you input the file-id, $XREF will assume that the library is a conventionally named source file and append the S. prefix by default. For example:

```
$54 COPY LIBRARY:SL UNIT:205
```

causes file S.SL on unit 205 to be used as the copy library.

When you have specified a copy library, the prompt is redisplayed, so that you can specify further copy libraries. Note that a book is always copied from the first library specified in which it exists, so that, for example, if the same book were in both the first and second copy libraries, it would be copied from the first library.

The reply <CR> to the copy library prompt signifies that you have finished specifying copy libraries. If you reply <CR> to the first copy library prompt this means that no copy library is needed, and COPY statements found in the source will be flagged as errors.

If you reply <CR> to the unit prompt then the file is assumed to be on the same unit as the previous copy library specified. However, if you reply <CR> to the very first unit prompt, the first copy library is considered to be the same unit as the source file itself.

4.2.3 The Listing Unit Prompt

Once any copy libraries required have been specified, the listing unit prompt is displayed:

```
$54 LISTING UNIT:205
$54 X-REFERENCE OPTION:
```

If the suffix of the source file is SA100 then the above example will cause the listing to be written to file X.SA100 on unit 101. If the listing unit is a direct access device rather than a printer then the file will be allocated the largest available space on the volume. Any unused space will be released at the end of the cross-reference.

If <CR> is keyed in response to the prompt the file will be placed on unit $PR. You should note that when the listing is output to a real printer, the print file is only opened when the utility is ready to produce the listing at the beginning of its second pass. If you have forgotten to turn the printer on and it is a very simple device unable to provide Global System Manager with status information, this may cause the cross-reference to mysteriously "hang" after a few minutes. The remedy is to follow the instructions laid down in your Global Operating Manual, and always switch any printers on when you bootstrap your system.

4.2.4 The Option Prompt

When the listing unit has been specified the option prompt is displayed:

```
$54 X-REFERENCE OPTION:
```

You may reply with any single valid cross-reference option or <CR>. If you specify a cross-reference option the option prompt is redisplayed so that you can input further options. The reply <CR> indicates that you have finished specifying options, and that the cross-reference processing is to commence. For example, the following dialogue indicates that references are to be prefixed by the first two characters of the section name, and that the first 31 characters of each symbol are significant:

```
$54 X-REFERENCE OPTION:SN
$54 X-REFERENCE OPTION:LN
$54 X-REFERENCE OPTION:<CR>
$54 SOURCE PASS
```

The available cross-reference options are described in detail in Appendix B of the Global Cobol Language Manual, and are summarised in table 4.2.

If an option has been specified in an OPT statement appearing in the source file it can be overridden by respecifying it in response to the option prompt. If, in response to the option prompt, an option is specified twice, or conflicting options are specified, then the last one to be specified is the one which is used.

4.2.5 Source Program Pass

Once any options required have been input the message:

```
$54 SOURCE PASS
```

is displayed, and the initial phase of the cross-reference, the scanning of the source program, begins.

OPTION	**EFFECT**
LN/**NLN**	Use long (31 characters) names.
PL=nnn	Page length (default $$PAGE).

SN/**NSN**	X-reference by section-id.
TE/NTE	Terminate job management if errors are detected.
TW/**NTW**	Terminate job management if warnings are detected.
XR/**NXR**	X-reference unreferenced items in copy books.

Table 4.2 - Summary of X-Reference Options (Defaults Underlined)

A program to be cross-referenced should normally be free from compilation errors. If, however, the program does contain syntax errors then the erroneous lines may not be analysed by $XREF and will be displayed on the console, together with the corresponding compiler error message, as described in Appendix G of the Global Cobol Language Manual. References contained in such erroneous lines may not be included in the cross-reference listing produced.

Note that the validation done by $XREF is much less rigorous than the checking done by $COBOL, and should not be used as a substitute.

Once the source program has been scanned the message:

```
$54 END OF SOURCE PASS
```

is displayed and the references are then sorted into alphabetical order and printed.

4.2.6 Summary and Completion Messages

When $XREF completes normally the files are closed and the following summary messages are output:

```
$54 NUMBER OF ERRORS eeee
$54 NUMBER OF WARNINGS wwww
```

where the quantities eeee and wwww are the decimal numbers of syntax errors and warnings reported.

$XREF then displays its final message:

```
$54 X-REFERENCE COMPLETED
```

and then returns control to the monitor.

4.2.7 Respecifying the Listing File Unit

There are two cases in which you may be asked to change the listing file unit, indicated by the message:

```
$54 PRINTER IN USE
```

or:

```
$54 INSUFFICIENT SPACE ON LISTING FILE
```

preceding the prompt:

```
$54 SPECIFY NEW LISTING UNIT:
```

The first message can only occur if you are attempting to write to a real printer in a multi-user environment. (Not recommended - you should use a spool unit.) The second message means that the listing file has become full and there is no room to write any more output.

You should reply to the prompt by specifying the unit-id of the new device to which the listing file is to be written. You can specify the same device if you believe the printer is no longer in use, or, in the case of the second message, you are willing to let the later part of the listing overwrite the earlier part. The following special replies are also valid:

<CR> write the remainder of the listing to the logical printer assigned to $PR;
<CTRL A> suppress further printing of the listing and continue;
<ESCAPE> abandon the job and return to the monitor.

4.2.8 Example

You require to cross-reference source program S.SA100 on unit 205 to produce a cross-reference listing X.SA100 on the logical printer unit assigned to $PR. Two copy libraries are to be used, S.SL on unit $C and a special copy library S.SL100 on unit 205, which contains new versions of some of the books in S.SL, and hence must be specified first so that the new versions are used. The following dialogue takes place:

```
GSM READY:$XREF
$54 SOURCE:SA100 UNIT:205
$54 COPY LIBRARY:SL100 UNIT:<CR>
$54 COPY LIBRARY:SL UNIT:$C
$54 COPY LIBRARY:<CR>
$54 X-REFERENCE OPTION:<CR>
$54 SOURCE PASS
(the source pass now takes place)
$54 END OF SOURCE PASS
(there is a pause whilst information is sorted. Then
the cross-reference listing is produced.)
$54 NUMBER OF ERRORS 0
$54 NUMBER OF WARNINGS 0
$54 X-REFERENCE COMPLETED
GSM READY:
```

5. Library Maintenance Using $LIB

You can use the librarian, $LIB, to inspect, create or update program or compilation library files, each of which may contain up to 100 executable programs or subroutines. There are special features to allow you to set up dispersed libraries with files on a number of exchangeable volumes, and to run the command under job management.

You should note that $LIB is not used to maintain Global Cobol copy libraries. These are merely specially structured source files and are therefore updated by $EDIT.

INSTRUCTION	GROUP AND FUNCTION
	INSPECT AND EXTRACT OPERATIONS (5.2)
LIS	List the index of a selected library (see note-1)
PRI	Print the index of a selected library (see note-1)
EXT	Extract a member from a selected library (see note-1)
EXD	Extract and delete a member from a selected library
	INCLUDING AND COMPARING MEMBERS (5.3)
MER	Merge one or more selected members into target library
COP	Copy one or more selected members into target library
COM	Compare one or more selected members with originals
	RENAME AND DELETE OPERATIONS (5.4)
CHA	Change target library-id or title
REN	Rename or retitle target library member
DEL	Delete one or selected members from the target library
	SPACE SAVING OPERATIONS (5.5)
NSD	Remove symbolic debug records from target library
TRU	Truncate target library, leaving specified free space
	DISPERSED PROGRAM LIBRARY SUPPORT (5.6)
OFF	Introduce stubs for selected offline members
LNK	Introduce stubs for selected online members
DLN	Delete stubs for selected offline members
	JOB MANAGEMENT SUPPORT (5.7)
I	Mount named input volume

Note-1: The inspect, print and extract operations are the only instructions which do not require a target library to be established.

Table 5.1 - Librarian Instructions

5.1 Introduction

$LIB can process two types of libraries: compilation libraries and program libraries. Each such library is held as an individual file, containing up to 100 **members**. The file-id of a library (sometimes referred to as the **library-id**) must begin with the prefix P. to identify a program library, or C. in the case of a compilation library. The members of a compilation library are either subroutines created by the Global Cobol compiler, $COBOL, or maps produced by the Screen Formatter, $FORM. A program library contains main programs or overlays created by $LINK, the linkage editor, or relocatable programs produced by $RELOC, as described in the Global Cobol Assembler Interface Manual. Members from the two types of library cannot be mixed.

Each member contained in a library is uniquely identified by means of its **member-id**. The member-id of a subroutine is the program name defined in the PROGRAM statement that begins its source listing. The member-id of a map is the map-id specified when $FORM was used to create it. The member-id of a main program, overlay or executable program is initially the file-id of the individual program file which contained the member when it was first included in the library. However, the librarian's REN instruction can be used to rename program library members, although the member-ids of compilation library members cannot be changed.

If you require to create or amend a library, you must first of all establish it as the current **target library.** Then you can use the instructions listed in Table 5.1 to include new members or update old ones. $LIB can only operate on one target library at a time, but the dialogue is structured in such a way that you can process any number of targets, one after another, during a single run.

When you amend a library file it is updated in place. Spare space must be reserved when the library is created to allow for new members or extensions to old ones. $LIB keeps track of any "holes" in the library due to deletions and, when updating is complete, re-organises it so that all the free space is collected together at the end of the file.

The remainder of this introduction describes the way in which the librarian dialogue is structured, and shows in outline how you inspect, create or amend a library. It then describes how and when re-organisation takes place, how duplicate globals are reported, and the way in which errors are handled. The remaining sections of the chapter define the groups of related librarian instructions listed in Table 5.1.

5.1.1 General Dialogue Structure

When you run $LIB it begins by asking you to identify a target library:

```
GSM READY:$LIB
$95 TARGET LIBRARY:
```

If you only need to list or print a library index you need not specify a target. However, if you wish to create a new library, or amend an existing one, you must establish it as the target library by keying its file-id and unit-id (together with other information described later). In either case, eventually the library maintenance prompt will appear:

```
$95 LIBRARY MAINTENANCE
:
```

You may then key any appropriate instruction from the list in Table 5.1. The instruction you select will continue with its own dialogue, and eventually redisplay the library maintenance prompt so that you can supply a further instruction, or terminate the process by keying END. When you do so the target library prompt will re-appear, so that you can operate on another library, or return to the monitor by keying <ESCAPE>. Any dialogue with the librarian therefore assumes the following general form:

```
GSM READY:$LIB
$95 TARGET LIBRARY:
(Identify first target library, if required)
$95 LIBRARY MAINTENANCE
:first instruction
.........
......... (operate on first target library)
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:
(Identify next target library, if required)
$95 LIBRARY MAINTENANCE
:first instruction
.........
......... (operate on next target library)
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY
.........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

Whenever you have established a target library, you must ensure it remains online until the next target library prompt appears following your keying of END. This is because the END processing is responsible for rewriting the updated library directory, and possibly compacting the members so that all the free space is collected together at the end of the file. If this is not done, the library will be invalid, and will be rejected when it subsequently comes to be used by the linker or loader, or indeed the librarian itself.

5.1.2 Inspecting an Existing Library

If you only wish to inspect an existing library, or extract a member from it, there is no need to identify a target library, so you simply key <CR> to its prompt:

```
$95 TARGET LIBRARY:<CR>
$95 LIBRARY MAINTENANCE
:
(Only LIS, PRI and EXT instructions can be used)
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:
```

If you attempt to use an instruction which requires a target library to be established your request will not be honoured. Instead, the message:

```
NO TARGET LIBRARY
```

will appear, followed by the library maintenance prompt, so that you can continue with an appropriate instruction.

5.1.3 Creating a New Library

To create a new library, you must key its library-id and unit-id to the target library prompt. Then you key Y to the subsequent confirmation prompt, and supply size and title. For example:

```
$95 TARGET LIBRARY:P.SA UNIT:205
$95 NEW?:Y SIZE:100K
$95 TITLE:SALES LEDGER SYSTEM
```

```
$95 LIBRARY MAINTENANCE
:
(MER or COP instructions to include the initial members, and
any other instruction, such as TRU, where appropriate)
:END
$95 TARGET LIBRARY:
```

If you key N, <CR> (or any single character apart from Y) to the NEW?: prompt, the creation operation will be abandoned, and the target library prompt will re-appear. This allows you to take corrective action if you really meant to update an existing library, but miskeyed its name or unit.

In response to the SIZE: prompt you may key: a number of bytes; a number followed by the letter K, indicating that so many kilobytes of storage are to be allocated; or simply 0, in which case the maximum amount of contiguous unused space will be made available. You do not have to be particularly precise in your size request since once the library is built you normally use the TRU instruction, described in 5.5.2, to truncate the new library and release any unwanted storage for subsequent re-allocation. TRU allows you to specify an amount of free space to remain in the library so that you can update it in place if you need to add or amend members later.

The title you specify can be up to 30 characters in length. It is displayed on the console when a program library is attached, and, in the case of a compilation library, appears on the map listing when it is used in a linkage edit.

5.1.4 Amending an Existing Library

To amend an existing library you must key its file-id and unit-id to the target library prompt, and then key Y, <CR> (or any single character apart from N) to the subsequent process prompt to confirm that the library it identifies is indeed the one you require to modify. For example:

```
$95 TARGET LIBRARY:P.SA UNIT:205
$95 PROCESS SALES LEDGER SYSTEM OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:
(MER or COP instructions to add or update members and any
other instruction, such as REN or DEL where appropriate)
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:
```

If the library title or date of last modification, which appear as part of the process prompt, are not as expected, you should respond by keying N. Then a new target library prompt will appear to allow you to respecify the library you require to update.

5.1.5 To Quit

To quit and return control to the monitor you must type END to the library maintenance prompt and key <ESCAPE> to the subsequent target library prompt:

```
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

When a target library has been created or amended you must perform the END processing to update its directory. In this case <ESCAPE> is treated in exactly the same way as <CR>. If keyed by mistake to the library maintenance prompt itself the warning:

```
$95 KEY END TO REWRITE THE TARGET LIBRARY DIRECTORY
```

appears, followed by the library maintenance prompt itself.

5.1.6 Library Re-organisation

If you delete one or more members from a target library, $LIB will re-organise it so that all the spare space is held as a contiguous area at the end of the file. For performance reasons re-organisation does not take place after every deletion, but only occurs when necessary: before adding a member if the free storage is too fragmented to contain it; before processing instructions, such as NSD and TRU, which cannot operate satisfactorily if the library contains "holes"; or as a part of END processing, so that the library is left in a fully compacted form. The message:

```
$95 RE-ORGANISING LIBRARY
```

appears, and the normal dialogue is temporarily interrupted whilst the file is scanned and the members are moved.

5.1.7 Global Symbol Checking

When you have finished creating or amending a compilation library, the END processing checks whether the library contains multiple definitions of the same global symbol, since unplanned duplicates can cause problems at linkage edit time. Whenever a duplicate definition is found $LIB displays a warning message of the form:

$95 WARNING - GLOBAL *symbol* DEFINED IN BOTH *member1* AND *member2*

For example:

```
$95 WARNING - GLOBAL MLR DEFINED IN BOTH SA100 AND SA204
```

Such messages immediately precede the new target library prompt. Each identifies the symbol involved, the first routine (in alphabetical order) in which it is defined, and the second or subsequent routine in which a duplicate definition appears.

If more than 250 global symbols are defined by the members of a library the warning message:

```
$95 WARNING - MORE THAN 250 GLOBALS IN LIBRARY
```

appears. This indicates that a complete index record cannot be built, and, in consequence, subsequent attempts to linkage edit programs using the library may fail. Therefore, in the unlikely event of this limit being reached, you should immediately create two or more smaller library files from the affected one.

5.1.8 Some Common Errors during Library Processing

If you have established a program library as a target, you cannot process compilation files or libraries, and similarly, when the target

is a compilation library, you cannot operate on program files or libraries. If you attempt to do so the warning message:

```
WRONG TYPE
```

is displayed, and the library maintenance prompt is redisplayed so that you can correct your error.

If you demount the target library by mistake, before you have keyed END, you will be requested to remount it by a prompt of the form:

$95 REMOUNT *volume-id* ON *unit-address* FOR *library-id*:

where volume-id is the name of the volume occupied by the target library. For example:

```
$95 REMOUNT SAPROG ON 100 FOR P.SA:
```

Mount the volume containing the library, then key <CR>, or any single character, to continue.

The warning messages:

```
FILE NOT FOUND
MEMBER NOT FOUND
$95 INSUFFICIENT SPACE ON TARGET LIBRARY
$95 TARGET LIBRARY FULL
```

may appear at certain stages of the instruction dialogue. Processing of the current instruction is then terminated and the library maintenance prompt is redisplayed so that you can take corrective action. The first warning is output if you name a file which is not present on the specified unit. The second if you attempt to process a member which is not in the designated library. The third will appear if there is insufficient space available when you are updating or adding to a library. The fourth means that the target library already contains its full complement of 100 members and there is no space in its directory to hold the details of a new one.

5.1.9 I/O Error Handling

I/O errors are handled using the normal Global retry mechanism:

$04 ERROR ON *unit-address file-id error-message*
$04 RETRY?:

If you decide the error is irrecoverable and key N to the retry prompt, then, if only an input file was affected, $LIB will "clean up" to prevent a partial, inconsistent member from being added to the target library, and a new library maintenance prompt will appear so that you can continue with other work. You would normally use the LIS or PRI instruction to obtain an up-to-date index of the target library before continuing.

If an irrecoverable I/O error affects the target library itself, the librarian will attempt to write back a consistent index so that the damaged library can be used as input to $LIB itself, even though it cannot be submitted to the linker or librarian. Possibly you will be able to copy all but one affected member into a new library. When $LIB

is able to recover the library in this limited way it outputs the message:

```
$95 END FORCED BY TARGET LIBRARY I/O ERROR
```

This is immediately followed by another target library prompt, so that you can set about creating the new library.

If $LIB is unable even to write back the index, the command is terminated with STOP 9501 and the monitor's ready prompt appears. In this case the error is catastrophic, and you must recover the affected target library from your own backup copy.

Figure 5.2a - The Last Page of a Global System Manager Command Library

Figure 5.2b - An Intermediate Page of the Global System Manager System Library

5.2 Inspect and Extract Operations

Three of the four instructions described in this chapter are unique inasmuch as they are the only ones that do not require a target library to be established in order to operate successfully. LIS and PRI allow you to display or print any library index, and EXT enables you to create an individual program file from a specified member of a selected library. EXD, which does require a target library, behaves as the EXT instruction but also deletes the selected member from the library.

5.2.1 LIS - List the Index of a Selected Library

You can use the LIS instruction to display the library index page by page on the screen. You must specify the library-id together with the unit it occupies:

```
$95 LIBRARY MAINTENANCE
:LIS LIBRARY:library-id UNIT:unit-id
............
............ (listing of specified library)
............
$95 LIBRARY MAINTENANCE
:
```

For example, if you wanted to display the index of library P.SA from unit 205 you would key:

```
:LIS LIBRARY:P.SA UNIT:205
```

and the first page would appear.

If the LIS instruction requires to output more information than can be contained on a single screen display the prompt:

```
$95 NEXT PAGE?:
```

is output on the base line. You must key Y, <CR> (or any single character apart from N) to obtain the next page of output. If you reply N no more of the index will be displayed, and the library maintenance prompt will re-appear.

The screens shown in Figures 5.2a and 5.2b show selected pages from P.$CMLBO, the Global System Manager command library (a program library) and C.$APF, a system library (a compilation library). The format of the display is that of a header, appearing on every page, an information line for each member contained in the library, and trailer information, only output on the very last page. The information lines are arranged in alphabetical order of member-id.

The **header** contains the library-id, the title and today's date.

Each **information line** consists of the member-id, followed by its creation date, type, size and title. The date indicates when a subroutine was last compiled, a map was created or updated, and an executable program last linked or relocated. The size is the number of bytes of file space occupied within the library, not the amount of main storage the member requires. Table 5.2.1 defines the codes that may appear in the TYPE column:

TYPE	MEANING
AS	Absolute assembler program created by $NF.
CO	Compilation file (i.e. subroutine) created by the Global Cobol compiler.
MA	Map created by the Screen Formatter, $FORM.
MC	Global Cobol program or command, created by $LINK.
PI	Position independent program created by $NF.
Rn	Relocatable program, address form n, created by $RELOC.
ST	Stub locating an offline member, created by the librarian's OFF instruction described in 5.6.

Table 5.2.1 - Types of Library Member

The trailer information at the very end of the listing indicates the date at which the library was last updated, its size, the number of members it currently contains, and the amount of free space available for new members.

5.2.2 PRI - Print the Index of a Selected Library

You can use the PRI instruction to cause a specified library index to be printed on the unit assigned to $PR:

```
$95 LIBRARY MAINTENANCE
:PRI LIBRARY:library-id UNIT:unit-id
(The specified library index is now printed)
$95 LIBRARY MAINTENANCE
:
```

For example, if you wanted to print the index of library P.SA from unit 205 you would key

```
:PRI LIBRARY:P.SA UNIT:205
```

and the index would be printed, then another library maintenance prompt would appear.

When $PR is assigned to a direct access volume, rather than a printer, the PRI instruction writes print records to the file D.$LIB, extending the file if it already exists. This allows you to accumulate

information concerning a number of volume directories on a single direct access file when no printer is immediately available.

5.2.3 EXT - Extract a Member from a Selected Library

The EXT instruction enables you to create an individual program file from a specified member of a selected library. The library is read-only as far as EXT is concerned. Therefore, despite the name, the member is not actually physically extracted, but only a copy of it is made. You must specify the library and its unit, together with the member-id and the name and unit for the new file to be created from it:

```
$95 LIBRARY MAINTENANCE
:EXT LIBRARY:library-id UNIT:unit-id
$95 EXTRACT:member-id TO:file-id UNIT:unit-id EXTRACTED
$95 LIBRARY MAINTENANCE
:
```

5.2.4 Operating Notes

Although the LIS, PRI and EXT instructions can function without a target library, they also operate when one has been established. In this case, to process the target library itself, you need only key <CR> to the LIBRARY: prompt. For example, supposing you intend to update program library P.SA. You might decide to list its contents prior to including new members with COP and MER instructions. You might start as follows:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:205
$95 PROCESS SALES LEDGER SUITE OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:LIS LIBRARY:<CR>
...........
...........            (first page of P.SA's index)
...........
$95 NEXT PAGE?:<CR>
...........
...........            (second, and last page of index)
...........
$95 LIBRARY MAINTENANCE
:
```

If you now decided to examine library P.NEW on 205 which you intend to merge with P.SA, you would continue:

```
$95 LIBRARY MAINTENANCE
:LIS LIBRARY:P.NEW UNIT:205
............
............           (first page of P.NEW's index)
............
$95 NEXT PAGE?:
etc, etc.
```

5.2.5 EXD - Extract and Delete a Member from a Selected Library

The EXD instruction enables you to create an individual program file from a specified member of a selected library deleting the specified member. You must specify the member-id and the name and unit of the new file to be created from it:

```
$95 LIBRARY MAINTENANCE
:EXD
```

```
$95 EXTRACT AND DELETE:member-id TO:file-id UNIT:unit-id
EXTRACTED DELETED
$95 LIBRARY MAINTENANCE
:
```

PROMPT	RESPONSE	NORMAL ACTION
FILE:	file-id	If the file consists of an individual map, subroutine or program, process it. If the file is a library itself, respond with the MEMBER: prompt to see which members of that library you require to process.
	<CR>	Respond with file-id?: selection prompts for all appropriate libraries and files on the specified unit, so you can process them if you wish.
	<CTRL A>	Terminate the instruction.
	<CTRL B>	Process every appropriate member on the unit.
file-id?:	Y	If the file consists of an individual map, subroutine or program, process it. If the file is a library itself, respond with the MEMBER: prompt to see which members of that library you require to process.
	<CR>	Skip this file, and proceed to the next appropriate one on the specified unit.
	<CTRL A>	Do not process this file. Terminate the instruction.
	<CTRL B>	Process the member or members this file contains, and every member from each appropriate file remaining on the unit.
MEMBER:	member-id	Process the specified member.
	<CR>	Respond with member-id?: selection prompts for all members in this library, so that you can process them if you wish.
	<CTRL A>	Process no members from this library.
	<CTRL B>	Process all the members from this library.
member-id?:	Y	Process this member.
	N, <CR>	Skip this member and proceed to the next.
	<CTRL A>	Do not process this member, nor any more from the current library.
	<CTRL B>	Process this member and every other one remaining in the current library.

Table 5.3 - Responding to File, Member and Selection Prompts

5.3 Including and Comparing Members

The instructions described in this section are employed in creating or amending a target library. You can include new members using COP or MER; replace existing ones with COP; and check that members have been included correctly by comparing them with the originals using COM.

5.3.1 COP - Copy Member(s) into Target Library

The COP instruction allows you to copy one or more selected members into the target library. You first of all specify the input unit containing the new members, which are read-only as far as the copy operation is concerned. Then in the simplest case, when you just wish to include a member held as an individual file, you key the file-id to

the subsequent prompt. For example, to copy program OURSORT on 205 into target library P.SA:

```
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:205
$95 FILE:OURSORT INCLUDED
$95 LIBRARY MAINTENANCE
:
```

If you supply a library-id in response to the FILE: prompt a subsequent MEMBER: prompt asks you to specify which member or members you require. For example, to include $SORT from library P.FREE on 209:

```
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:209
$95 FILE:P.FREE MEMBER:$SORT INCLUDED
$95 LIBRARY MAINTENANCE
:
```

These are the two simplest ways in which to use the FILE: and MEMBER: prompts. In fact, as Table 5.3 shows you may also key the special responses <CR>, <CTRL A> and <CTRL B> to serve as selection codes, with meanings similar to those used in $F. For example, a single COP instruction might be used to copy program SALES, the entire contents of P.FREE, and members SA200 and SA201 of library P.NEW, into target library P.SA on 205. All these are on unit 209 together with program file SATEST and program library P.SAOLD which are not required:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:205
$95 PROCESS SALES LEDGER SYSTEM OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:209
$95 FILE:<CR>
SALES ?:Y
SATEST ?:N
P.FREE ?:Y MEMBER:<CTRL B>
      $A          INCLUDED
      $C          INCLUDED
      .........
      .........
      .........
      $SORT       INCLUDED
P.NEW ?:Y MEMBER:<CR>
      SA100   ?:N
      SA200   ?:Y
      SA201   ?:Y
      P.SAOLD ?:<CTRL A>
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

The copy operation may be used to replace an existing member with an updated version. Both the old and the new one must have the same member-id. Then, whenever $LIB detects that you are attempting to include a member whose member-id matches one already present on the target library, it outputs the warning:

```
MEMBER ALREADY EXISTS - DELETE?
```

If you reply Y the existing member will be replaced by the new version. If you key N, <CR>, (or any single character apart from Y) the target library will remain undisturbed and the new member will be

ignored: if you are including a number of members the copy process will then continue with the next satisfying the selection criterion.

5.3.2 MER - Merge Member(s) into Target Library

The MER instruction enables you to include one or more selected members into the target library, rather like a copy operation. The difference is that a merge cannot be used to update an existing member, since members already present on the target library take precedence over new ones with the same member-id. When a match is detected $LIB simply outputs the warning:

```
MEMBER ALREADY EXISTS
```

and ignores the later member: if you are including a number of members the merge process will then continue with the next to satisfy the selection processing.

Merge dialogue is exactly the same as that used for a copy operation, except that the MER instruction is used in place of COP. Simple examples such as:

```
$95 LIBRARY MAINTENANCE
:MER FROM UNIT:100
$95 FILE:OURSORT INCLUDED
$95 LIBRARY MAINTENANCE
```

or:

```
$95 LIBRARY MAINTENANCE
:MER FROM UNIT:209
$95 FILE P.FREE MEMBER:$SORT INCLUDED
$95 LIBRARY MAINTENANCE
```

are not particular realistic, since in these cases it would be more flexible to use the COP instruction, in case you needed to update an old version of OURSORT or $SORT. A more relevant use of the merge operation is provided by the following example.

Suppose diskette SRBASE contains subroutines required by a development project, and represents the status of the routines when they were throughly unit tested, say, a month previously. In the interim period, corrections have been made to some of these routines, and new ones have been added. The additional members involved are all on SRNEW. Then the following procedure could be used to create an up-to-date subroutine library, C.SR, on SRLIB:

```
GSM READY:$LIB
$95 TARGET LIBRARY:
$95 TARGET LIBRARY:C.SR UNIT:205
$95 NEW:Y SIZE:0
$95 TITLE:COMBINED SUBROUTINE LIBRARY
$95 LIBRARY MAINTENANCE
:
(Mount SRNEW on diskette drive 100 before replying)
:MER FROM UNIT:100
$95 FILE:<CTRL B>
..........
.......... (all members from SRNEW are included, from both
.......... individual files and compilation libraries)
..........
$95 LIBRARY MAINTENANCE
:
(Mount SRBASE on diskette drive 101 before replying)
:MER FROM UNIT:101
```

```
$95 FILE:<CTRL B>
.........
......... (members from SRBASE are included, except when
......... they have already been superseded members
......... previously merged from SRNEW)
.........
$95 LIBRARY MAINTENANCE
:TRU NEW SPARE SPACE:<CR>     (explained in 5.5.2)
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

Of course, a similar effect could have been achieved using a COP instruction, providing SRBASE had been processed before SRNEW. However, whenever an update occurred you would need to reply Y to the member already exists prompt, which would be operationally less convenient. There would be more data transferred, since old members would have been copied prior to deletion. Furthermore because of the deletions a library re-organisation would be necessary which is not needed using the merge technique.

5.3.3 COM - Compare Member(s) with Originals

The COM instruction is provided so that you can compare members previously copied or merged into the target library with their original versions. It is particularly useful when you require to distribute software on an inherently unreliable medium such as diskette. The dialogue is similar to that employed in a copy or merge operation, except that the COM instruction is used.

For example, suppose you decide to update library P.SA1 with program SALES, $SORT from P.FREE, and the entire contents of P.NEW, and you wish to check that the copy process has been performed accurately. This requires just one COP instruction, followed by a similar COM:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA1 UNIT:100
$95 PROCESS V6.0 SALES LEDGER SYSTEM OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:209
$95 FILE:<CR>
SALES   ?:Y INCLUDED
SATEST  ?:N
P.FREE  ?:Y MEMBER:$SORT INCLUDED
P.NEW   ?:Y MEMBER:<CTRL B>
      SA100       INCLUDED
      .........
      .........
      .........
      SA920       INCLUDED
P.SAOLD ?:<CTRL A>
$95 LIBRARY MAINTENANCE
:COM FROM UNIT:209
$95 FILE:<CR>
SALES   ?:Y IDENTICAL
SATEST  ?:N
P.FREE  ?:Y MEMBER:$SORT IDENTICAL
P.NEW   ?:Y MEMBER:<CTRL B>
      SA100       IDENTICAL
      .........
      .........
      .........
      SA920       IDENTICAL
P.SAOLD ?:<CTRL A>
```

```
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

Providing that every byte of the member selected from the input unit, apart from its creation date and title, matches those of the copy on the target library, the confirmation message:

```
IDENTICAL
```

appears, as shown in the example. The warning messages:

```
DISCREPANCY AT nnnn
```

or:

```
NOT FOUND
```

are displayed if either the copies do not tally, or if the member-id of a selected member was not present in the target library. The quantity nnnn is the number of the first byte within the two members not to match, counting from 0.

5.4 Rename and Delete Operations

The instructions described in this section can be used to change the file-id or the title of the target library, and rename, retitle or delete one of its members. Titles can be at most 30 characters in length.

5.4.1 CHA - Change Target Library-id or Title

The CHA instruction allows you to change the library-id or title of the target library. For example, to change P.SA, entitled SALES LEDGER SYSTEM, to P.SAOLD, SALES LEDGER SYSTEM (FEB 88):

```
$95 LIBRARY MAINTENANCE
:CHA P.SA TO:P.SAOLD
OLD TITLE SALES LEDGER SYSTEM
NEW TITLE:SALES LEDGER SYSTEM (FEB 88)
$95 LIBRARY MAINTENANCE
:
```

You may key <CR> in place of the new library-id or new title, in which case the existing library-id or title will remain unchanged.

You may replace selected characters at the beginning of the old title by keying your amendment, terminated by <CTRL A>, as the new one. For example, to change the title of library C.SA from V1.0 SALES LEDGER ROUTINES to V6.0 SALES LEDGER ROUTINES:

```
$95 LIBRARY MAINTENANCE
:CHA C.SA TO:<CR>
OLD TITLE:V1.0 SALES LEDGER ROUTINES
NEW TITLE:V6.0<CTRL A>
$95 LIBRARY MAINTENANCE
```

5.4.2 REN - Rename or Retitle Target Library Member

You can use the REN instruction to change a particular member-id within a target program library, and to alter the title of any type of member. The dialogue is of the general form:

```
$95 LIBRARY MAINTENANCE
```

```
:REN :old-member-id [AS:new-member-id]
OLD TITLE old-title
NEW TITLE:new-title
$95 LIBRARY MAINTENANCE
:
```

The part in square brackets is omitted for a compilation library, because it is only possible to rename subroutines or maps by recompiling using $COBOL, or by creating an updated map using $FORM. For example, to correct a spelling mistake in the title of subroutine SAMLR:

```
$95 LIBRARY MAINTENANCE
:REN :SAMLR
OLD TITLE BANK RATE APPLICATION
NEW TITLE:BANK RATE APPLICATION
$95 LIBRARY MAINTENANCE
:
```

You may key <CR> in place of a new member-id or new title, in which case the existing member-id or title will remain unchanged.

You may replace selected characters at the beginning of the old title by keying your amendment, terminated by <CTRL A>. For example, to adjust a version code by changing the title of program SA100 from V1.0 SALES ACCT ROOT to V6.0 SALES ACCT ROOT:

```
$95 LIBRARY MAINTENANCE
:REN :SA100 AS:<CR>
OLD TITLE V1.0 SALES ACCT ROOT
NEW TITLE:V6.0<CTRL A>
$95 LIBRARY MAINTENANCE
:
```

Note that if you rename a program library member and the new member-id you choose is already present on the target library, the warning prompt:

```
MEMBER ALREADY EXISTS - DELETE?:
```

appears. If you key Y the old member will be deleted and the rename operation will continue as though it had never existed. A reply of N, <CR> (or any single character apart from Y) causes the instruction to be abandoned and the library maintenance prompt to re-appear.

5.4.3 DEL - Delete Member(s) From the Target Library

The DEL instruction enables you to delete one or more selected members from the target library. To delete one particular member, you simply key its member-id in response to the MEMBER: prompt. For example, to remove SA100 from the target library:

```
$95 LIBRARY MAINTENANCE
:DEL MEMBER:SA100 DELETED
$95 LIBRARY MAINTENANCE
:
```

You may also use the special responses <CR>, <CTRL A> and <CTRL B>, to delete a selection of members, using the scheme summarised in the lower half of Table 5.3. For example:

```
$95 LIBRARY MAINTENANCE
```

```
:DEL MEMBER:<CR>
      SA200 ?:N
      SA205 ?:N
      SA300 ?:Y DELETED
      SA310 ?:Y DELETED
      SA400 ?:N
      SA500 ?:Y DELETED
      SA600 ?:<CTRL A>
$95 LIBRARY MAINTENANCE
:
```

Here SA300, SA310 and SA500 are deleted from the target library, which has been displayed member by member for selection, following a <CR> response to the member prompt. The keying of <CTRL A> to the select prompt terminates the process.

5.5 Space Saving Operations

The two instructions described in this section help to limit the amount of direct access storage required by the target library, either by reducing the size of the members themselves (NSD), or by returning unwanted free space to the volume for re-allocation (TRU).

5.5.1 NSD - Remove Symbolic Debug Records

Normally Global Cobol compilation files created by $COBOL contain symbolic debug records which, in turn, become part of any program files produced by linking the compilations. The records do not affect normal program execution in any way, since they are only interrogated by $DEBUG, the symbolic debugging system. However, they may occupy a sizeable proportion of a library, since there will be an 11-byte record for each symbol employed in a subroutine, or for each symbol of each subroutine used in constructing a program. The symbolic debug records can be removed at compile or link time by specifying the NSD option when $COBOL or $LINK is executed. However, it may prove simpler to eliminate them from the target library itself when they are no longer needed by using the NSD instruction:

```
$95 LIBRARY MAINTENANCE
:NSD target-library-id?:Y SYMBOLIC DEBUG RECORDS REMOVED
$95 LIBRARY MAINTENANCE
:
```

Following your keying of NSD the library-id of the target is displayed in a confirmation prompt, to check that you really do wish to remove symbolic debug records from the library in question. The operation will only proceed if you key Y. For example:

```
$95 LIBRARY MAINTENANCE
:NSD P.SA?:Y SYMBOLIC DEBUG RECORDS REMOVED
$95 LIBRARY MAINTENANCE
:
```

5.5.2 TRU - Truncate Target Library

You use the TRU instruction to return any unwanted free space in the target library to the volume it occupies, so that the storage thus released becomes available for re-allocation. The dialogue allows you to specify how much free space you require to remain in the library, once truncation has taken place:

```
$95 LIBRARY MAINTENANCE
:TRU NEW SPARE SPACE:required-free-space TRUNCATED
```

```
$95 LIBRARY MAINTENANCE
:
```

If you never intend to update the library you will require no spare space at all, and in this case you may key <CR> or 0 in response to the NEW SPARE SPACE: prompt. Normally, however, you should leave enough room to hold several extra members, to prevent re-organisation having to take place unnecessarily.

The special response -A will leave sufficient spare space to create stubs in the library for up to 100 members in total. It is intended for use under Job Management by installation software.

The following example shows how a program library might be set up initially:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:205
$95 NEW:Y SIZE:0
$95 LIBRARY MAINTENANCE
:
(COP or MER instructions to include the initial members).
$95 LIBRARY MAINTENANCE
:TRU NEW SPARE SPACE:40K
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

By specifying an initial SIZE request of 0 in the third line of the dialogue, the maximum amount of contiguous space available is allocated to the library initially. The TRU instruction returns any unwanted space to the volume, but keeps 40K bytes spare for subsequent expansion.

Note that if you request more spare space than is actually present on the library the warning message:

```
TOO LARGE
```

will appear and the truncation operation will be abandoned.

5.6 Dispersed Program Library Support

The two instructions described in this section are mainly for use In software installation jobs. They allow large program libraries to be split up into smaller ones and dispersed onto separate disks, but interlinked so that Global System Manager knows where to find all the programs required.

A dispersed program library consists of a number of files, each of which has the same library-id but occupies a different volume. For example, when Global Cobol used to be installed on low-capacity diskettes, the command library, P.$CMLIB, was treated as a dispersed program library. As a result identically named P.$CMLIB files, each containing different commands, occupy the four system volumes SYSRES, SYSDEV, SYSLNK, and SYSMAP. Alternatively, if Global Cobol is now installed onto hard disk, a volume is allocated on the domain called SYSDEV containing P.$CMLIB, which is linked to the P.$CMLBO on SYSRES.

A dispersed program library therefore consists of two or more P. files with the same name occupying different volumes. The volumes may be exchangeable (e.g. diskettes all occupying the same unit address) or they may occupy different addresses (e.g. different volumes on a domain or a library split between diskette and RAM disk).

By using the OFF or LNK instructions you can add small dummy members, known as **stubs**, to each part of a dispersed library. Each stub contains the volume name of the disk containing the program with that name and also the unit address if it is on a different unit. If you list the library (as shown in screen 5.2a) the stub will appear as:

```
OFFLINE - LOOK ON SYSRES
```

or:

```
OFFLINE - LOOK ON SYSRES - 201
```

If the operator attempts to run a program represented by a stub, Global System Manager reassigns $P or $CP as necessary and makes sure the correct volume is mounted, giving a message such as:

```
PLEASE MOUNT SYSRES ON LEFT HAND DRIVE DISKETTE DRIVE – 100 AND KEY <CR>:
```

if it is not.

Note that if a stub had not been present for the volume, Global System Manager would have simply output the program prompt which it uses whenever a required program or command is missing. For example:

```
$01 PROGRAM SALES IN LIBRARY P.SL REQUIRED ON 100:
```

The operator would now have to guess which of the other volumes actually contained the program. If the wrong one was mounted, the program prompt would simply re-appear. There might be a frustrating search until the right volume was obtained.

You can use the OFF instruction to interconnect your own dispersed libraries if you are developing systems which exceed the capacity of the available direct access volumes. Each separate file of such a library must be created or amended as an individual target library. The members whose code the file actually contains are included using copy or merge operations in the normal way. Then stubs for **every one** of the offline members, residing on the other volumes occupied by the dispersed program library, must be introduced using the OFF instruction.

5.6.1 OFF - Introduce Stubs for Offline Members

The OFF instruction allows you to introduce one or more selected stubs into the target library. Operationally the dialogue is very similar to that of the MER (merge) instruction, except that OFF is keyed in place of MER, and there is an additional prompt to determine the volume to be occupied by the offline members

For example, to introduce a stub into the target library to inform Global System Manager that program SALES will be offline on SAPRG1:

```
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:100 VOLUME:SAPRG1
$95 FILE:SALES OFFLINED
$95 LIBRARY MAINTENANCE
```

```
:
```

Note that SALES may not actually be on SAPRG1 at the time of using the OFF instruction: you may move it to the correct volume at some later stage before you come to use the target library. Similarly, in the following example, stubs are introduced into the target library, P.SA, for all the members currently occupying P.NEW. Eventually, if the dispersed library is correctly constructed, all those members will either have to be part of another P.SA library on SAPRG1, or will have to appear as individual program files on that volume:

```
$95 TARGET LIBRARY:P.SA
$95 PROCESS SALES LEDGER SYSTEM OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:100 VOLUME:SAPRG1
$95 FILE:P.NEW MEMBER:<CTRL B>
      SA100 OFFLINED
      SA200 OFFLINED
      .........
      .........
SA950 OFFLINED
$95 LIBRARY MAINTENANCE
:
```

Note that if the members whose stubs you are including are already occupying the destination volume, you can simply key <CR> to the VOLUME: prompt. The librarian will then obtain the volume-id to use in the stubs from the label of the input volume itself.

A stub introduced by the OFF instruction will replace one with the same member-id which is already present on the target library. However, a stub will never replace the code of an existing member. If this is attempted the warning message:

```
MEMBER ALREADY EXISTS
```

appears in place of the confirmation OFFLINED which is normally displayed to show that the proper interconnection has been made.

Sometimes the members selected for processing by the OFF instruction may themselves be represented by stubs introduced by a previous use of OFF. In this case the volume information in these secondary stubs is copied unchanged to the target library, and displayed as the volume-id in the confirmation message, providing of course, that the corresponding real member is not already present on the target library. Example 2 below shows how secondary stubs may be used.

5.6.2 LNK - Link Together On-line Libraries

The LNK instruction is similar to OFF, but is intended for use when the two libraries are usually on-line, but occupy different unit addresses. You might use LNK to connect together libraries for two separate but related products installed on two different volumes of a domain (or indeed, on two volumes on separate domains). You can also use LNK if one library is resident on RAM disk and the other is on diskette or hard disk. This is usually necessary because most RAM disks are too small to hold the complete library, and so you can only put the frequently used overlays in the RAM disk.

The dialogue is the same as for the OFF instruction, except that there is an additional prompt to determine the unit to be occupied by the

programs not in the library. For example, to introduce a stub to indicate that file SALES, currently on unit 100, is to be off-line on SAPRG1 on unit 109:

```
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:100 VOLUME:SAPRG1 ON UNIT:109
$95 FILE:SALES OFFLINED
$95 LIBRARY MAINTENANCE
```

If the program is already on the correct volume and unit you can simply key <CR> to the "volume" and "on unit " prompts. If the file name you supply is a library, you will be prompted for each member as for the OFF instruction.

A stub introduced by LNK will replace one with the same member name which is already present in the library. However, a stub never replaces a program that is already in the library. If you link to a library that already contains stubs, the information in these stubs will be copied to stubs in the target library.

5.6.3 Operating Notes

Bear in mind that as each file of a dispersed library is built, **every member** belonging to the total library must be held as an individual file on the same volume, or have a stub created for it by the OFF instruction. Failure to obey this rule will mean that either some members, or some stubs, will be missing - a situation which will normally lead to an unexpected program prompt appearing.

If a stub contains wrong information Global System Manager may issue a mount prompt that the operator is unable to honour, or a program prompt if the volume exists but does not contain the member in question. If Global System Manager detects a stub for a member, mounts the indicated volume, and then finds only another stub, it terminates the load operation with a stop code: a stub must always identify the volume upon which the code of its member resides.

Although you can use DEL to delete a stub, the other instructions that can affect individual members (COP, MER, COM and REN) do not operate on stubs. In consequence members represented only by stubs are omitted from selections obtained by keying <CR> or <CTRL B> to the MEMBER: prompt. If you request to copy or merge a specific member, but there is only a stub for it, the instruction will be terminated with the NOT FOUND warning message. The same will occur if you attempt to compare a stub with a real member, or rename a member which is only represented by a stub within the target library.

5.6.4 Example 1 - Interconnecting Two Library Files

Suppose $LIB has already been used to create two library files named P.SA, which occupy diskettes named SAPRG1 and SAPRG2, because there is not enough room to hold the entire library on a single volume. The two files have been created by copying or merging members in the normal way. They are quite suitable for development working where program prompts are acceptable. Now, however, it is decided to interconnect them using the OFF instruction so that mount prompts appear whenever a volume change is needed. Both library files have sufficient spare space (46 bytes per stub) to allow the update to take place. SAPRG1 is on 100, and SAPRG2 on 101:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:100
$95 PROCESS SALES LEDGER PART ONE OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:101 ON VOLUME:<CR> (SAPRG2 will be used)
$95 FILE:P.SA MEMBER:<CTRL B>
      ........
      ........ (Stubs added to P.SA on SAPRG1 for offline members on SAPRG2)
      ........
$90 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:P.SA UNIT:101
$95 PROCESS SALES LEDGER PART TWO OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:100 ON VOLUME:<CR> (SAPRG1 will be used)
$95 FILE:P.SA MEMBER:<CTRL B>
      ........
      ........ (Stubs added to P.SA on SAPRG2 for offline members on SAPRG1)
      ........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

5.6.5 Example 2 - Interconnecting a Third File

Some little time after the previous example, more sales ledger programs have been developed, and the new components occupy library P.SA on SAPRG3. It is decided to interconnect the new library file with the existing two. Initially SAPRG1 is on 100, and SAPRG3 on 101:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:100
$95 PROCESS SALES LEDGER PART ONE OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:101 ON VOLUME:<CR> (SAPRG3 will be used)
$95 FILE:P.SA MEMBER:<CTRL B>
      ........
      ........ (Stubs added to P.SA on SAPRG1 for offline members on SAPRG3)
      ........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:
(Replace SAPRG1 by SAPRG2 before replying)
$95 TARGET LIBRARY:P.SA UNIT:100
$95 PROCESS SALES LEDGER PART TWO OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:101 ON VOLUME:<CR> (SAPRG3 will be used)
$95 FILE:P.SA MEMBER:<CTRL B>
      ........
      ........ (Stubs added to P.SA on SAPRG2 for offline members on SAPRG3)
      ........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:P.SA UNIT:101
$95 PROCESS SALES LEDGER PART THREE OF 25/03/88?:Y
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:100 ON VOLUME:<CR> (SAPRG2 will be used)
$95 FILE:P.SA MEMBER:<CTRL B>
      ........
      ........ (Stubs added to P.SA on SAPRG3 for offline
      ........ members on SAPRG2 and, because of secondary
      ........ stubs, for members on SAPRG1)
      ........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

This dialogue therefore performs three OFF operations:

> OFF P.SA from SAPRG3, to include the required new stubs in P.SA on SAPRG1;
>
> OFF P.SA from SAPRG3, to include the required new stubs in P.SA on SAPRG2;
>
> OFF P.SA from SAPRG2 to include the stubs needed by the new P.SA library on SAPRG3.

Note that the fourth operation that you might have expected:

> OFF P.SA from SAPRG1 to include the stubs needed by the new P.SA library on SAPRG3;

is in fact unnecessary, because secondary stubs already present in P.SA on SAPRG2 establish the necessary interconnection to SAPRG1 during the third OFF operation.

5.6.6 Example 3 - Installing Part of a Library on RAM Disk

Suppose that a suite of programs has been split by $LIB into a small library of frequently used overlays, P.SA1, and a second library of the remaining components, P.SA. This has been done so that the programs in P.SA1 can be held in RAM disk when installed onto a suitable computer.

The libraries are to be installed on volume SAPROG on unit 100. However, before the programs are run, P.SA2 will be copied to the RAM disk ($$WORK on 109) and renamed as P.SA. In order that the programs will be automatically loaded from the correct unit, the libraries must be linked as follows:

```
GSM READY:$LIB
$95 TARGET LIBRARY:P.SA UNIT:100
$95 PROCESS SALES LEDGER MAIN PART OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:LNK FROM UNIT:<CR> ON VOLUME:$$WORK ON UNIT:109
$95 FILE:P.SA1 MEMBER:<CTRL B>
       ........
       ........    (stubs added to P.SA)
       ........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:P.SA1 UNIT:100
$95 PROCESS SALES LEDGER COMMON OVERLAYS OF 14/02/88?:Y
$95 LIBRARY MAINTENANCE
:LNK FROM UNIT:<CR> ON VOLUME:<CR> ON UNIT:<CR>
$95 FILE:P.SA MEMBER:<CTRL B>
       .........
       .........   (stubs added)
       .........
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:
```

5.6.7 DLN - Delete Stub Links

The DLN instruction allows you delete stubs for a single linked member or a selected group of members according to the volume-id and unit criteria.

The dialogue is as follows:

```
$95 LIBRARY MAINTENANCE
:DLN
$95 UNLINK STUBS VOLUME:volume-id UNIT:unit-id
$95 MEMBER:member-id
```

If you want to delete a single line you must key <CR> to the volume and unit prompts and key the member name to the member prompt. For example, to delete the link member SA100 you must key:

```
$95 LIBRARY MAINTENANCE
:DLN
$95 UNLINK STUBS VOLUME:<CR> UNIT:<CR>
$95 MEMBER:SA100 DELETED
```

To delete all linked members found on a specific volume-id you must key the volume-id to the volume-id prompt, <CR> to the unit-id prompt and <CTRL B> to the member prompt. The dialogue to delete all links to volume SAPRG1 would be as follows:

```
$95 LIBRARY MAINTENANCE
:DLN
$95 UNLINK STUBS VOLUME:SAPRG1 UNIT:<CR>
$95 MEMBER:<CTRL B>
      SA100       DELETED
      SA200       DELETED
      .........
      .........
      SA950       DELETED
```

To delete members with links to a specified unit-id you must key <CR> to the volume prompt, the unit to the unit-id prompt and <CTRL B> to the member prompt. Should you only want to delete members with both a specific unit and volume you must reply to both volume and unit prompts and key <CTRL B> to the member prompt.

5.7 Job Management Support

The features described in this section are intended for use when the librarian runs under job management. They allow you to check that the correct output volume is loaded before processing a target library, and that specific input volumes are mounted when required. There is special support for the COP instruction, so that identical dialogue can be used both to add new members or replace existing ones.

5.7.1 Mounting the Target Library Volume

If you key the special response <CTRL C> to the target library prompt $LIB will ask you for a volume-id and then repeat the prompt. For example:

```
GSM READY:$LIB
$95 TARGET LIBRARY:<CTRL C> VOLUME-ID:SAPRGA
$95 TARGET LIBRARY:
```

Then, when you have identified the target library, $LIB will open it using volume-id checking, and if the volume you have specified is not on the unit you have indicated, it will be requested by a normal mount prompt. For instance, the dialogue introduced above might continue:

```
$95 TARGET LIBRARY:P.SA UNIT:100
PLEASE MOUNT SAPRGA ON 100 - LEFT HAND DISKETTE DRIVE AND KEY <CR>:
```

A mount prompt always appears, even when the rest of the responses supplied to $LIB are obtained from the dialogue table, since it is one of the special ones which bypass job management. The operator must key Y, <CR> (or any single character apart from N) to cause the job to continue.

The mount prompt will be redisplayed, over and over again if necessary, until the correct volume is loaded. However, a reply of N causes the mount operation to be abandoned and, if job management was in control, it will be terminated.

5.7.2 I - Mount Input Volume

The I instruction enables you to control the mounting of a specified input volume required for subsequent library maintenance operations. It prompts you for the volume-id, which it remembers so that it can open the next file specified by an instruction using volume-id checking. This will cause a mount prompt to appear if the volume is not already present on the indicated unit. For example, suppose you wish to copy the file named OURSORT from volume OURDEV on 100 to be the target library on 205. You can use the I instruction to control the mounting of OURDEV as follows:

```
$95 LIBRARY MAINTENANCE
:I VOLUME-ID:OURDEV
$95 LIBRARY MAINTENANCE
:COP UNIT:100
$95 FILE:OURSORT
```

If at this stage OURDEV is not present on unit 100, the mount prompt:

```
PLEASE MOUNT OURDEV ON 100 - LEFT HAND DISKETTE DRIVE AND KEY <CR>:
```

will appear. As explained above, since this is one of the special prompts that bypass job management it will always be displayed on the console, even when the other responses to $LIB are supplied in the dialogue table. The operator must load the requested volume and key Y, <CR> (or any single character apart from N) to continue. A reply of N terminates job management.

5.7.3 The COP Instruction under Job Management

You will recall from 5.3.1 that when the COP instruction is used to replace an existing member on the target library with an update the prompt:

```
MEMBER ALREADY EXISTS - DELETE?:
```

appears, allowing you to choose whether or not to replace the member in question. When $LIB runs under job management a response of Y is automatically supplied to this prompt. This allows you to code exactly the same dialogue, whether you intend to add a new member, or replace an existing one. For example:

```
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:100
$95 FILE:OURSORT
```

will cause OURSORT to either be added to the target library, if it is not already present, or replaced with the new version, if it is.

5.7.4 Example Job Management Dialogue

In the previous section we described how a sequence of COP and OFF instructions could be used to install a dispersed library named P.SA on newly initialised volumes named SAPRG1 and SAPRG2. P.SA is created from libraries P.SAA and P.SAB residing on distribution volumes named SAA and SAB respectively. The following dialogue could be encoded in a job description so that the installation process could be performed under job management. The new volumes are to be created on unit 100, the existing ones read from 101:

```
GSM READY:$LIB
$95 TARGET LIBRARY:<CTRL C> VOLUME-ID:SAPRG1
$95 TARGET LIBRARY:P.SA UNIT:100
$95 NEW?:Y SIZE:0
$95 TITLE:DISPERSED LIBRARY PART 1
$95 LIBRARY MAINTENANCE
:I VOLUME-ID:SAA
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:101
$95 FILE:P.SAA MEMBER:<CTRL B>
$95 LIBRARY MAINTENANCE
:I VOLUME-ID:SAB
$95 LIBRARY MAINTENANCE
:OFF FROM UNIT:101 ON VOLUME:SAPRG2
$95 FILE:P.SAB MEMBER:<CTRL B>
$95 LIBRARY MAINTENANCE
:TRU NEW SPARE SPACE:<CR>
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<CTRL C> VOLUME-ID:SAPRG2
$95 TARGET LIBRARY:P.SA UNIT:100
$95 NEW?:Y SIZE:0
$95 TITLE:DISPERSED LIBRARY PART 2
$95 LIBRARY MAINTENANCE
:COP FROM UNIT:101
$95 FILE:P.SAB MEMBER:<CTRL B>
$95 LIBRARY MAINTENANCE
:I VOLUME-ID:SAA
:OFF FROM UNIT:101 ON VOLUME:SAPRG1
$95 FILE:P.SAA MEMBER:<CTRL B>
$95 LIBRARY MAINTENANCE
:TRU NEW SPARE SPACE:<CR>
$95 LIBRARY MAINTENANCE
:END
$95 TARGET LIBRARY:<ESCAPE>
GSM READY:
```

You may find it instructive to compare this with the original example in 5.6.5 where comments were used to describe the volume mounting requirements and the effect of the various COP and OFF instructions.

6. Linkage Editing Using $LINK

The $LINK command is used to linkage edit selected modules to form a loadable program, which can then be brought into main memory by the operator keying its name in response to the ready prompt, or as the result of a program already in memory executing a LOAD, EXEC or CHAIN statement. The modules involved may be either compilations created by $COBOL or maps produced by $FORM. They may be held as individual files, or may be members of a compilation library.

The linkage edit process is responsible for determining the storage locations that the program will occupy when it is loaded, together with the Global Cobol address of its entry point. Each module is originally created to start at Global Cobol address 0, so $LINK has the problem of modifying all the addresses involved appropriately. In addition the linkage edit must attempt to satisfy all global references to symbols not contained in the current map or compilation by including the module or modules defining them.

Table 6.1 lists the Global Cobol statements which reference or define global symbols. You will note that many of them refer to globals which are defined within members of the system libraries. $LINK normally includes this library in every linkage edit so that the appropriate subroutines for formatted display working, editing, file handling (access methods), display mapping and sorting are included in the resulting program as and when required. In addition if there are any calls on system routines, such as COPY$, CONV$... etc, the appropriate modules from the system libraries will be brought in. There is not enough space in Table 6.1 to show them all, but if you need to know the member or members included by a particular call you can refer to the appendix of the manual describing it, entitled "Included Routines".

You should note, for completeness, that a map produced by $FORM defines its map-id as a global, addressing the first location that its data tables occupy. If the map refers to a user validation routine then it will contain a global reference to the entry name of that routine.

There are normally two outputs from $LINK: the program file, whose name, the **program-id**, is used to identify the program in LOAD, EXEC or CHAIN statements, or when run from the console; and the link map listing, detailing the input files, the modules included in the program, and any error or warning conditions. There is an annotated link map listing of a simple example program in Appendix A.

You should note that a single invocation of $LINK produces just one program file and link map, so you have to use the command repeatedly for an application involving an overlay structure, where the root, together with each of the lower level overlays, each count as separate programs.

In the description which follows we first of all explain how $LINK can be used to create **independent** programs, which contain all the modules they use. Then we describe how you can employ the command to produce **dependent** programs, to optimise the storage requirements of certain overlay structures. A dependent program is able to access some or all

of the modules it requires from separate "information" overlays known to be already resident when the dependent is entered.

COBOL STATEMENTS GENERATING GLOBAL SYMBOLS	**WHERE THE GLOBAL SYMBOL IS DEFINED**
ACCEPT...LINE, DISPLAY...LINE	In member QS$A of C.$MCOB
CALL entry-name [USING...]	In a module where entry-name occurs in an ENTRY Statement, or is a local symbol made into a global definition by the GLOBAL statement
CLEAR	In member GA$A of C.$MCOB
COMMON SECTION name	The section name is defined as a global where the statement is coded
EDIT	In member QL$A of C.$MCOB
ENTRY entry-name [USING...]	The entry-name is defined as a global where the ENTRY statement itself is coded
EXTERNAL SECTION name	In a module containing a COMMON SECTION with the same name
FD...ORGANISATION RELATIVE-SEQUENTIAL	In member AR$B of C.$MCOB if the FD possesses a BLOCK CONTAINS statement, otherwise in AR$A.
FD...ORGANISATION INDEXED-SEQUENTIAL FD...ORGANISATION OR$83 FD...ORGANISATION OR$84 FD...ORGANISATION OR$85 FD...ORGANISATION DMAM	In member AI$A of C.$MCOB In member AT$A of C.$MCOB In member AV$A of C.$APF In member AB$A of C.$APF In member AM$A of C.$APF
GLOBAL symbol	In a compilation where symbol occurs as a program name in the PROGRAM statement, an entry name in the ENTRY statement, the name of a COMMON SECTION, or is locally defined but rendered global by a GLOBAL statement, or in a map where symbol is the map-id
MD...[MAP map-id]	In member SM$A of C.$APF [if the optional MAP clause is present another global reference is generated, to a symbol located at the start of the map named map-id]
PROGRAM program-name	The program-name is defined as a global where the PROGRAM statement itself is coded
SCROLL	In member QB$A of C.$MCOB
SORT, RELEASE, RETURN	In member QO$A of C.$MCOB

Table 6 - Global Cobol Statements Generating Global Symbols

6.1 Linkage Editing an Independent Program

This section describes how you use $LINK to create an independent program, which is to contain all the modules it requires. Such a program may be a single, stand-alone application; the root of an overlay structure; or even a lower level overlay if there is no need

to minimise main storage requirements by including modules common to several overlays only once.

6.1.1 The LINK Prompt

When you run $LINK the link prompt is displayed, so that you can specify the start address of the independent program, and the input files and units containing the modules to be linkage edited:

```
GSM READY:$LINK
$44 LINK:
```

You may key one of the following responses:

#hhhh — where #hhhh is an even hexadecimal number in the range #0 to #FFFE, specifying the program start address. (This response is invalid unless supplied to the very first link prompt);

file-id — the name of a compilation or map file containing a single module to be included in the program file;

library-id/member-id — to indicate that a specific member of a given compilation library is to be included in the program file;

library-id — to search the specified library for any modules containing global definitions matching outstanding global references;

<CR> — to terminate the link list.

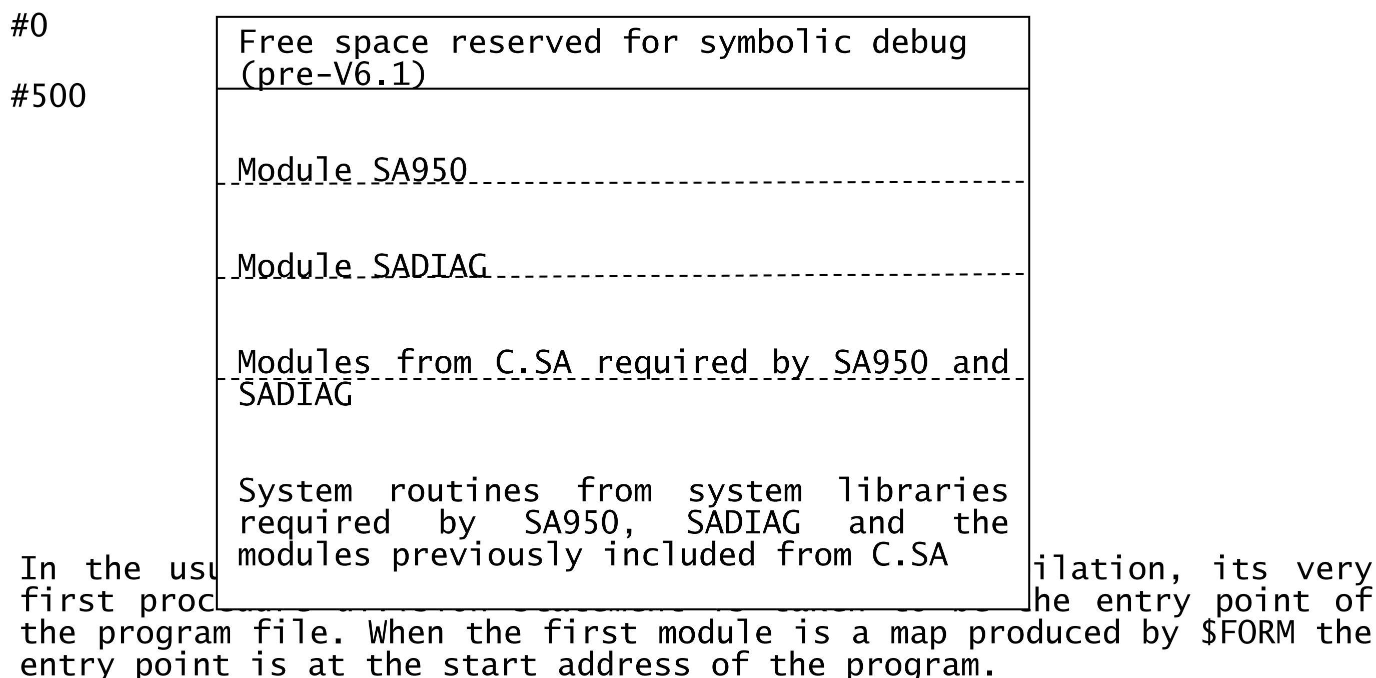
Following the second, third and fourth responses, you are prompted for the unit on which the input file resides. The link prompt is re-displayed until you key <CR>, allowing you to specify details for up to 100 input files. You can respond <CR> to a second or subsequent unit prompt, to cause the last address actually keyed to be used. You may omit the C. prefix when keying a file-id or library-id and $LINK will append it by default. If you do not specify a starting location, $LINK will create the program to start at address #500 to leave space for the overlays of the symbolic debugging system.

In the following example a program is to be linked from module SA950 of file C.SA950 and member SADIAG of library C.TEST. In addition any routines required from library C.SA are to be included. Space is to be left for the symbolic debugging system and all the input files are on unit 205:

```
GSM READY:$LINK
$44 LINK:SA950 UNIT:205
$44 LINK:TEST/SADIAG UNIT:<CR>
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
$44 PROGRAM:
```

Since all but the most trivial programs require modules from the system libraries, they are included automatically when you terminate the link list.

The figure below shows a storage map of the program thus linkage edited:



In the usu ... ilation, its very first proc ... he entry point of the program file. When the first module is a map produced by $FORM the entry point is at the start address of the program.

6.1.2 To Quit

To quit and return control to the monitor, key <ESCAPE> in response to any prompt output by $LINK.

6.1.3 The Program Prompt

The program prompt appears after you terminate the link list by keying <CR> to the link prompt. It requests you to supply the program-id and the address of the unit upon which the program file is to be created:

```
$44 PROGRAM:program-id UNIT:unit-id
$44 LISTING UNIT:
```

For example, to specify that program SALES is to be produced on unit 101:

```
$44 PROGRAM:SALES UNIT:101
$44 LISTING UNIT:
```

You may key <CR> to the PROGRAM: prompt, in which case the name of the program file is derived from the first module in the link list. If it is a compilation, the name used is the one coded in its PROGRAM statement. If the module is a map produced by $FORM, the name is its map-id.

If you key <CR> to the UNIT: prompt the program file is created on the unit occupied by the first module. Therefore, returning to the example introduced in the previous section, the following dialogue could be used to create program file SA950 on unit 205:

```
GSM READY:$LINK
$44 LINK:SA950 UNIT:205
$44 LINK:TEST/SADIAG UNIT:<CR>
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
```

```
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:
```

6.1.4 The Listing Unit Prompt

After you have defined the program file to be produced by $LINK you are asked to key the unit-id of the device to hold the link map listing. You may key <CR> to indicate that you wish to use $PR, the logical printer, or <CTRL A>, to suppress production of the listing. If you specify a direct access device, for example:

```
$44 LISTING UNIT:205
$44 LINK OPTION:
```

then the report is written to a file whose name is constructed from the prefix M. concatenated with the first 6 characters of the program-id. Thus if the above dialogue were used to continue the previous example, it would result in the link map listing being output to M.SA950 on unit 205.

6.1.5 The Link Option Prompt

Once you have specified the listing unit, the link option prompt is output. You may key:

NSD — to suppress the production of symbolic debug records;

OSD — V6.0 format symbolic debug record;

TW — if $LINK is to be run under job management, to indicate that job management is to be terminated in the event of one of the warning messages beginning $44 WARNING appearing;

I=nnnn — Initialise any gaps in the program file (due to uninitialised data in a sub-routine) up to nnnn bytes. The default is 6 bytes.

SD=xxxxxxxx — Create a separate symbolic debug file called xxxxxxxx.

<CR> — to terminate the option list.

LIVE — to link the code at #0000 (DEBUG will set this back to #0500). This is for use with V6.0 or earlier systems.

The link option prompt is output repeatedly, until you key <CR>, so that you can specify several options if you require them:

```
$44 LINK OPTION:NSD
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
```

If you key <CR> to the first occurrence of the prompt (the normal case) symbolic debug records will be produced and the program will not terminate job management if a warning message is displayed.

6.1.6 Processing and Normal Completion

Once you reply <CR> to the link option prompt the message:

```
$44 LINKAGE EDITING
```

is output and $LINK begins the process of combining the modules you specified in the link list together with any that are required from the system library to form the new program file. Your inputs to the link list are processed one by one, in the order they were submitted. Where an input specifies an individual module, this is immediately added to the program file. The globals defined in the new module may be used to resolve previous unsatisfied references, whilst any extra ones are added to the list of currently outstanding references.

Where you specified a library, rather than an individual module, $LINK searches it for missing globals. Only those members of the library which define currently outstanding globals are selected, but as each such member is included in the new program file, any unsatisfied global references that it itself contains are added to the currently outstanding list, and may therefore cause yet more members of the same library to be selected. The arrangement of the members within a library is not important, except if it contains multiple definitions of the same global (discussed in example 2, below). The order in which libraries themselves are specified to the link prompt clearly is significant, because of the way members are selected to satisfy currently outstanding global references. Normally you will include your own user libraries (if any) at the end of the link list, so that they are searched just before the system library, which, since it contains globals referenced by nearly all modules, must be processed last of all.

If at the end of its processing $LINK finds that there are still unsatisfied global references remaining it outputs a warning message of the form:

$44 WARNING *nnnn* UNDEFINED GLOBAL(S)

This message is then followed by a list of the global symbol names which are missing from the program just linked. Except during testing when you have deliberately omitted some modules and plan to exercise the program without them, this message indicates an error situation.

There are two other warnings that may appear during the linkage edit processing:

$44 WARNING NO ROOM FOR SYMBOLIC DEBUG RECORDS
$44 WARNING *xxxxxx* OBSOLESCENT

The first of these appears if the program file is too small, but it is still possible to produce a valid program, providing the symbolic debug records are omitted. The second will only occur if you are linking a program initially developed under an earlier version of Global System Manager and it contains a reference to system routine xxxxxx which is now obsolescent. Although the routine in question is still supplied, the intention is to remove it from later versions of the system.

Even if warning messages occur the linkage edit process may still produce a valid program file and link map listing. When this is accomplished satisfactorily the command outputs the message:

```
$44 LINKAGE EDIT COMPLETED
```

and control returns to the monitor.

6.1.7 Operating Notes

The first filename you specify in response to the link prompt must not be an unqualified library-id, since there is clearly no point in beginning the linkage edit by searching a library when there are no outstanding global references. However you may respond:

library-id/member

since this, of course, specifies that the named member is to be included, not that the library is to be searched. For example, if SA950 were a member of C.SA, rather than being held as an individual file named C.SA950, the link list used as an example in 6.1.1 would become:

```
$44 LINK:SA/SA950 UNIT:205
$44 LINK:TEST/SADIAG UNIT:<CR>
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
```

The result, as before, is to create a program file from SA950, the first module, SADIAG from C.TEST, and any necessary modules from C.SA and the system subroutine libraries.

It is essential that all the files used by $LINK, i.e. those specified in the link list, the system libraries (if they are used), the program file and listing file, remain online throughout the linkage edit process.

The system libraries to be used are specified in a list contained within the $$LINK file, which is located on the $S unit. Normally this link list specifies C.$APF and C.$MCOB to be used (both located on $S), but you may create your own $$LINK file to provide a different set of libraries for inclusion.

When Global System Manager is installed $S addresses the system residence device used for SYSDEV and the system libraries occupy this volume.

There is some room on this volume to add your own user libraries, if you wish.

If the listing file is written to the same unit as the program file it is allocated just 10K bytes, sufficient for 126 lines of output. The program file obtains the maximum remaining contiguous space, and is truncated appropriately at the end of the linkage edit. If either of these allocations proves insufficient, or if you do not comply with certain restrictions concerning the maximum number of modules and globals (as defined in Appendix B of the Global Cobol Language Manual), the linkage edit will fail. A single error message of the form:

```
$44 LINK ABORTED - reason [FILE nn][MODULE nn]
```

will be displayed, and, if possible, printed on the listing. Control will then return to the monitor. Partially completed program and listing files may then be in existence: they can be deleted either by running $LINK again, or by using the file utility's DEL instruction. There is a full description of the various messages that may occur in Appendix B.

The linkage edit will be terminated in a similar way, but without the link aborted message, if there is an irrecoverable I/O error on any of the files involved and you key N to the retry prompt.

6.1.8 Example 1 - The Simplest Linkage Edit

In the very simplest case you use $LINK to linkage edit a single main program, which is not part of an overlay structure, and which uses no subroutines apart from the ones provided as part of Global System Manager in the system library. Suppose $COBOL has been used to produce compilation file C.SA100 on unit 205 from source file S.SA100. Then to create program file SA100, also on 205, and print the link map listing, use the following dialogue:

```
GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

This example creates symbolic debug records in the program file, and makes it load at location #500 onwards, so that a 1280-byte "hole" is reserved at the start of the Global Cobol user area, for use by the debugging system. When the program is fully tested, if it would benefit from having more storage available to it, it could be linked at location 0. This will prevent non-destructive use of the debugging system, whose overlays will corrupt the start of the program. If debug is not expected to be used, or the source is not being provided, you can reduce the size of the program file (though not its main memory requirement) by removing symbolic debug records.

The following dialogue links SA100 to start to location 0, and removes the unwanted records from the program file:

```
GSM READY:$LINK
$44 LINK:#0
$44 LINK:SA100 UNIT:205
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:NSD
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

6.1.9 Example 2 - Using Your Own Subroutine Libraries

If the system you are developing contains a number of common routines, you may store them in one or more compilation libraries to be searched at the end of the link list. A typical scheme might involve two such libraries, C.SA and C.SANEW. The first, updated only infrequently, contains proven code. C.SANEW on the other hand contains either very new routines, or updates to existing ones already on C.SA. The following dialogue will cause the new modules to be selected in preference to the old ones. Both libraries occupy the same unit as C.$MCOB and C.$APF:

```
GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:SANEW UNIT:$S
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:NSD
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

In disk-based systems where plenty of online storage is available, some users find it convenient to keep all the component routines of a system in a single library. Suppose then, that all the components are in C.SA, including the "mainline", SA100. Then to link it to start at location #500:

```
GSM READY:$LINK
$44 LINK:SA/SA100 UNIT:$S
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

6.1.10 Example 3 - Duplicate Global Definitions

Normally you should avoid creating a compilation library where two or more members define the same global and for this reason the librarian, $LIB, outputs a warning message whenever it detects a duplicate global definition within a compilation library. However, there are cases in which duplicate definition may prove useful. Before giving an example, we explain how $LINK handles duplicates.

Clearly duplicate global definitions must reside in different members of the library, otherwise the modules involved would not compile correctly. Therefore if $LINK is searching a library and finds that a global with a duplicate definition satisfies an outstanding reference, it resolves the ambiguity by including only the containing member whose member-id is lowest in ASCII collating sequence (i.e. nearest the front of the alphabet) and in this case there is no problem. However, if $LINK includes a module, for whatever reason, and finds that it contains a definition of a global which has already been defined by a previous inclusion, this is considered to be a fatal error, and the linkage edit is aborted.

Suppose therefore that program SA100 calls a calculation subroutine whose entry name, CALC, is defined by ENTRY statements in two members, CALCA and CALCB, of library C.SA. In outline these members might have been coded as follows:

```
PROGRAM CALCA                       PROGRAM CALCB
DATA DIVISION                       DATA DIVISION
     .....                               ....
     .....                               ....
PROCEDURE DIVISION                  PROCEDURE DIVISION
ENTRY CALC [USING...]               ENTRY CALC [USING...]
     .....                               .....
     .....                               ....
     EXIT                                EXIT
ENDPROG                             ENDPROG
```

The following linkage edit will create program SA100 to use the calculation subroutine from CALCA:

```
GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

However, this one creates a program named SANEW, identical to SA100, except that it uses the calculation subroutine from CALCB:

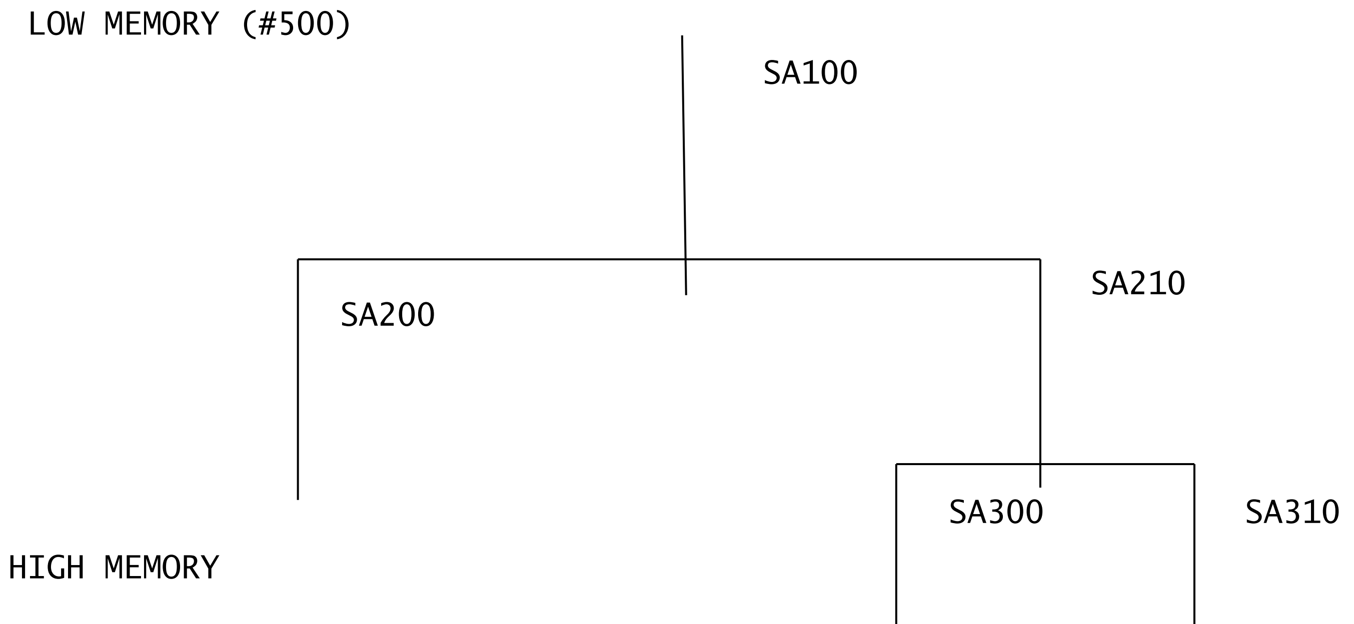
```
GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:SA/CALCB UNIT:$S
$44 LINK:SA UNIT:<CR>
$44 LINK:<CR>
$44 PROGRAM:SANEW UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

Note that if, by mistake, you forced the inclusion of both calculation routines, by starting, for example, like this:

```
GSM READY:$LINK
$44 LINK:SA100 UNIT:101
$44 LINK:SA/CALCA UNIT:$S
$44 LINK:SA/CALCB UNIT:$S
```

then the subsequent linkage edit would abort due to global symbol CALC being doubly defined.

The Ideal Structure



A Possible Implementation

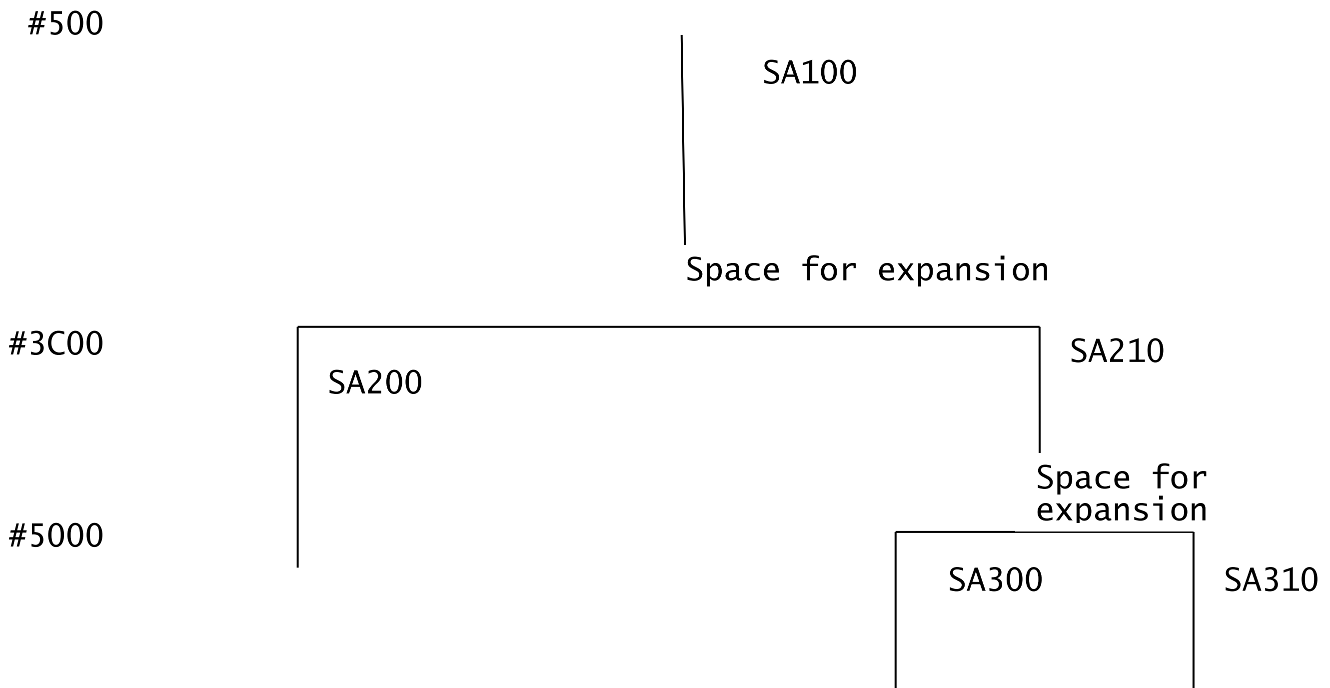


Figure 6.1.11 - Overlay construction using independent programs

6.1.11 Example 4 - Overlay Construction

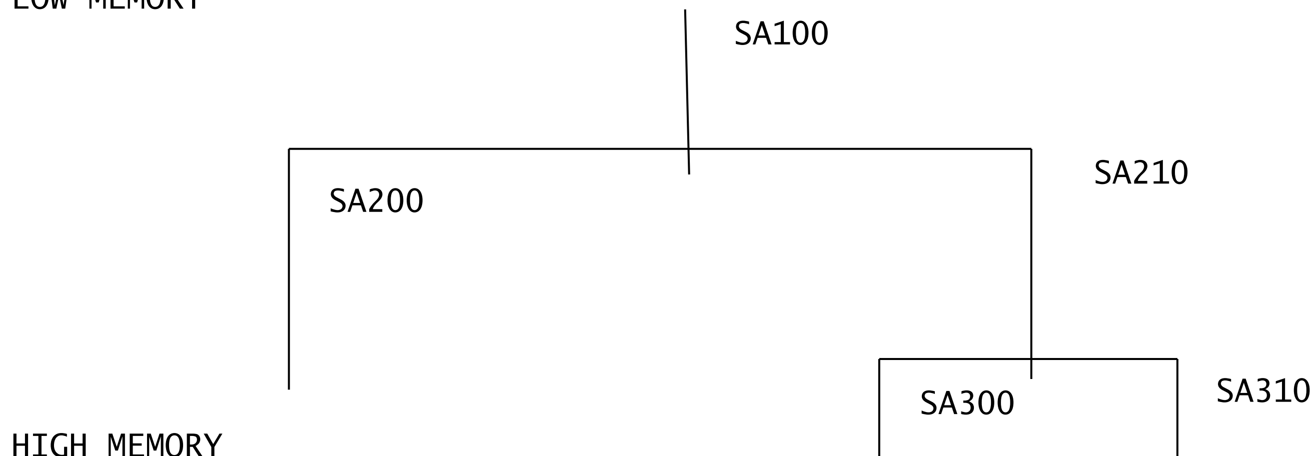
In the top part of Figure 6.1.11 the ideal structure of a simple overlay scheme is shown. There are five programs, SA100, SA200, SA210, SA300 and SA310. The root, SA100, occupies the lowest memory locations. It uses a LOAD or EXEC statement to load one of the overlays SA200 or SA210. The latter can itself load SA300 or SA310 in the same way.

The lower part of the figure shows how the structure might be implemented using five independent programs. First you link the root, SA100, making use of the techniques that have already been described. You examine the link map listing and see that if you begin the next overlay level at location #3C00 there will be space for SA100 to expand slightly if it is subsequently modified. You therefore link SA200 and SA300 to begin at this address. For example, the following links SA200 together with your own subroutine library C.SA:

```
GSM READY:$LINK
$44 LINK:#3C00
$44 LINK:SA200 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

You next link SA210 in the same way and, after examining the listing, determine that SA300 and SA310 should begin at location #5000. Once these are processed, the entire structure will have been created. In all $LINK will have been invoked 5 times, once for each overlay involved.

There are some weaknesses in this scheme. The calculation of start addresses is clumsy, and can lead to errors unless you are careful. Much more of a problem comes about because the five programs are independent of each other, insofar as each must contain all the modules it requires. Thus if SA100 and SA200 each contain an indexed sequential file definition in their working storage, there will be two copies of the indexed sequential access method, module AI$81, in memory at the same time, one linked into SA100, and one in SA200. This can be avoided by special programming, by holding all the FD's in SA100, for example, and passing pointers to them as the lower level overlays are called. However, it would be much simpler if SA200 could simply know about SA100 and use the modules it contains, rather than including them again, unnecessarily. In this case, however, SA200 would be a **dependent** program, since its successful execution would depend on the use of information from another overlay. The linking of dependent programs referencing information overlays is the topic of the next section.



PROGRAM	TYPE	LEVEL	INFORMATION OVERLAYS
SA100	Independent	A	None
SA200	Dependent	B	SA100
SA210	Dependent	B	SA100
SA300	Dependent	C	SA100, SA210
SA310	Dependent	C	SA100, SA210

Figure 6.2 - Overlay construction using dependent programs

6.2 Linkage Editing Dependent Programs

You can create overlay structures which require a minimum of main memory by using a special feature of $LINK which allows you to create what are termed dependent programs. Such a program does not necessarily contain all the modules it requires. Instead it may access modules from other programs which are known to be resident when the dependent is loaded. These other programs are known as **information overlays**, and are the dependent's predecessors in the structure. Thus, in Figure 6.2, SA300 has two information overlays SA100 and SA210. If these three programs all use the indexed sequential access method, and SA210 and SA300 are both linked as dependent programs, then only one copy of the access method module, AI$81, will be included in the structure, and it will be a part of SA100. Note that the first level of the structure, the **root program**, SA100 in the example, is unique in having no information overlays. It need simply be linked as an independent program using the techniques already described.

Linking of a dependent is slightly more complicated than that of an independent program, since you have to specify link lists defining each of the information overlays preceding the dependent. Apart from this the principles remain unchanged. The information overlays themselves do not become part of the resulting program file: they are simply used in resolving global references and determining where in memory the dependent program is to be loaded. A single invocation of $LINK still produces just one program file, so the command must be executed once for every program of the structure.

6.2.1 Defining Information Overlays

When you run $LINK to create a dependent program you must key <CTRL A> to the very first link prompt to indicate that you wish to define its information overlays:

```
GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:
```

The linker responds by displaying a confirmation message noting that you are about to define the first information overlay (the root program), considered to be at level A. The link prompt is then redisplayed so that you can begin to key the link list defining it. The contents and order of this list should be exactly the same as the one used to create the information overlay originally. However, when you come to the last link prompt, you may either key <CTRL A> if you need to define another information overlay, or <CR> if the information definitions are complete and you wish to proceed to link the dependent program itself.

For example, let us assume a very simple system in which there is not even a user subroutine library. Then the root, SA100, could be linked as described in 6.1.8:

```
GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:<CR>
etc, etc.
```

This will cause a program file loading at location #500 to be developed. To link dependent program SA210, so that it uses modules already indicated in SA100 whenever possible, you must specify the same link list again, to define the information overlay at level A. Since there is only one such overlay, you terminate the list by keying <CR>, then link the dependent program itself:

```
GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:205
$44 LINK:<CR>
$44 DEFINE DEPENDENT PROGRAM AT LEVEL B
$44 LINK:SA210 UNIT:205
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

You will note that the dialogue following the line beginning "$44 DEFINE DEPENDENT PROGRAM" is exactly the same as that used for an independent program. The dialogue to link SA310, a dependent of SA100 and SA210 at level C, is therefore:

```
GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:101
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL B
$44 LINK:SA210 UNIT:101
$44 LINK:<CR>
```

```
$44 DEFINE DEPENDENT PROGRAM AT LEVEL C
$44 LINK:SA310 UNIT:<CR>
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

6.2.2 Error Recovery

If you terminate a link list by keying <CTRL A> when you should have keyed <CR>, or vice versa, you can correct this error by making the right response to the next link prompt. <CTRL A> will return you to information overlay definition mode, whilst <CR> will allow you to create the dependent program at the current level. For example:

```
.........
$44 LINK:<CTRL A> (keyed erroneously)
$44 DEFINE INFORMATION OVERLAY AT LEVEL D
$44 LINK:<CR> (the correct response)
$44 DEFINE DEPENDENT PROGRAM AT LEVEL D
$44 LINK:
```

6.2.3 Start Address Considerations

If none of the link lists involved specify an absolute start address, then the root information overlay will be assumed to begin at location #500, and the other information overlays and the dependent program itself will follow on and occupy contiguous storage. You may of course decide to start the root at some other location, such as address 0, in which case you must begin its link list whenever it is used with a #hhhh response. In our example, the list might then appear as:

```
$44 LINK:#0
$44 LINK:SA100 UNIT:205
$44 LINK:
```

You may also start other programs of the overlay structure at absolute addresses, but this is not generally useful, since you normally will need to relink all the dependents of a particular overlay if it changes at all, because the global symbols it defines will have moved.

In very special circumstances you may start a program at an absolute address in order to force it to overwrite part of its previous information overlay when you know it is no longer required. This technique is not recommended, since in most cases it is better to introduce an additional program in place of the overwritten portion. In any case, if $LINK detects that by specifying an absolute address you have forced one program into the memory area occupied by another, it outputs a warning message of the form:

```
$44 WARNING LEVEL y CONFLICTS WITH LEVEL x MEMORY AREA
```

For example, if you decide to force SA200 to begin at location #2000, actually within the root, SA100, the dialogue appears as:

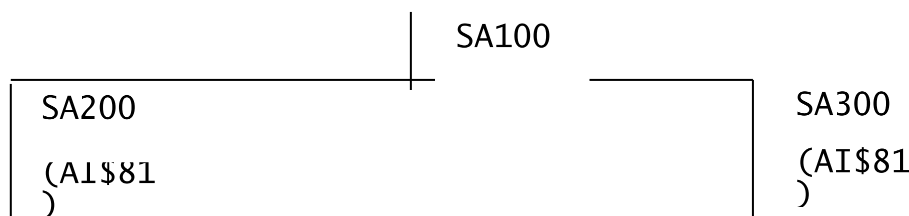
```
GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:205
$44 LINK:<CR>
```

```
$44 DEFINE DEPENDENT PROGRAM AT LEVEL B
$44 LINK:#2000
$44 LINK:SA200 UNIT:205
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 WARNING LEVEL B CONFLICTS WITH LEVEL A MEMORY AREA
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

When one or more information overlays are linked at an absolute address, then a subsequent overlay with no start address specified will be linked following the highest memory location occupied by **any** of the previous information overlays.

6.2.4 Removing Duplicate Modules from the Overlay Structure

Suppose in our example that programs SA200 and SA300 use IS files, but the root, SA100 does not. If no special steps are taken the indexed sequential access method, module AI$81, will appear twice in the overlay structure thus:



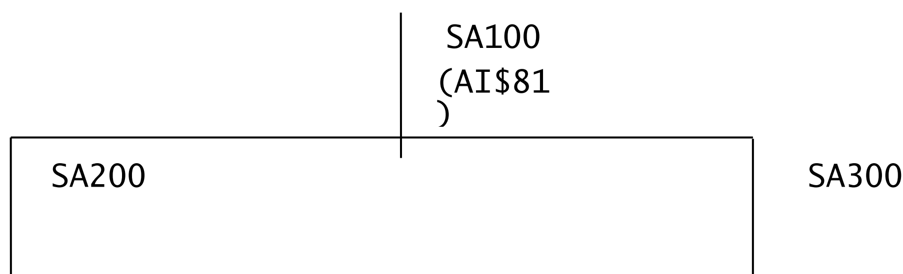
However, you can force AI$81 into the common information overlay shared by its users (i.e. in this case the root program, SA100) in two ways. You can either link it explicitly into SA100 by extending its link list, for example:

```
$44 LINK:SA100 UNIT:205
$44 LINK:$MCOB/AI$81 UNIT:$S
$44 LINK:
```

Or you can change the source of SA100 to include a reference to AI$81 and then recompile it. Rather than setting up an unnecessary indexed sequential FD, simply code a statement:

```
GLOBAL AI$81
```

at the front of the data division preceding the first file, map or data definition. In either case the result will be an overlay structure such as:



The main memory requirement will be identical to that of the previous structure: the saving will be made on the file space required by the overlays, and, in consequence, the time taken to load them. We recommend that when you are developing an application you examine the link map listing carefully with a view to moving duplicated modules to the root (or an appropriate overlay at a lower level) since the total savings can be considerable.

A similar problem arises if you link a dependent program which uses blocked relative sequential files together with an information overlay which uses only unblocked RS files. The dependent requires access method AR$B, whereas the information overlay already includes AR$A, which is really superfluous, since AR$B contains a superset of its functions. The message:

```
$44 WARNING 2 VERSIONS OF RSAM PRESENT
```

will be displayed in this case. You can remove AR$A from the structure by linking AR$B explicitly into the relevant information overlay, or one of its predecessors in the structure. Alternatively, you can update the appropriate source code and recompile, having coded:

```
GLOBAL AR$80$
```

at the front of the data division. (In this very special case, you must not code GLOBAL AR$B as you might expect. This is due to technical problems involved in the handling of blocked and unblocked RSAM, which are beyond the scope of this manual).

6.2.5 Operating Notes

You may specify up to 9 information overlays when linking a dependent program. Each is identified by means of its level code, which is A for the root program, B for the overlay adjacent to it in main storage, and so on, down to I. The dependent program is linked at the level below that of the last information overlay you specify.

The program occupying each level is defined by its link list, which is automatically supplemented by including the system libraries at the end in the normal way.

If, at the end of linking either an information overlay, or the dependent program itself, there are still outstanding global references, a warning message of the form:

$44 WARNING *nnn* UNDEFINED GLOBALS AT LEVEL *x*

appears, together with a list of the missing globals. The quantity nnn is the decimal number of distinct symbols not present, and x is the level code of the information overlay affected, or the dependent program itself. Except when you have deliberately omitted modules during testing, this message normally indicates an error condition.

6.2.6 Example - Linking an Entire Structure

This example shows the dialogue necessary to link all five programs of the overlay structure shown in Figure 6.2. Compilation files C.SA100, C.SA200, etc. are assumed to reside on 205, on which the new program files are developed.

All parts of the structure may use routines from a user subroutine library named C.SA. The link map listings are all produced on the unit assigned to $PR, the logical printer.

The dialogue is intended to be used under job management, so the TW option is specified to cause job management to terminate if any of the conditions that give rise to a $44 WARNING message arise:

```
(First link step - link SA100)

GSM READY:$LINK
$44 LINK:SA100 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:

(Second link step - link SA200)

GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 DEFINE DEPENDENT PROGRAM AT LEVEL B
$44 PROGRAM:SA200 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:

(Third link step - link SA210)

GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 PROGRAM:SA100 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 DEFINE DEPENDENT PROGRAM AT LEVEL B
$44 LINK:SA210 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:

(Fourth link step - link SA300)

GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:205
```

```
$44 LINK:SA UNIT:$S
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL B
$44 LINK:SA210 UNIT:205
$44 LINK:<CR>
$44 DEFINE DEPENDENT PROGRAM AT LEVEL C
$44 LINK:SA300 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

(*Fifth link step - link SA310*)

```
GSM READY:$LINK
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL A
$44 LINK:SA100 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CTRL A>
$44 DEFINE INFORMATION OVERLAY AT LEVEL B
$44 LINK:SA210 UNIT:101
$44 LINK:<CR>
$44 DEFINE DEPENDENT PROGRAM AT LEVEL C
$44 LINK:SA310 UNIT:205
$44 LINK:SA UNIT:$S
$44 LINK:<CR>
$44 PROGRAM:<CR> UNIT:<CR>
$44 LISTING UNIT:<CR>
$44 LINK OPTION:TW
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETED
GSM READY:
```

7. Symbolic Debugging Using $DEBUG

INSTRUCTION	GROUP AND FUNCTION
	INTRODUCTION (7.1)
H	Display help text
Q	Quit, returning control to the monitor
	THE SCREEN DISPLAY (7.2)
W size	Change window size
	ESTABLISHING THE SYMBOLIC DEBUG TABLE (7.3)
N name	Name the compilation
U	Change symbolic debug table library
	THE DIAGNOSTIC REPORT (7.5)
D	Produce diagnostic reports and dumps
	INSPECTING AND MODIFYING VARIABLES (7.6)
I address [format]	Inspect a variable
M address [format]	Modify a variable
X index1 [index2]	Index the next I or M instruction
Y length	Establish length of repeating group
S string	Search for string in memory
? address	Displays address as address (base+offset)
	RESUMING PROGRAMS FROM $DEBUG (7.7)
R [address]	Resume execution of an interrupted program
A	Advance a single step
	SETTING TRAPS (7.8)
T address	Set a trap on a Global Cobol statement
C [address]	Clear the trap on a statement
O program-id	Trap the loading of an overlay/chained program
V address [format] condition value	Value trap
	LISTING FILE INSTRUCTIONS (7.9)
E	End or start inspecting a file
L number	Get line number
F string	Find source line string
G hex	Get source line containing address
	TEMPLATES (7.10)
K	Keep template
	ALTERNATIVE DEBUGGING MODES (7.11)
P	SCF debugging mode
Z	Assembler debugging mode

Table 7.1 - Debug Instructions

7.1 Introduction

The symbolic debugging system described in this chapter is intended for programmers responsible for developing Global Cobol applications. The full debugging system explained below enable you to inspect and modify variables, set traps in programs and on variables, and so on, using symbolic information if it is available. The debugger runs in a separate partition from the program being debugged. This is usually a second partition on the same screen, but can be a completely separate screen from the one on which the program is running. Because it uses a

separate partition the V6.1 debugger cannot be used on a single user system.

Note: The V6.1 debugger should not be used with programs that gain exclusive control.

This chapter begins by explaining the concepts underlying the system. A description of the individual instructions used in debugging follows, with related instructions grouped together in the same section so that it is easy to appreciate the scheme on a first reading.

7.1.1 Running the $DEBUG command

To start debugging a program you run the $DEBUG command in the normal way from a ready prompt. For example:

```
GSM READY:$DEBUG
```

You are then asked for the number of the partition you want the program you would like to debug to run on:

```
Partition to be debugged:partition
```

The partition must be a partition other than the one on which you are running $DEBUG. To debug a program running on another screen, you should key the operator-id followed by the partition number; for example:

```
Partition to be debugged:NER/2
```

At this point the debugger attempts to make the partition resident in memory. If this is not possible because there are no free memory banks available, the following message will appear on your screen:

```
TOO MANY TASKS ARE CURRENTLY KEPT RESIDENT IN MEMORY FOR
THE TARGET PARTITION TO BE KEPT RESIDENT AS WELL.

YOU MAY TRY REDUCING THE NUMBER OF RESIDENT TASKS (BY
TERMINATING DEBUG SESSIONS FOR EXAMPLE), AND TRY AGAIN. (SEE
CU-7 FOR DETAILS)

Key <CR> to retry or <ESC> to abandon:
```

You may be able to free some memory banks by reducing the number of tasks resident in memory. This can be done on the system by stopping some of the other programs that may be running, especially programs that may be explicitly keeping a partition resident, for example $DEBUG. If you are not in a position to try this, key <ESC> at this point to abandon the debugging session. If you can do this, do so, and then key <CR> to the prompt. The debugger will then again attempt to find an available memory bank.

Figure 7.1 - Selecting the template

It maybe that the partition being debugged is not readily available in memory because it has been swapped to disk. You will then get the following message:

```
THE TARGET PARTITION IS NOT CURRENTLY RESIDENT IN MEMORY.
```

```
YOU SHOULD BE ABLE TO CAUSE IT TO COME INTO MEMORY BY
RUNNING A PROGRAM IN THE TARGET PARTITION (SEE CU-7 FOR
DETAILS)

Waiting for target partition to become active. Key <CTRL G> to abandon
```

The debugger will now be in a state where it is waiting for you to activate the target partition. It should be possible to do this by switching to the partition and running a program and then going back to a ready prompt. You will then need to switch back to the debug partition. If you are at the next prompt the partition to be debugged has become resident. If this is not so you may want to try again or key <CTRL G> to terminate the debugging session.

Once the target partition is resident if there is a template data library present the debugger will then ask you for the name of the template you want to load:

```
Key template name, ? to list, <CR> for none:
```

If you key ? to this prompt the list of available templates will be displayed allowing you to choose one as shown in figure 7.1.

You can key the name or number of the template you want to choose or D to delete any old templates.

A template contains the details of the debugging system for a specific program. These include:

- the program to be debugged;
- the overlay to be trapped;
- bases;
- variables to be shown in the formatted area;
- file under inspection;
- window of application last selected.

Further detail on templates is given in section 7.10. If you have a file being inspected on a spool unit and there is more than one file of this name, you will be asked at this point which file you want to inspect as explained in section 7.9.1.

If no template is specified or there is no template file, $DEBUG will then ask for the program you want to debug and the overlay you want to set the initial trap at:

Program to be debugged *program* Overlay to be trapped *overlay*

The overlay trap will be signalled after the overlay requested is loaded. The default will set the overlay trap on the actual program to be debugged.

$DEBUG will now run the program in the partition you specified so it is advisable to make sure this partition is at a ready prompt.

You may not want to debug a program at this particular time but you may just want to run the debugger in case you get an unexpected error in the target partition. In this case you should just key <CR> to the program prompt. You may now key debugger commands to set some initial

debug information (the instructions are described in further chapters). You must then key R to resume the debugging session. The following message will then appear:

```
AWAITING PROGRAM CHECK IN PARTITION BEING DEBUGGED
```

You may now switch to the partition being debugged and continue with your application. If an error occurs the debugger will be activated.

You can now begin debugging your program. Instructions are provided to allow you to inspect and modify the program loaded and eventually cause it to be executed. You can also set traps on selected Global Cobol statements so that, when a statement is encountered, the debugging system is reactivated. This enables you to test a program in a highly controlled manner since you can determine exactly those points at which debug is to be re-entered to allow you to inspect or modify storage, or even set further traps.

7.1.2 Entering Debug from an Active Program

If there is no debugger active for the partition then following any program check, stop code, exit code, or trap, the monitor loads a diagnostics overlay and one of the following messages will appear:

```
$91 PGM CHECK number AT location
$91 TERMINATED - STOP code
$91 TERMINATED - EXIT code
$91 TRAP AT location
$91 OVERLAY TRAP ON program-id
$91 VALUE TRAP AT location
```

This will be followed by the diagnostics prompt:

```
Activate debugger, key D for diagnostics report, H help, or <ESC> to exit:
```

The diagnostics overlay is also entered when the operator interrupts the program using <CTRL W> and replies Y to the resulting break prompt:

```
<CTRL W>
$91 BREAK?:Y
Activate debugger, key D for diagnostics, H for help, or <ESC> to exit:
```

Keying <ESC> to this will return you to the ready prompt.

A reply of D or <CR> will print a diagnostics report on unit $PR. You may key up to 10 lines describing the events causing the program error. These lines can be edited in the same way as text in $TED (see OM-6), except that the commands are not available. You will then be asked if a full memory dump is required:

```
Full memory dump?:
```

A reply of Y will cause the diagnostics to print a full memory dump to the $PR unit.

A reply of D<CTRL A> to the diagnostics prompt will cause the concise report to be displayed on the screen.

You may want to debug the program in which case you will need to switch partition and activate a debugger as described in section 7.1.1.

7.1.3 Debugging User System Requests

When debugging user system requests you must always run the debugger as described in section 7.1.1 and not as described in section 7.1.2. The memory you will inspect if you run $DEBUG after the diagnostics prompt has been reached, will have been replaced by the calling program and will not be that of the system request itself. The debugger must also be run from a separate screen.

7.1.4 Debug Instructions

Table 7.1 summarises the format of the debug instructions described in detail in this chapter. Most of them take one or more operands, though in some cases these may be optional.

7.1.5 Address Specification

Many of the commands take an address operand, to specify an address on which you want to apply the operation. The address format is as follows:

```
[@]address [(symbol)|(decimal)]
```

where address is one of the following:

```
symbol
hex
decimal.
,hex
,decimal.
b,[hex]
b,decimal.
```

and:

```
@ means indirection
b is a single digit indicating the base to be added.
```

You can simply specify the hex address from address zero by keying a "," followed by the hex value. For example:

```
,500
```

You can also express this address as a decimal value as follows by following the decimal value with a decimal point:

```
,1280.
```

The address of a symbol can be indicated by keying the symbol name. If a symbol is also a valid hexadecimal address then the symbolic format is assumed. Subject to the limitation that hexadecimal operands can consist of a maximum of four characters, a 0 prefix can be used to indicate hexadecimal format (e.g. AA1 may be a symbol, but 0AA1 must be hexadecimal).

If you want to access an address offset from a base address (explained later) you key the base followed by the hexadecimal or decimal address. For example, to access (base 1 + #500) you key:

```
1,500          (or 1,1280. for the decimal)
```

A hexadecimal address not prefixed by "," will always have the default base, base 0, added to it. For example:

```
500
```

will access (base 0 + #500)

You can express an index using a decimal value or a computational symbol. For example, if the following symbols have been introduced by an N instruction (explained later):

```
77    ABC   PIC X(3)
            VALUE "ABC"
77    ALPHA REDEFINES ABC OCCURS 3 PIC X
```

and:

```
77    INDEX PIC 9(4) COMP
      VALUE 3
```

then:

```
ALPHA(INDEX) will access "C"
```

and:

```
ALPHA(2) will access "B"
```

The "@" prefix is provided to allow you swift access to data structures which are linked by pointers. It is also useful for handling linkage section or system area data if you are unable to refer to it symbolically.

For example if you wish to access the area pointed at by the pointer PAPTR whose address is #500 from base 9 you can key:

```
@PAPTR
```

or:

```
@9,500
```

or:

```
@9,1280.
```

7.1.6 Format

For some instructions you may need to specify a format in which you want the value specified by the address to be displayed or accepted. The format must be separated from the address by at least one space. The available formats are:

Cn or Cn,m	computational 9(n,m)
C	9(4) COMP
Xn	X(n)
X	X(1)
H	hexadecimal 16 bytes
Hn	hexadecimal n bytes
P	pointer

D   date
F   floating point

If the format is omitted and the address is not a symbol a format of H16 is assumed.

7.1.7 H - help

If you want to refer to the help text given in the debugger you just key the instruction H to the command prompt.

7.1.8 Q - quit from the debugger

To quit from the debugging session and return control to the monitor, when the problem that caused the error is fully understood, key Q to the command prompt. You will be asked if you want to terminate the program in the partition being debugged. You will then be asked if you want to keep the current template. If you reply Y to this you will be asked for the program name, overlay trap record name and title as explained for the K command in section 7.10.

7.2 Screen display

When you are in a position to debug your program you will find the screen divided into three areas. First there are two lines of heading. These contain the basic status information such as the last program check, stop code, exit code, or trap as well as any outstanding overlay or value trap. Next is a window area followed by a scrolled area for keying commands.

The window area can be used to display various information by keying function keys F1-F5 to the command prompt:

```
Command
```

The windows available are:

F1 - Selected variables;
F2 - User screen;
F3 - Source file;
F4 - Diagnostics;
F5 - Scientific variables.

7.2.1 Selected variables window

The selected variables window is chosen by keying F1 to the command prompt. The last name table loaded using the N instruction will be shown on the second header line. This window allows you to display up to 30 variables in your program. The variables will be displayed in the window at a specified position, and their values updated when there is an error or a trap in your program, or when you modify the variable using an M command.

To enter the mode in which you can edit the variable window area you key one of the following:

cursor up   to position yourself in the top left hand corner of the window;

cursor down to position yourself in the bottom left hand corner of the window;

home to position yourself to the last place previously positioned in the window.

You can now use the cursor positioning keys to move around the window area.

To select a variable that you would like to be defined in the window you position the cursor at the place where you want it displayed. You then key F1 and are prompted for the address and format of the variable as follows:

Variable *address* [*format*]

The address and format are as described in sections 7.1.4 and 7.1.5.

The variable will now be displayed at the defined position in the window.

Other functions are provided to allow you to edit the variable window. You can delete a variable at the current cursor position in the window by keying F2 and insert and delete a line at the current cursor position in the window by keying F3 and F4 respectively. If you specify a new variable whose display overlaps that of a previously defined variable, the earlier definition is automatically deleted.

Once you have completed editing the window you key <ESC> to return to the command prompt.

The values of the variables in the symbol window will be updated every time there is an error or trap in the program you are debugging, whenever you modify the variable using the M instruction or as you advance through the program using the A command.

7.2.2 User screen window

This window is chosen by keying F2 to the command prompt. The partition number of the user partition will be shown on the second header line. The window shows the screen of the partition being debugged and is useful if you are debugging another screen which is remote. You can view the sections of the screen that do not fit into this window by keying cursor down and cursor up respectively to the command prompt. You can also move up and down a number of lines as follows:

Command +*n*

or:

Command -*n*

where n is the number of lines.

To return to viewing the start of the screen you key the home key.

This window will be refreshed whenever the program you are debugging has an error or trap.

7.2.3 File inspect window

This window is chosen by keying F3 to the command prompt. It shows the file specified by the E command described in section 7.9., the name and unit of which will be shown on the second header line. You can move forward or back a screen by keying cursor down or cursor up key and move up and down a number of lines by keying + or - as with the user screen window. You can also use the home key to display the first lines of the listing. Other instructions provided for inspecting the file are explained in section 7.9.

Provided that the listing file corresponds to the program being debugged, the base 0 address correctly set, and the "Track addresses in listing file" option has been selected, the debugger will track the line in the listing file which contains the address of the last error, trap or value trap.

7.2.4 Diagnostics report window

This window is chosen by keying F4 to the command prompt. It shows a concise diagnostic report, which is often sufficient to pin-point the cause of the error without further investigation. Figure 7.3 shows the format of the diagnostics report, together with an example.

You can move back and forwards and go back to the start in the diagnostics window as previously described for the user screen and listing file windows.

The window will be updated at every error or trap in the program you are debugging.

7.2.5 Scientific variables window

This window is chosen by keying F5 to the command prompt. If you have any scientific variables in the program you are debugging, it shows the last scientific instruction together with the scientific variables and their values. You can move the display up and down as previously described.

The window will be updated at every error or trap in the program you are debugging.

7.2.6 W - Change window size

You can change the depth of the window area shown on the screen by using the W instruction as follows:

```
Command W size
```

where size is the size of the window area. This size excludes the lines drawn at the top and bottom of the window.

7.3 Establishing the Symbolic Debug Table for a Compilation

The instructions described below allow debug to access the short names and attributes of all symbols defined in a single compilation. You can then use any such symbol (rather than a hexadecimal location and attributes) as the location operand of the instructions which allow you to inspect and modify variables, display memory, clear and set traps, and so on.

The symbols introduced by the N instruction remain available either until another N instruction is used, until you quit the debugger or until the program is no longer on the current program unit.

Only the first six characters of each symbol are retained in the symbolic debug table so if you are testing a program which has been compiled using the "long names" option and several of its names begin with the six characters you key, this will be interpreted as a reference to the very first such name appearing in the compilation. If you supply a symbol as an operand and it is not present in the debug table the following warning message is displayed:

```
INVALID - REINPUT
```

In programs compiled with pre-V6.1 Global Cobol, global symbols only appear for the compilation in which they are defined, not for every compilation which references them. In particular, variables in external sections are not available.

7.3.1 N - Name the Compilation for Symbolic Debugging

To name the compilation in which any symbols to be used in subsequent instructions are defined, you must key the command:

```
N
```

You are then asked to key the template name:

```
Key template name, ? to list, <CR> for none
```

This allows you to use a previously created template that displays the required variables in the selected variables window.

Whether or not you use a template you are next asked to supply the compilation name and the program-id:

```
Compilation name name Program-id program
```

where the name you supply is that used in the compilation's PROGRAM statement and program is the program which contains the compilation. You may reply <CR> to either of these to accept the default displayed. (This default option is useful when you wish to work with the **main program** of a program constructed according to the recommended naming conventions, under which the program-id and main program name are identical.)

7.3.2 N - Examples

A sales invoicing program (program-id SA100) is under test and you wish to examine data within its main program (program name SA100). This was the last program loaded.

You need only key:

```
Command N
Key template name, ? to list, <CR> for none <CR>
Compilation name <CR> Program-id <CR>
```

to establish its symbols in the symbolic debug table because in this, the most usual case, all the default conditions prevail.

Next you need to examine data within one of this program's subroutines (program name SASUBR). You key:

```
Compilation name SASUBR Program-id <CR>
```

The symbols of SA100 are replaced by those of SASUBR.

You now wish to continue execution of the program, and expect that it will load an overlay. You use debug's R instruction (explained later) to continue executing the program and it loads the overlay SALESOVL, which terminates in error with a program check. SA100 is no longer the last program loaded - that is SALESOVL - so if you wish to look at data within SASUBR again you must key:

```
Compilation name SASUBR Program-id SA100
```

7.3.3 N - Operating Notes

When the N instruction is successful your reply to the "Program-id" prompt is immediately followed by a command prompt to allow you to key the next instruction. In some cases, however, the N instruction will fail and a warning message will precede the command prompt.

If the program is not found, of the wrong type or corrupt in some way, one of the following messages will appear:

```
PROGRAM FILE NOT FOUND
```

or:

```
INVALID PROGRAM FILE
```

If no symbolic information was present for the compilation in the program specified the following warning message will appear:

```
NO SYMBOLIC DEBUG RECORD
```

Either you have used the N instruction incorrectly, and the wrong compilation name is being searched for, or no symbols are present in the program file. The latter will be the case if the module was compiled or linked using the NSD option, or if the symbols were eliminated later during library maintenance by using $LIB's NSD instruction. It may also mean debug is unable to access the file containing the symbols for some reason.

7.3.4 U - Change Name Table Library

The N instruction will first look for the program module from which to load the name table in the last attached program library. The U instruction described below allows you to change this library. This is especially useful if you have used $LINK to create separate symbolic debug modules (CU-6.1) and have placed them in a separate library. It also allows you to load the name table even if the program you want to debug attaches another program library.

To change the library from which the symbols are loaded you must key the command:

```
U
```

You are then asked to key the library name and unit as follows:

```
U Library name name Unit unit
```

The default library name and unit supplied is that of the currently attached library.

7.3.5 U - Operating Notes

If you do not supply a library name or unit then the currently attached library will be used for the N instruction.

If the program library is not present on the unit then the following warning message will appear:

```
NOT FOUND OR WRONG TYPE
```

7.4 Setting the Program Base

There are 10 base registers (0-9) for you to use to set **base addresses.** When you successfully use the N instruction to name the compilation whose symbolic debug tables are to be loaded, $DEBUG automatically establishes the base 9 address and the default base - base 0 - to be the start address of the compilation thus identified. The first effect is to cause location information appearing in the diagnostics window and any of the error messages:

```
TRAP AT location

VALUE TRAP AT location

EXIT CODE nnnn AT location

STOP CODE nnnn AT location

PGM CHECK nn AT location

ADVANCED TO location
```

that appear in the header lines will be displayed in the form:

```
address (base 0 + offset)
```

For example, if program SAPROG has been linked to start at location #500, and the base has been set to #500, then if an overflow program check takes place at location #0648, the message:

```
PGM CHECK 11 AT 0648 (0500+0148)
```

appears on the header line. The importance of the **offset**, 0148 in this example, is that it relates to the location printed on the compilation listing and saves you having to perform hexadecimal subtractions to convert the addresses used at run-time to listing locations.

The base addresses are also used in establishing addresses as described in section 7.1.4.

7.4.1 - Examine Base and Set Base Explicitly

Bases 0-8 can be set up explicitly. Base 9 can only be set by an N instruction. To examine a base you must key the base number to the command prompt:

```
Command n
```

The address will then be shown on the same line. With bases 0-8 you will be allowed to edit the address in the format described in section 7.1.4. Base 0 is the base used when displaying the location in the error messages, diagnostics report and symbol window. It is also the base used to calculate the address in the listing file in the G instruction which is explained later.

7.4.2 Bases - Operating Notes

Bases are most useful when you are debugging programs where symbolic information is not available, because it allows you to work with the locations printed on the compilation listing rather than absolute addresses, which require hexadecimal conversion. They can be used in defining an address as described in section 7.1.4. Before you use the other debug instructions you should determine from your linkage edit map listing the starting location of the compilation in which you are interested. This is the value that appears in the LOCN column next to the program name in the PROGRAM column. The value is always hexadecimal 500 for main programs linkage edited at the linkers default address.

You can avoid having to use the link map in this way by exploiting the symbolic debugging facility and relying on the N instruction to set the program base.

Base addresses can also be used to ease access of linkage section item offsets (see I instruction).

Figure 7.2 - Example dump

Figure 7.3 - Example diagnostics report

7.5 The Diagnostic Report

The D instruction described in this section allows you to print the diagnostic report, together with selected dumps of main memory. The diagnostics facility, entered following a program check when the debugger is not activated also produces a diagnostic report, and this can be followed by a dump of the whole of memory.

Figure 7.5 describes the format of the diagnostic report and the photograph shows a report produced following an error in program SAPROG, which was deliberately constructed to fail by dividing by zero.

7.5.1 D - Produce Diagnostic Report and Dumps

To produce a diagnostic report (and possibly follow this by printing memory dumps) you must key the instruction:

```
D
```

You will then be asked for a print unit:

```
Print unit unit-id
```

The initial default will be $PR. This causes the diagnostic report to be written as a print file to the unit-id you have specified. For example, to write the report to the standard printer:

```
Command D Print Unit $PR
Dump from
```

The "dump from" prompt that now appears allows you to specify the start address (in the standard format address) of the memory area you require to output, together with its length:

```
Dump from address Length length
```

For example:

```
Dump from SAREC Length 100
Dump from
```

or:

```
Dump from EC0 Length #3C0
Dump from
```

These examples show how the first input you supply, address is an address of the format explained in 7.1.4, and the second input, the length, may be either specified as a decimal number or as a hexadecimal value introduced by a #-character. Figure 7.2 shows the dump produced as a result of the second example, when the program base is #500.

Once the information has been printed, a new dump prompt appears so that you can print out another area, if you wish. If you do no want to print any further dumps then reply <CR> to the dump prompt. This closes the print file and displays the debug prompt.

For example:

```
Command D Print unit $PR
Dump from SAREC Length 100
Dump from EC0 Length #3C0
Dump from CB-DAT Length #8E
Dump from <CR>
Command
```

7.5.2 D - Operating Notes

Usually the dump is written to the standard printer, but it may be produced on a direct access device. In the latter case it is output to a file named D.$DEBUG.

When you key the D Print unit unit-id instruction any existing file named D.$DEBUG is deleted, and a new file with that name created. The file is allocated the maximum amount of contiguous space available on the volume. Any unused space is returned to the system for re-allocation when you reply <CR> to the dump prompt to obtain a new debug prompt.

The prompt:

```
Reset?
```

appears if an I/O error occurs when debug attempts to open the dump file in response to a D Print unit unit-id instruction. If you reply Y

this will cause any files currently in use by the program under test with the printer in question to be closed, and the open operation to be retried.

This may allow the dump to be produced successfully if the original error arose because the program under test already had a file open on the printer. If you reply Y you may be able to produce the dump.

There are, of course, other possibilities for error if you attempt to write the dump to printer in a multi-user environment. You are likely to find it is in use by another program. You should normally try to produce the dump on a direct access spool unit or work unit, or your own private work unit.

If you reply N, <CR> (or any single character apart from Y), the dump request will be abandoned and the command prompt will re-appear.

7.5.3 Notes on the Diagnostic Report

The last file access may be one performed using a system routine, for example CONV$.

The final part of the report, the stack listings, will normally only be of interest if you are loading modules onto one of the stacks, as described in the Global Cobol Assembler Interface Manual. It may also contain modules loaded by Global System Manager. In particular, job management and field editing all involve the loading of modules onto the stack. If a stack is empty, then the listing is compressed into a single line of the form:

```
SYSTEM STACK EMPTY nnnnn BYTES SPARE

check-type [CODE code] AT location-1
[SYSTEM system SERIAL NUMBER serial]
LAST PROGRAM LOADED program ACCUMULATOR VALUE accumulator
LAST TRANSFER FROM location-2
[LAST FILE ACCESS operation ON file-id INDEX index]
GSM version CALLED      RETURN ADDRESS
[name]      location-3  location-4
......      ..........  ..........  |  One such line for
......      ..........  ..........  |  each level of control
......      ..........  ..........  |  in the calling
......      ..........  ..........  |  sequence at the point
......      ..........  ..........  |  of failure. The
......      ..........  ..........  |  deeper the level, the
......      ..........  ..........  |  later it is listed
......      ..........  ..........  |

SYSTEM STACK MODULE TYP USER  ENTRY   SIZE
             module typ user  address size
             ...... ... ....  ....... ....  |  One such line for each
             ...... ... ....  ....... ....  |  module in the system
             ...... ... ....  ....... ....  |  stack.
             ...... ... ....  ....... ....  |

nnnn MODULES IN STACK space BYTES SPARE
SYSTEM STACK MODULE TYP USER  ENTRY   SIZE
             module typ user  address size
             ...... ... ....  ....... ....  |  One such line for each
             ...... ... ....  ....... ....  |  module in the user
             ...... ... ....  ....... ....  |  stack.
             ...... ... ....  ....... ....  |

nnnn MODULES IN STACK space BYTES SPARE
```

Figure 7.5 - The Diagnostic Report

Where:

check-type is the type of program check that occurred, for example EXIT or OVERFLOW. See UM-A.

code is the exit or stop code, and is only present following a $91 TERMINATED message. See UM-B or UM-C.

location-1 is the address of the Global Cobol instruction at which the error was detected.

system indicates the type and serial number of the system on which the error
serial occurred. This line is omitted when the report is displayed on the screen.

program is the program-id or command name of the last module loaded.

accumulator this is the value loaded into the accumulator when the error occurred.

location-2 is the address of the last transfer of control instruction to have been executed before the error occurred.

operation indicates the last file processing statement executed (e.g. OPEN OLD, READ
file-id NEXT, CLOSE ... TRUNCATE). The index will only appear for a DMAM file
index or SpeedBase file accessed using SPAM. The line is omitted if no such statement has been executed since the previous ready prompt or main menu display.

version number specifies the Global System Manager version and release

name is the first five characters of a CALLed entry name or PERFORMed section name appearing in the control path. This will be blank if the module containing the entry name or section name was compiled with the NTR option.

location-3 is the address of an EXECed or CALLed entry point, or a PERFORMed section or paragraph.

location-4 is the address to which control would return were an EXIT statement issued at this level. That is it is the address of the statement following the EXEC, CALL or PERFORM that passed control to location-3.

module is the program-id of a module loaded onto a stack, either by the Global System Manager LOAD customization instruction, or by the LOAD$ system routine described in the Global Cobol Assembler Interface Manual. Modules on the user stack with names $JOBhhhh indicate that the program is running under job management.

typ is P for program modules, or D for data modules.

user is the number of the user who loaded the module onto the stack. Temporary entries on the user stack have the user number suffixed with [T]. If the module was loaded on the system stack as a result of LOAD customisation the word SYSTEM will appear. If the module has been unloaded, but the space cannot be released due to a subsequently loaded module, the word FREE will appear.

address is the address of the entry point of a module on a stack.

size is the size of a module on a stack.

space is the free space in bytes remaining on the user or system stack. For the user stack, this free space represents the currently unused part of the user area.

7.6 Inspecting and Modifying Variables

The I and M instructions described in this section allow you to inspect and modify variables. If you are using symbolic debugging you need only refer to a variable by name for debug to display it, and allow you to modify it, according to the picture clause information you supplied when the program containing it was compiled. In the case of groups, subgroups, file and map definitions, which have no picture clause, debug treats them as PIC X items whose length is the sum of

their constituent fields. However, it is normally better to use the H format to display the fields of such a group in hexadecimal.

Items in the linkage section can be referred to symbolically, but since they are based dynamically it is essential that the base has been set up previously. For example, the operands of the USING clause of an ENTRY statement cannot be inspected until the ENTRY statement has been executed.

If you inspect an entry name, section name or paragraph name, then the first two bytes of the statement at that location are output (in hexadecimal). The first byte will be odd if the trap flag is set on the statement, and even otherwise.

Both the I and M instructions can be followed by BACKTAB or TAB to step backwards or forwards.

7.6.1 I - Inspect a Variable

To inspect a variable you key the I instruction in response to the command prompt. If you have established a symbolic debug table by identifying the relevant compilation in a previous N instruction, you can use the symbols in defining the address:

```
I address [format]
```

The symbol table of a Global Cobol compilation listing contains, in addition to the location, hhhh, of each symbol, a compressed form of picture clause which can be used to deduce the format of a variable.

7.6.2 I - Examples

Subroutine SASUBR, linkage edited to begin at 1CE4, is passed a parameter area which contains the field PAPTR. This in turn addresses a 16-byte data area which you wish to examine. The parameter area is of course defined in the linkage section.

If the symbols in SASUBR have been introduced by a previous N instruction, proceed as follows:

```
Command I @PAPTR H
0648 (0648+0000) 4141 5320 0000 0102 0001 FFFF FFFF 00FF
Command
```

Alternatively, when no symbolic information is available you will have to consult the compilation listing's **symbol table** for PAPTR. It appears as:

```
PAPTR 02 P 001A(01E4)
```

This indicates that the pointer to the linkage section group is at location 1E4 relative to the start of SASUBR and PAPTR is offset 1A bytes from the origin of that group. You can now examine the data area as follows:

```
Command 1 1CE4                          (base SASUBR)
Command 2 @1,1E4                        (base group)
Command I @2,1A H                       (display 16 bytes)
0648 (0648+0000) 4141 5320 0000 0102 0001 FFFF FFFF 00FF
Command
```

You should note that linkage section group pointers are set up at run-time by ENTRY or BASE statements, and therefore a linkage section item cannot be addressed using the @ prefix until one of these statements has been executed for the linkage group which is to be examined. Similarly, a based area should not be referenced before its pointer has been initialised.

The base address of program SA100, linkage-edited to begin at the location immediately following the debug area, is #500. The compilation listing shows that:

- PNTR, a PIC PTR variable is at location 10;
- STRNG, a PIC X(8) variable is at location 12A;
- TOTAL, a PIC 9(6) COMP variable is at location 156;
- RATE, a PIC 9(4,2) COMP variable is at location 2AE;
- TODAY, a PIC DATE variable is at location 2CD.

If the symbols of SA100 have been established in the debug table by a previous N instruction, these variables can be examined using the following dialogue:

```
Command I PNTR
132A (0500+0E2A)
Command I STRNG
"ABCDEFGH"
Command I TOTAL
181204
Command I RATE
23.09
Command I TODAY D
23/03/1988
```

Alternatively, when no symbolic information is available, you have to supply the hexadecimal locations and attributes:

```
Command 1 ,500
Command I 1,10 P
132A (0500+0E2A)                              (pointer)
Command I 12A X8
"ABCDEFGH"                                    (string)
Command I 156 C6
181204                                        (integer)
Command I 2AE C4,2
23.09                                         (fixed point)
Command I 2CD D
23/03/1988
```

Note how, whenever a pointer is inspected, its hexadecimal value is displayed followed by base and offset information in brackets.

To examine the next/previous block of memory you only need to key TAB or BACKTAB respectively to the next command prompt.

For a hexadecimal display only the number of bytes that can fit in the scroll area will be displayed.

7.6.3 M - To Modify a Variable

To modify a variable you key the M instruction in response to the debug prompt. If you have established a symbolic debug table by identifying the relevant compilation in a previous N instruction, you can key the short name of the variable you wish to modify in an instruction of the form:

```
M address [format]
```

The instruction operates exactly as Inspect, described above, except that once the value has been displayed you are allowed to edit it.

When modifying values in **hexadecimal format** you can move around the area using the cursor movement keys and you can switch from the hexadecimal to the ASCII area using the tab key. You will only be able to edit the amount that fits into the scrolled area. To modify the next section of memory you only have to key TAB to the next command prompt or for the previous section BACKTAB.

To modify a **pointer**, simply key one to four hexadecimal digits giving the desired new value. (This facility can be used to set any two adjacent bytes of storage to any specific value).

To modify a **character or display numeric variable** simply key the ASCII characters required. If you key less characters than displayed previously, your input will be padded with rightmost blanks.

To modify a **computational variable**, key the new number as appropriate. It can be signed and have a maximum of 18 significant digits of which no more than 15 may precede the decimal point and no more than 7 may succeed it, depending on the format supplied.

To modify a **PIC DATE** variable just key the date in dd/mm/yy form.

To modify a **floating point** number just type in the new floating point number.

7.6.4 M - Example

Program SA100 has failed with a numeric conversion program check at location 1C62. It was linkage edited to start at location #500, the first byte following the area used by debug.

Examining the listing you find that a MOVE statement at listing location #1762 has failed. This is because of an erroneous value in the PIC 9(4) display numeric variable ALPHA whose listing location is 8E.

If the symbols in SA100 have been established in the debug table by a previous N instruction (which has the side effect of setting the base at #500) then the following dialogue shows how you might correct ALPHA:

```
Command M ALPHA   1111
Command
```

Alternatively, when no symbolic information is available, you have to supply the hexadecimal locations and attributes of ALPHA after having established the base:

```
Command 0 500
Command M 8E X4   1111
Command
```

in both cases the spurious value of AA** is initially displayed and overtyped with the correct value of 1111.

7.6.5 X - Index the Next I or M Instruction

The X instruction is provided to help you use the I and M instructions to operate on variables within tables or repeating groups established by means of the Global Cobol OCCURS clause.

Note, however, that you cannot use the X instruction in conjunction with the symbolic form of the I and M instructions to index a variable which is greater than 255 bytes in length, or index a field within a repeating group whose entries are more than 255 bytes long. This limitation is due to symbolic debug table size restrictions. If you mistakenly attempt an X instruction in these circumstances the result will be unpredictable.

To **select a particular occurrence** of a variable, key the instruction:

X *index*

in response to the debug prompt. The index should be a decimal integer between 1 to 9999 inclusive. This will cause the next I or M instruction, which should define the first occurrence of the variable, to actually process the occurrence specified by the index.

To select a range of occurrences, key the instruction:

X *index1 index2*

in response to the debug prompt. Both the indices specified must be decimal integers between 1 and 9999 inclusive, and index2 must be greater than index1. The next I or M instruction, which should define the next occurrence of the variable, will actually process the occurrences in the range defined by the indices.

In the case of an inspect instruction each entry value will be output and only when the entire range has been displayed will the next debug prompt appear.

In the case of the modify instruction each entry value will be displayed on a new line followed by a modify prompt. You can then either change the value as described in section 7.5.3 or key <CR> to leave the value unaltered and proceed to the next entry. You will obtain the next debug prompt only when the entire range of values has been processed in this way.

The X instruction only affects the very next inspect or modify instruction. A new X instruction must therefore be keyed for every I or M that is to be indexed.

Note that you can inspect (or modify) a single indexed variable by using the form:

```
I address(index) [format]
```

where index may be either a number in the range 1-9999 or the name of a computational variable present in the currently attached symbols.

7.6.6 X - Example

A table within program SA100 is defined as follows:

```
01    TABLE OCCURS 20
  02  TANAME      PIC X(8)
  02  TACOUNT     PIC S9(4) COMP
```

The compilation listing shows that the first occurrence of TACOUNT is at location 8E relative to the start of the program. SA100 has been linkage-edited to begin at location #500, the byte immediately following the debug area.

The following examples show you how the index instruction might be used to modify the 3rd, 5th and 6th occurrences of TACOUNT.

If the symbols in SA100 have been established in the debug table by the previous N instruction, proceed as follows:

```
Command X 3 6                          (select items 3 to 6)
Command M TACOUNT
-1104 108                              (3rd entry changed)
208 <CR>                               (4th entry unchanged)
13 0                                   (5th entry changed)
1287 1                                 (6th entry changed)
Command                                (range processed)
```

7.6.7 Y - Establish Length of Repeating Group

The Y instruction is provided to allow you to inspect variables inside a repeating group when no symbols are available.

To establish the length of a repeating group, key the instruction:

```
Y length
```

in response to the debug prompt. The length should be a decimal integer in the range 1 to 32767. The instruction must be keyed immediately following an X instruction, and immediately before an I or M instruction which specifies a hexadecimal (not symbolic) address. If used under any other circumstances it has no effect.

For example, if you wished to inspect TACOUNT in the example in 7.6.6, but symbolic information was not available, you would have to use the following sequence of instructions:

```
Command X 3 6
Command Y 20
Command M 8E C4
etc, etc.
```

Note that if you do not use the Y instruction in the above example, $DEBUG will have no way of telling the length of the group, and you will get erroneous results.

7.6.8 S - Search for String

To find the location of a particular string in memory, you key the instruction:

```
S "string"
```

or:

```
S hex
```

where hex is the hexadecimal value of the string. The whole of the partition, except for the user stack and debug area, is then searched for occurrences of the string. Each time the string is found, its start address is displayed on a new line. For example, to find occurrences of the string "123":

```
Command S "123"    9316 (0500+8E16) 3121 (0500+2C21)
2AB5 (0500+25B5)
Command
```

The instruction is particularly useful if you do not have an up-to-date listing of the program, or if you want to locate a data item allocated from free memory.

7.6.9 ? - Display Address as Base and Offset

To convert any address to the format address (base+offset), you key the instruction:

```
? address
```

For example:

```
? @PTNR          0501(0500+0001)
```

7.7 Resuming Programs from Debug

If a break, trap, exit or program check occurs debug will be re-entered. You can then examine or alter the program and possibly restart it using the resume instruction. A typical sequence might be:

- Run $DEBUG which loads the subject program;
- Use N to obtain symbolic information;
- Set traps at significant points using T;
- Modify data for testing purposes using M;
- Resume program using R (debug is re-entered when a trapped instruction is encountered);
- Inspect variables using I;
- Set additional traps using T;
- Resume subject program using A or R.

A and R are employed to resume the program which was active when debug was entered. The program will be resumed in the target partition but the debugger will not switch you to that partition unless input from the screen is required by the program being debugged. If this does happen then you will not again be switched to the debug partition unless there is an error or trap. While the program is being executed the $DEBUG will display the message:

```
Key <CTRL G> for command prompt
```

If you want to quit from the debugging session any time during the execution you may key <CTRL G> to get to the command prompt followed by the Q instruction as explained in section 7.1.8.

If you do key <CTRL G> to get the command prompt the following will appear on the status line:

```
OPERATOR INTERRUPT
```

Caution must be taken in performing instruction other than Q at this stage, because the user partition is still active, and the program being debugged may still be executing.

7.7.1 R - Resume a Program from Debug

To resume a program which has entered debug for any reason you must key the instruction:

```
R
```

or:

```
R address (relativised)
```

in response to the debug prompt.

The first format, R, is used to continue at the interrupted instruction following a trap or break.

The second format will normally only succeed in resuming the program successfully if a program check occurred because of invalid data and it is possible to correct this using the M (modify) instruction. If this is possible you must resume by specifying address as the location of the failing **statement**. This location may not correspond exactly to the program check location because a single statement often expands into a number of internal instructions, any of which might fail. You must resume the program on a statement boundary at the same level of control as the failing instruction or unpredictable errors may arise.

7.7.2 R - Operating Notes

If you have removed an application volume to mount SYSRES so that the debugger can be loaded, you **must** replace your disk before executing the R instruction, or there is a risk that the system volume will be corrupted if the application writes records to it.

7.7.3 R - Examples

The following dialogue takes place when you set a trap at location 104E of program SA100 in order to inspect certain variables before continuing the program:

```
GSM READY:$DEBUG
Partition to be debugged 2

Program to be debugged SA100
Overlay to be trapped SA100
Command 0 500
Command T B4E                        (set trap, explained later)
Command R                            (execute program previously loaded)
.........
.........                            (dialogue from SA100 carried
.........                             on in partition 2)
TRAP AT 104E (0500+0B4E)             (appears on the header line)
```

```
Command: etc etc                    (instructions to inspect data)
Command R
.........
.........                           (more dialogue from SA100)
.........
PGM CHECK 11 AT 218C (0500+1C8C)    (appears on the header line)
Command
```

At the end of this you have unexpectedly entered debug again with program check 11 (overflow). If you can determine the cause of the problem you may be able to patch data fields and continue testing the program. For example, the following dialogue resumes program execution at location 2184 (listing offset 1C84) which you have determined to be the location of the failing statement, 8 bytes earlier than the instruction which caused the program check:

```
Command R 1C84
```

Note that if the symbols in SA100 have been established in the debug table by a previous N instruction, then you can use a symbol as the operand of the R instruction. This would allow you to resume at a particular section or paragraph. For example:

```
Command R BA-OPE
```

resumes the program at the beginning of the section named BA-OPEN-FILES.

7.7.4 A - Advance to a Transfer of Control Instruction

When a program has entered debug for any reason, you may single step it by keying:

```
A
```

This causes execution to proceed to the next transfer of control instruction or memory modification instruction. This is typically equivalent to executing a single Global Cobol statement.

When the next transfer of control instruction or memory modification instruction is reached, this is confirmed by a header line message of the form:

```
ADVANCED TO location
```

For example:

```
Command A <CR>
ADVANCED TO 0536 (0500 + 0036)
Command
```

You may want to single step through your program for a while, to observe the route of execution. To get into this continuous advance mode you must key the following to the command prompt:

```
A<CTRL B>
```

To return to the command prompt you must key <CTRL G>.

7.7.5 A - Operating Notes

Some complex statements (such as DO FOR) contain a number of transfer of control and memory modification instructions. In such cases you will have to Advance a number of times to fully execute the statement in question. A few Global Cobol statements generate no such instructions and will therefore not cause the program execution to halt when Advancing.

Advance will interrupt execution on the next statement, even if it is in a Global system routine or subroutine. You should normally Trap the program after the exit point of the subroutine as it may take some time to Advance through all the statements it contains. Notwithstanding this, Advance will not interrupt execution of privileged code (used extensively in the Global System Manager monitor and in some Global system routines such as LOAD$).

The continuous advance mode will stop execution if a program error or trap occurs.

7.7.6 A - Example

Your program contains a DO-loop and you wish to examine the first iteration of the loop in detail. The start of the DO-loop is labelled START, and the loop count is held in COUNT. You proceed as follows:

```
GSM READY:$DEBUG

Partition to be debugged:2
Program to be debugged PROG Overlay to be trapped PROG      (load program)
Command N
Compilation name <CR> Program-id <CR>     (introduce symbols)
Command T START                           (set trap at start of loop)
Command R
...
...
TRAP AT 050C (0500+000C)                  (trap at start of loop)
Command I COUNT 1                         (check first iteration)
Command A
ADVANCED TO 0512 (0500+0012)              (advance through expected path
Command A                                 in the loop)
ADVANCED TO 051E (0500+001E)
Command A
ADVANCED TO 0524 (0500+0024)              (advance to unexpected
                                          instruction in loop, inspect
                                          variables to find what may have
                                          caused this and modify variables
                                          to rectify the error)
Command R 1E                              (resume program to avoid this path)
...
...
TRAP AT 050C (0500+000C)
Command I COUNT 2                         (check second iteration)
```

7.8 Traps

Debug's T and C instructions, explained below, may be used to set or clear the trap flag associated with any Global Cobol procedure division statement. The trap flag may be set on any number of statements within the subject program.

The interpreter detects when a program attempts to execute an instruction with the trap flag set, and causes the debugger to interrupt it with:

```
TRAP AT location
Command
```

You can then use whatever debug instructions you require, and normally resume the program at the indicated location, by means of the R instruction. A trap remains in force until either the
subject program is reloaded, or it is explicitly cleared using the C instruction.

The O instruction described in this section extends the concept of a trap by allowing you to request that debug is entered whenever a named overlay is loaded.

The V instruction also traps when a value of a variable reaches a set condition.

7.8.1 T - Trap a Statement

To set a trap on a statement you must key the instruction:

```
T address (relativised)
```

in response to the debug prompt. The trap flag is then set and the debug prompt again displayed.

By using the T instruction repeatedly, any number of traps may be set. For example, the dialogue:

```
Command N Program-id:<CR>                  (base address 0500)
Command T BA-OPE  06EA(0500+01EA)
Command T 171A    1C1A(0500+171A)
Command T 1C20    2220(0500+1C20)
Command
```

firstly names the currently loaded main program as the compilation whose symbols are to be used, and then sets a trap on BA-OPE, so that the first statement of the section named BA-OPEN-FILES is trapped. Subsequent T instructions are used to trap the statements at listing locations 171A and 1C20.

7.8.2 T - Operating Notes

If you set a trap on an ON OVERFLOW or ON EXCEPTION statement and the exception or overflow condition arises, then the message:

```
PGM CHECK 11 AT location
```

or:

```
TERMINATED - EXIT code
```

corresponding to that condition, rather than the normal TRAP message, will appear on the header line.

This indicates that the overflow or exception condition has been cleared which means that if you attempt to resume a program terminated in this way it will execute as though the condition had never occurred. However, if the ON OVERFLOW or ON EXCEPTION statement is met and no overflow or exception condition prevails, then the trap is handled in the normal way.

7.8.3 C - Clear the Trap Flag

To clear a trap on a statement you must key the instruction:

```
C
```
or:
```
C address (relativised)
```

in response to the debug prompt. The trap flag is then cleared and the debug prompt again displayed.

The first form of the instruction, C, clears the trap flag of the statement which last caused debug to be entered. The second form clears the trap on the statement at the location specified. By using the C instruction repeatedly, any number of traps may be cleared. For example, the following dialogue might take place after the traps introduced in the description of the T instruction in section 7.7.1 had been set, and would have the effect of removing all of them:

```
Command R                                    (resume program with traps set)
TRAP AT 1C1A       (0500+171A)               (appears on the header line)
Command C BA-OPE                             (clears T BA-OPE)
Command C (0500+171A)                        (clears T 171A)
Command C 1C20     2220(0500+1C20)           (clears T 1C20)
```

7.8.4 C - Operating Notes

When you use the R instruction to resume execution of the subject program following a trap, it will **not** cause the trap to be immediately repeated because the interpreter always ignores the trap flag when it is set on the very first instruction following a resume. This prevents you having to use the C instruction unnecessarily simply to proceed with your program.

7.8.5 O - Trap the Loading of an Overlay

To force a trap whenever a particular overlay is loaded (by CHAIN, EXEC, LOAD, RUN, CALL QLOAD$ or CALL LOAD$ statement) you key the instruction:

```
O program-id
```
or:
```
O ?
```
(to trap on every overlay load)

in response to the command prompt. Once the overlay trap has been set the command prompt is again displayed.

Only one program at a time can have a overlay trap set for it, and the latest O instruction determines the program affected. You may key the instruction:

```
O
```

with no parameter (type a space in order to clear the default) to clear the overlay trap. The trap is also automatically cleared whenever you quit the debugger.

The trap takes place as soon as the program identified by the O instruction has been loaded but before it is entered. This enables you to set additional traps using the T instruction. When you have finished setting traps, or otherwise modifying the program, you can resume execution by using the R instruction.

7.8.6 O - Example

Program SA100 invokes overlay SA103 by means of an EXEC statement. You wish to set traps in this overlay before it is executed. The following dialogue takes place:

```
GSM READY:$DEBUG
Program to be debugged SA100 Overlay to be trapped SA103
Command R
OVERLAY TRAP ON SA103                    (appears on the header line)
Command
```

7.8.7 O - Operating notes

You should note that overlays loaded by pre-V6.2 LOAD$ and QLOAD$ will not be trapped using the O ? command.

7.8.8 V - Value Trap

The value trap instruction allows you to force a value trap when the value of a variable satisfies the condition you specify. To set a value trap you key the following instruction to the command prompt:

V *address* [*format*] *condition value*

The trap takes place at the first memory modification instruction where the value at address satisfies the criteria given by the condition and value. The address and format of the variable are as described in section 7.1. However, the format must not describe a variable longer than 8 bytes, as this is the maximum that can be compared (so the default format is H8 and not H16).

Normally the instruction on which the trap occurs will be the one which has changed the variable to the specified value. However, if you are testing programs that use asynchronous assembler process such as communications routines, the variable may have been changed by such a process. Also the value trap will not be realised during the execution of privileged code (as used in the Global System Manager monitor and by certain system routines) and in such cases the value trap will be realised at the end of the first memory modification instruction, after control returns from the monitor or system routine in question.

The conditions allowed to compare the variable to the value are:

<>, >, <, =, GE, LE, GT, LT, EQ, NE

the condition <> or NE is allowed in order to trap if the value of the variable changes in any way. In this case a value is not required.

If the variable is specified as a computational field, the test comparing the variable with the value will be signed, otherwise it will be unsigned.

The value to be compared with must be in the same format as the variable specified. If you specify a hex format then you must supply a hex string (without the preceding "#"), if you specify a computational format you must supply a numeric value etc. A character string should be specified between double quotes.

Only one value trap can be set at a time. To explicitly clear the trap you may key the instruction:

```
V 'space'
```

The 'space' is keyed in order to delete the default value set for the last value trap.

The value trap will be **automatically** cleared once the conditions have been satisfied and the trap has happened. It is also cleared when you quit from the debugger.

7.8.9 V - Example

Consider the example given for the advance instruction in section 7.7.10. We can trap at the start of the loop and check the first iteration as shown. We then also want to check the 20th iteration. To get to the twentieth iteration we can set a value trap as follows:

```
Command V COUNT EQ 20                 (set trap)
Command R                             (resume program)
...
...
VALUE TRAP AT 050C (0500+000C)        (trap after COUNT has reached 20)
Command I COUNT 20                    (check 20th iteration)
```

7.9 Listing file instructions

The following set of instructions are provided to allow greater flexibility when examining source files.

7.9.1 E - end or start inspecting a file

The E command automatically closes the listing file currently being inspected if there is one. It then asks you for another file to be inspected:

```
Command E Source file:file Unit:unit
```

where file is the file to be inspected and unit is the unit it is on. If you do not key a prefix on the file name "L." prefix will be taken by default. If the unit on which you are inspecting the file is a spool unit and there are more than one file of this name it will ask you for the file you want the most recent file first.

```
Command E Source file:L.SA100 Unit:$PR
Listing file 865SUE /4 OF 20/04/88?:N         (Not this one)
Listing file 863SUE /4 OF 19/04/88?:Y         (this one)
```

If you do not want to inspect any file you just key <CTRL A> to the source file prompt.

Once the listing file has been found you are asked:

```
Track addresses in listing file?
```

If you key Y to this, then the listing (displayed in the F3 window), will track the instructions within the program as they are carried out. Note that tracking the listing file imposes an overhead on re-entering the debugger after a program check, so you may want to disable it to speed execution.

7.9.2 L - get line number

To inspect a specific line in the listing file you key:

```
Command L number
```

where number is the line number you want to inspect.

The window will then display the source file starting from the line specified. If this line number is greater than the number of lines in the file, the end of file message will be displayed.

7.9.3 F - find source line string

This instruction will find the string passed if it exists in the listing file. You key:

```
F string
```

The F instruction will retrieve the next line in the listing file containing the string. If the string is not found when the end of the listing file is reached, it will return to the start of the file and continue searching until the present line in the file is reached. If the string is not found then the part of the file displayed on the screen will not change.

7.9.4 G - get source line containing address

This instruction can only be used on "L." files produced by the compiler. It obtains the line in the listing file containing the 2 byte hex address passed to it:

```
G hex
```

As with the F instruction the G instruction will search to the end of file and then return to the start.

7.10 Templates

Templates allow you to keep details of the current debugging session. These details are:

- the program being debugger (see 7.1);
- the initial overlay to be trapped (see 7.1);
- the base registers (see 7.4.2);
- variables to be shown in the symbol window (see 7.2.1);
- file under inspection (see 7.9);
- the window of application last selected (see 7.2).

Templates are provided so that these details do not have to be entered again when debugging the same program in a different debugging session. They are stored in a data library called $$DEBUG which will be created on the SYSRES unit.

7.10.1 K - keep template

The K command allows you to keep the current debug template. It asks you to confirm the name of the program and the initial overlay trap:

```
Command K Program to debugged:Program Overlay to be trapped:Overlay
```

Record name:*Record* Title:*Title*

If the data library is full you will get the option of deleting other templates. If you do not wish to do so you will be unable to store the template.

7.11 Alternative debugging modes

The debugger allows you to inspect and modify both SCF variables. To do this you need to enter a different mode of debugging.

7.11.1 P - SCF Debugging mode

The P command puts you into the mode where you can manipulate scientific variable using SCF programs. It can only be run when you are in the scientific variable window, and there are scientific variables available in the program you are debugging. To enter SCF mode you key:

```
Command P
scientific program
```

You will then be prompted for the scientific programs you want to apply. After each scientific program you apply the scientific variables displayed in the window will be updated accordingly. Once you have finished manipulating the scientific variables you key <ESC> to return to the command prompt.

7.11.2 P - Example

Your program may fail because of a syntax error, for instance COMPUTE "a = sqw b" as follows:

```
GSM READY:$DEBUG
Partition to be debugged:2
Program to be debugged SQROOT
Overlay to be trapped SQROOT
OVERLAY TRAP ON SQROOT (header line)
Command O ,0500
Command R
EXIT CODE 25401 AT 0546 (0500 + 0046) (invalid syntax)

(look at the listing file to find the syntax error
COMPUTE "a = sqw b " tracked; switch to the SCF window
to key the correct syntax)

Command P
scientific program
a = sqr b
<ESC>
Command R (resume the program with the correct values)
```

8. Record and Playback Using $RCP

The $RCP command is used in conjunction with <SYSREQ Q> (the record/playback system request) to enable you to record keystrokes typed on the keyboard and to subsequently play them back to repeat the original processing. The repeated playback facility can be used to test again programs which have had amendments applied. The record script is held in a text file which can be edited to account for any dialogue changes to the program being tested.

8.1 Recording Keystrokes

To record keystrokes you first key <SYSREQ> Q at the required point in the processing (normally this will be from some well defined commencement point, such as the system main menu). You can choose one of two options (record or playback) and you should select the record option.

You must now switch to a different partition, and run the $RCP command which manages the actual recording of the keyed data. You are first asked to identify the partition which you wish to record as follows:

```
GSM READY:$RCP
Record/Playback Maintenance

Target Partition:
```

You must specify the number of the partition in which <SYSREQ> Q is run $RCP will check that this partition is expecting to be recorded, that it resident in memory and otherwise suitable to be recorded. If there is some problem an error or warning message is displayed, which will suggest any appropriate recovery action you might take.

Having selected the partition to be recorded, $RCP will prompt for the filename and unit of the script file to be created. If the file already exists the prompt:

```
FILE ALREADY EXISTS - DELETE?:
```

will appear. If you reply Y then the existing file will be deleted and replaced by the new file you are creating. If you reply with N then you are asked if you wish to extend the existing file. If you elect not to extend the existing file, you will be prompted again for the script file to be created. If you do wish to extend the existing file then you will be asked to select a filename and unit for the new script file to be created, and the contents of the existing file will be copied into that file and will have the current record session appended to it.

[Note that it will be most common to use this latter facility to extend an existing script after playing it back to position the dialogue in the appropriate place. This facility is most useful when it has been necessary to break off creating a script for some reason.]

Having selected the new script file to be created you are returned to the partition in which <SYSREQ Q> was run to begin typing keystrokes to be recorded.

8.1.1 Keystrokes Recorded

The following screen processing is recorded by the record/playback programs:

Character Accepts
All character accepts, including use of the ACCEPT verb, calls to ACCE$ and PASS$, screen formatting accepts managed by the MAPIN statement, calls to BASEL$ and accepts generated by SpeedBase are recorded.

Single Character Input
Characters input by CHAR$, CHARX$ and single character handling within SpeedBase Presentation Manager are recorded.

BELL Statements
To monitor correct operation of script files on playback, all incidences of the BELL verb, including those generated internally by Global System Manager, are recorded.

System Requests
Global System Manager system requests (<SYSREQ> A, C, D, E in particular) are recorded specially, as is the dialogue used to run the system request.

The following processing is not recorded, and must therefore not be included in a playback script:

Accept operations
Any Global System Manager command program or Global application not using the standard accepts as described above will not be recorded. This means, for example, that $T and certain other terminal test programs cannot be recorded, along with Writer, Planner and Finder and Reporter.

Interrupt Keys
Global System Manager system requests which do not run a program (for example <SYSREQ> L, R, Z), together with the BREAK and <CTRL G> keys are not recorded. The use of such keys during creation of a script file is strongly discouraged, especially if their use might cause variations in the dialogue of the program being recorded.

Variable Dialogue
If a program contains variable dialogue, which is sensitive to the keying of an interrupt key, such as <CTRL G>, or to the presence of type-ahead, it is very unlikely that this can be successfully recorded. The activation of variable dialogue during creation of a script file is strongly discouraged, and any programs which respond in a significant way to the presence of type-ahead may not be capable of being recorded.

SpeedBase Help Function
The SpeedBase help function used within an accept in SpeedBase Presentation Manager will not be recorded. The use of the help function is discouraged as its use may change the screen display.

8.1.2 Annotating the Script File and Data Checking

At any point during the recording you may make annotations to the current script file by changing to the partition where $RCP is being run. Here you will see a list of the lines which have been created in the script file, and you may enter maintenance mode by keying <CTRL G>. This will take you into standard text edit ting mode (as in $TED) and will allow you to modify up to the last 50 lines created. You may move these lines around, making changes if required and adding comments (usually to describe the purpose of the line for future reference and maintenance) - spurious lines may be deleted (such as the response to an inadvertent break prompt) and checking of screen data may be established.

To check data you should position the cursor at the appropriate point in the script for checking to be performed. Checking will take place before the processing of any subsequent keystrokes, and the check should usually be inserted at the end of the current script). You then key Command (F3) followed by the number 1, 2, 3 or 4 to select one of the check windows (initially positioned at the top left, top right, bottom left and bottom right corners of the screen respectively). The position of each window is remembered during this session of recording, so you may establish default check windows over areas of the screen which are likely to be of interest. You can now position the selected window using the cursor keys, and when it is in the correct position key F1. The size of the window can now be changed using the cursor keys. Keying F1 again to return to positioning mode if necessary). Note that the data checked is **all** of the characters covered by the displayed box, including those covered by the box edges.

When the appropriate data has been selected for checking key <CR> to cause CHECK statements to be generated in the script file. If you decide that checking is not, in fact, required key <ESC> to return to script editing.

Once changes to the script are complete you should key <ESC> to exit from editing, and you will be returned to the recorded partition to enter more keystrokes.

8.1.3 Finishing Record Script Creation

When you have finished creating your record script, you key <SYSREQ> Q again. This stops the recording process and transfers you to the partition running $RCP so that you may make any final modifications to the script. When the script file is satisfactory, key <ESC> and $RCP will write out the remaining lines of script to the file and return to the main menu.

8.1.4 Editing a Script File

Once created a script file may be edited using any text editor ($EDIT, $ED, even $TED if it is sufficiently small), although obviously care should be taken with any changes made. The script syntax definition is described later.

The most usual reasons to edit a script file are to correct changes in the dialogue of the program run, and to establish a linked set of scripts by judicious use of the CHAIN and EXEC statements. In the latter case care should be taken to remove extraneous SYSREQ 'Q'

statements from the individual record files as these will terminate playback if they are encountered.

8.2 Playing Back a Script

To initiate playback of a script file you key <SYSREQ> Q, and then select the playback option. You must then transfer to another partition and run $RCP, specifying the original partition as the one into which playback is to be directed.

You are asked to select a playback filename and unit, and to indicate whether this is a simple playback or debugging test. In the former case you are returned to the original partition and playback commences, whereas in the latter case you are asked to specify a breakpoint in the playback (see below) first.

Once playback has started it will control the partition until it has run its course. The playback will be stopped by one of the following events:

End of Script
At the end of a script, or set of script files executed using EXEC and CHAIN statements, the playback is terminated and the partition once again accepts operator input.

SYSREQ 'Q'
If a line in the script asks for SYSREQ Q to be loaded this terminates the playback and the partition once again accepts operator input.

Failed CHECK
If a CHECK does not match the contents of the screen this will be reported and the playback terminated with an error.

Syntax Error
If a line in the script file contains a syntax error this will be reported and the playback terminated with an error.

BELL (Program Error)
Use of the BELL verb usually indicates a program error. If a BELL verb is executed and there is not a corresponding BELL in the script (indicating that the BELL arose when the script was originally created, and is hence presumably dealt with by subsequent dialogue) this will be reported and the playback terminated with an error.

Accept Mismatch
If the wrong kind of accept is encountered (a character accept when a CHAR$ call was expected, or vice versa) then this will be reported and the playback terminated with an error.

Additionally the execution of the script may be interrupted by the operator.

8.2.1 Interrupting and Debugging Scripts

At any stage during playback you may switch to the partition running $RCP, where a list of lines read from the script file is displayed,

and key <CTRL G> to interrupt. Playback will also be interrupted when a breakpoint is encountered.

When playback is interrupted, you are prompted as follows:

```
Key Breakpoint, Single Step, Q to quit:
```

If you key <CR> then execution resumes (but note that any interruption of execution, even by operator keying <CTRL G>, clears any outstanding breakpoints). The following options are also available:

Breakpoint
You may specify a breakpoint by selecting a script filename, unit and line number. Execution will be interrupted just before that line of the named file is processed. This is usually used to locate problems in a set of playback scripts.

Single Step
This causes the script to be run for the next processing statement (CHECK, or any accept or system request load), and then execution is suspended again until the operator keys <CR> for the next step. Keying <ESC> exits single step mode and returns to the interrupt prompt. Note that as there is quite a bit of partition switching involved in single step operations, it is normally most convenient to set a breakpoint in the vicinity of a piece of suspect code and then single step through it.

Quit
If the debugging session is finished and there is no point in attempting to run the remainder of the script, then the operator may quit from the playback file, which will leave the partition running the playback dialogue exactly where it was.

8.3 Record or Playback Failure

In some situations, when $RCP terminates abnormally, the partition which is having its responses recorded, or into which data is being played back, may be left in a state where nothing can be keyed. If this should happen simply run $RCP again, specifying the appropriate partition, and the following message should appear:

INVALID PLAYBACK STATE *x* - CANCEL *mode*?:

Where x will be a number identifying the failing operation type (of internal interest only) and mode will be one of RECORD or PLAYBACK as appropriate. If you reply Y to this prompt the record or playback process will be disconnected and the partition returned to accept operator dialogue from the console in the normal way.

Please bear in mind that while $RCP is editing the record file you cannot key responses in to the recorded partition - to restart recording in this case move to the partition running $RCP and key ESCAPE to exit from edit mode.

8.4 Requirements for Running Record and Playback Software

The record and playback facilities are only available under Global System Manager V8.0 and later. In addition programs using CHAR$ must

have been linked with V8.0 or later CHAR$, and programs written using SpeedBase must have been compiled with V8.0 or later SpeedBase Development System and be using the appropriate version of SpeedBase Presentation Manager.

8.5 Structure of the Script File

The script files are ordinary Global System Manager text files, which are created by $RCP with a default Q. prefix. They are processed in a line-oriented way by the playback software, which checks for a limited syntax for each line. Comments may be introduced by use of the '*' or ';' characters (not enclosed within ", ', < or [), and cause the rest of the line to be ignored. Completely blank lines, or those containing only comments, are ignored by the playback software.

The following characters are significant to the playback software, and will cause the remainder of the line to be processed in a particular way:

"	Introduces a character accept string
<	Introduces a CHAR$ or CHARX$ function
[	Introduces a SpeedBase function
'	Identifies a system request load
BELL	Identifies a BELL line
CHECK	Identifies a Check line
CHAIN	Identifies a Chain line
EXEC	Identifies an Exec line

If the first token on a line is one of the key words above, or begins with one of the special characters, it is processed, otherwise it is discarded and the process repeated with the next token.

8.5.1 Chain and Exec Lines

These take the form:

CHAIN *filename unit*

or

EXEC *filename unit*

where filename is the name of a script file to be processed (including any prefix) and unit is the unit on which the file is located (may be a logical unit-id).

The effect of a Chain is to transfer processing to the first line of the file thus identified - the remainder of the current file, if any, is not processed.

The effect of an Exec is to run the identified script, starting at the first line. When the end of that script is reached control returns to the line in the current script following the Exec line. Exec lines may be nested to a depth of at least three calls (exact depth depends on free user area). Note that none of the 'EXEC'ed scripts should contain a SYSREQ 'Q' line as this will terminate playback when it is encountered.

8.5.2 Playback Strings

Strings are used in the playback file in character accept, Check lines and Speedbase accept. These obey the normal rules for string

definition with " characters at the start and end of the string. However, if the 'terminating' " for a string is immediately followed by another " then this is taken to represent an actual " character in the string, and the actual " terminating the string lies somewhere to the right of this.

To take an example, the string **"ABC"** represents the characters **ABC,** but the string **"AB""C"** represents the characters **AB"C.** Similarly the string **"""A"""** represents the characters **"A"**.

8.5.3 Check Lines

A check line has the format:

```
CHECK line,col "check-string"
```

where line and col define the position on the screen where the data to be checked resides, and check-string is a playback string defining the expected contents of the line from that position.

Checks which cover more than one line cause multiple Check lines to appear in the script file, as do checks where the check-string is longer than 50 characters. All Check lines encountered are processed immediately before the following keystroke lines.

8.5.4 PAUSE lines

A pause line has the format:

```
PAUSE nn
```

or

```
PAUSE "pause string"
```

where nn is the number of seconds you wish the dialogue to pause for on playback. The second variation displayed the pause-string on the status line at that point in the dialogue playback, and then prompts, in the playback partition for the continuation of the dialogue.

8.5.5 BELL Line

BELL lines take the form:

```
BELL
```

and serve to identify one incidence of the BELL verb arising in the program(s) being recorded. When the script is played back each BELL generated by the software must be matched by a BELL line in the script file, coming before any keystroke lines or Check lines. Excess BELL lines in the script file will be ignored.

8.5.6 Keystroke Lines

There are five types of keystroke lines. Note that although these are produced with a 'keyword' by the record software, this is purely documentary and plays no part in the analysis of the lines by the playback software.

Character Accept Lines

These are recognised as any line containing a token starting with the " character which is not a Check line. There are two types of character accept lines which have the formats:

"*accept-string*" *<eof>* (ACCEPT statements)

or:

"*accept-string*" [*ch*] *func* (Speedbase accepts)

where accept-string is a playback string which defines the response to the character accept. For basic accepts eof is an optional end-of-field code (to be placed in $$EOF). If no value is supplied then a value of M (<CR>) is assumed. For Speedbase accepts, ch is a single character, or two-digit hex code representing the actual terminating end-of-field value returned by the accept function and func is the internal SpeedBase function number.

For those, relatively rare, cases when the accept deals with more than 50 characters, two consecutive character accept lines may be added together. The first line may not contain an eof
and instead has a single + character following the accept-string. The accept-string of the following line (ignoring an intervening lines which do not contain an accept-string) is added to the end of the first string before being passed to the prompt.

If the length of the total accept built up is greater than the expected maximum length of accept from the program, it is treated as an error and the playback is terminated.

EOFCH$, CHAR$ and CHARX$ Functions
These are recognised as any line containing a token starting with the < character which are not character accept lines. They have the format:

<ch> excep func count

where ch is either a single character or two-digit hex code representing the actual character value returned by the call, excep is the exception number generated, which may be omitted if it is zero, func is the function key number (returned in QT$FUNC) which will be omitted if it is zero, and count is the number of occurrences returned by CHARX$ only, omitted if it was 1.

If a count other than 1 is specified the program receiving the playback must have issued a CHARX$ call, otherwise any one of a CHAR$, EOFCH$ or CHARX$ call will be satisfied by this line.

SpeedBase Functions
These are recognised as any line containing a token starting with the [character. They have the format:

[*ch*] *func*

where ch is either a single character, or two-digit hex code representing the actual character value returned by the function and func is the internal SpeedBase function number.

System Request Functions
These are recognised as any line containing a token starting with the ' character. They have the format:

'*x*'

where x is the letter identifying the system request to be loaded ('_' is used for SYSREQ space).

A system request function is activated when the target partition requests any keystrokes and the next line in the script is a system request function. Subsequent dialogue will be fed to the system request until it exits, at which point the original operation causing keystrokes to be required will be reactivated (in the usual way).

Normally scripts will be created by using $RCP in record mode. While small changes to scripts may be safely made manually, large scale script creation by manual methods is not recommended as the scope for errors is too large.

8.5.7 Programming Note

Where programs insert characters into the type-ahead buffer for any reason, these will be intercepted by the record process and stored in the playback script. Consequently, when running in playback mode any characters inserted into the type-ahead buffer by a program will be discarded, and those present in the record file used instead. Great caution should be used when running such programs as changes in them may not be noticed by the playback software, possibly causing (or masking) errors.

Appendix A – Program Preparation Example

Each Global Cobol system comes complete with a sample program The program demonstrates interactive use of a relative sequential file, allowing you to create new records and then interrogate them by supplying a record number. Naturally it has been made fairly simple in order to keep its listings to a reasonable size. These are extensively annotated in the remainder of this appendix so that you have a reference example of the listings produced by the compiler, linker, cross-reference utility and string search utility.

We only supply you with the **source file** of the sample program (file-id S.SAMPLE) and even this has deliberate errors in it. By working through the procedures described below you will become familiar with the start-up procedure required to initiate Global System Manager, the file utility, the file print utility, the compiler, the text editor, the linker, and the symbolic debugging system, the cross-reference utility and the string search utility.

When you have completed the exercise you will have produced the listings, contained in pages A-8 to A-28, for yourself, and have run the program, obtaining results similar to those shown in the photographs of this appendix. You will also have made a temporary modification to the program using debug.

Initiating Global System Manager
Perform the start-up procedure described in your Global Operating Manual. Sign on and obtain a ready prompt:

```
GSM READY:
```

You can supply the names of the Global System Manager commands or programs you require to run in response to this prompt whenever it appears.

Preparing a Work Volume
Run $V, the volume maintenance utility described in the Global Operating and Global Utilities Manuals, to prepare a work volume. Once you have set this up successfully, copy file S.SAMPLE to it using the COP instruction of $F, the file utility, described in the Global Utilities Manual. You will find S.SAMPLE on SYSDEV.

Your First Compilation
Run $COBOL, the Global Cobol compiler described in section 4.1, to compile S.SAMPLE. The program does not require a copy library. Do not supply any compiler options. Write the listing to your work volume if your system does not have a printer. By the time the message:

```
$43 END OF FIRST PASS
```

appears, three statements will have been flagged in error. Key <CTRL W>, or <SYSREQ> W if your terminal does not support CTRL, in order to cause a console interrupt. Then key <ESCAPE> in response to the break prompt to obtain a new ready prompt. If the listing file is being written to diskette, aborting the compiler in this way will leave it allocated, and so before continuing it is advisable to delete L.SAMPLE using $F's DEL instruction described in the Global Utilities Manual, thus releasing the diskette space occupied.

Correcting the Errors
Run $EDIT, the text file editor described in Chapter 2, to remove the three erroneous statements and replace them by the three statements:

```
DISPLAY SPACES
DISPLAY "TRAINING EXERCISE COMPLETED BY your name"
DISPLAY SPACES
```

Printing the Source File
Run $PRINT, the file print utility described in the Global Utilities Manual, to produce a print-out of the corrected source file. This should be the same as the example on page A-8 onwards except that your name will appear in place of the name E.A.HART, because of the corrections you have made.

(If you are training on a machine without a printer omit this stage - all your printing will be done later as described in "Printing using Another Machine" at the end of the section.)

Recompiling
Run $COBOL to recompile the corrected source file. Use compiler option ST to produce a symbol table and TC to provide a table of contents. If possible, assign the listing file to a printer. (If you have no printer assign it to the unit containing the work volume).

This time there should be no first pass errors, and you should let the compiler run to completion. The messages:

```
$43 NUMBER OF ERRORS    0
$43 NUMBER OF WARNINGS  0
$43 COMPILATION COMPLETED
```

will appear followed by the ready prompt.

Linkage Editing
Run $LINK, the linkage editor described in Chapter 6, to create an executable program from the module you have successfully compiled.

You should create program SAMPLE, linkage edited at the first location following the debug area, on your work volume. The map listing should be assigned to the printer, if one is available, or to the unit containing the work volume otherwise.

Checking the Program Residence Device Assignment
Before you attempt to run the program you have just created, you may use the $A assignment command to check that unit-id $P - which determines the device from which application programs are loaded - is actually assigned to the unit address your work volume occupies. If this is not the case, you can make the required assignment, then key <ESCAPE> to obtain a ready prompt:

```
GSM READY:$A
.......
....... (list of all unit assignments)
.......
$69 UNIT:$P ADDRESS:209     (if your volume is on 209)
$69 UNIT:<ESCAPE>
GSM READY:
```

A rather simpler way of achieving the same effect is to terminate the program name with <CTRL A> rather <CR> when you key it in response to the ready prompt. This causes $A to delete any current assignment of $P and prompt for it again. For example:

```
GSM READY:SAMPLE<CTRL A>
$03 ASSIGN $P:209
etc. etc.
```

Running the Program
Key the name SAMPLE in response to the ready prompt and the program you have just created will be loaded and executed. The program signs on with the following display:

<PHOTO>

Key the name of the file to be created, e.g. DATA, and then select the create function, C. You will then be prompted to supply the physical unit address corresponding to the logical unit-id DSK used in SAMPLE, and you should reply with the unit address your work volume occupies, e.g. 209. Next you will be prompted to create text records by supplying their text. You reply <CR> when you have finished entering records, when the file will be closed and the display will look like this:

<PHOTO>

Now you can select the D function to display selected records, finally keying <CR> to successive prompts until the ready prompt is once again displayed:

<PHOTO>

Using Debug
Although the program now runs correctly it is instructive to execute it from debug and modify its processing slightly, just as you might when developing a production application.

Run $DEBUG in one partition and the SAMPLE program in another. Then key the N instruction to introduce the symbols of SAMPLE to the symbolic debug table, followed by the T instruction to set a trap on the DO statement that starts paragraph AA010. Key the R instruction to cause the program to be resumed.

The program will be interrupted with the trap program check as soon as it has output the blank line following your "training exercise completed" message. The diagnostic report will appear in the window area of the debugger.

Normally another n blank lines follow before the "specify file identifier" prompt. You can stop those blank lines being produced by using the M instruction to change the value in C-LINE from n to zero. When this is done, use the R instruction to resume the program. The "specify file identifier" prompt will appear on the next line. The last few lines of the dialogue should look like this:

```
Command:M C-LINE: 0
Command:R
```

```
SPECIFY FILE IDENTIFIER:
```

The value of n varies from terminal to terminal depending upon the screen size. For most terminals, it will have a value of 6.

Cross-referencing
For program maintenance purposes it is extremely useful to obtain a cross-reference listing of every program. Run $XREF, the Global Cobol cross-reference utility described in section 4.2, to produce a cross-reference listing of S.SAMPLE. No copy library is needed and the listing can be written to your work volume if your system does not have a printer. If you do not supply any cross-reference options you will obtain the default listing of page A-24, but if you use the SN option the listing of page A-26, with section name prefixes for each reference, will be obtained.

Searching for Strings
$SEARCH, the string search utility described in section 3.2, is another useful program maintenance aid. The listing on page A-28 shows the result of using $SEARCH to find all occurrences of the two strings C-LINE and $$ within S.SAMPLE.

Printing using Another Machine
If you have been training on a system without a printer you will need to move your work volume to a machine with a printer to obtain the source, compilation and map listings of the sample program.

Once this is done run $PRINT and request print-out for:

S.SAMPLE the version of the sample program containing your "correction"

L.SAMPLE the compilation listing

M.SAMPLE the linkage edit map listing

X.SAMPLE the cross-reference listing

D.$SRCH the string search listing

If no printer is available on any machine, you may examine your output list files on your terminal by running $INSPECT.

Source Listing, S.SAMPLE - Page 1
Source Listing, S.SAMPLE - Page 2
Source Listing, S.SAMPLE - Page 3
Source Listing, S.SAMPLE - Page 4
Source Listing, S.SAMPLE - Page 5
Source Listing, S.SAMPLE - Page 6
Compilation Listing, L.SAMPLE - Page 1
Compilation Listing, L.SAMPLE - Page 2
Compilation Listing, L.SAMPLE - Page 3
Compilation Listing, L.SAMPLE - Page 4
Compilation Listing, L.SAMPLE - Page 5
Compilation Listing, L.SAMPLE - Page 6
Compilation Listing, L.SAMPLE - Page 7
Compilation Listing, L.SAMPLE - Page 8
Compilation Listing, L.SAMPLE - Page 9

Link Map Listing, M.SAMPLE
Cross-Reference Listing, X.SAMPLE - Page 1 (Using Default Cross-reference Options)
Cross-Reference Listing, X.SAMPLE - Page 2 (Using Default Cross-reference Options)
Cross-Reference Listing, X.SAMPLE - Page 1 (Using Cross-reference Option SN)
Cross-Reference Listing, X.SAMPLE - Page 2 (Using Cross-reference Option SN)
String Search Listing, D.$SRCH

Appendix B – Linkage Editor Error Messages

The termination errors described in this appendix cause the linkage editor to write a message of the form:

$44 LINK ABORTED - *reason* [FILE *n* MODULE *m*]

to the operator's terminal and the similar message:

*** LINK ABORTED - *reason* [FILE *n* MODULE *m*]

to the link map listing as well, providing a listing has been requested and can be produced. Linkage editing is terminated when the first such message occurs and control is returned to the monitor.

The reason which appears in the message is a short explanatory text. The module number, m, will be 1 if the error occurred in the first module included in the program file, 2 for the second module, and so on. It will be omitted if the error cannot be localised in this way. The file number, n, is the input sequence number of the file in error, counting from one. It is useful in those messages where the module number does not appear.

In the remainder of this section the reasons for error termination are explained in detail, and a recovery strategy suggested.

LISTING FILE IS FULL
This error can only occur if the link map listing is being written to a direct access device rather than a printer. In this case 10K bytes, enough to hold 126 lines of print-out, are allocated to the file. This should be more than adequate for all but the very largest linkage-edits. If the error does occur you must relink, either assigning the listing unit to the printer or, alternatively, suppressing the link map.

NOT OVER 10K FREE FOR PROGRAM FILE
If you assign the link map listing and the program file to the same direct access unit, then the volume must contain at least 10K bytes of contiguous free space. Either relink using different units, or make more free space available by using the file utility's DEL and CON instructions to delete unwanted files and condense the volume.

PROGRAM FILE IS FULL
When a linkage edit begins the program file is allocated the largest amount of contiguous free space available on the volume to which it is to be written. This space will be reduced by 10K bytes if the link map listing is written to the same unit. If this error occurs, either relink using different units or make more free space available by deleting unwanted files and condensing the volume.

LINK LIST BEGINS WITH A LIBRARY
The first file specified in a link list used by this linkage edit is a library. This is clearly an error since $LINK cannot search the library to resolve outstanding global references because no globals are yet defined. Reread section 6.1 thoroughly to make sure you understand the principles involved, then repeat the linkage edit with the correct list.

TOO MANY INPUT FILES
The maximum number of files you can specify in the link list(s) employed by a single linkage edit is 100. If this restriction is met you should use the librarian to place some of your modules in a compilation library, since a library, although it may contain up to 100 members, still counts only as a single file.

TOO MANY INPUT MODULES
A program created by $LINK can be constructed from a maximum of 100 different modules, whether they are included from individual files or libraries. In the unlikely event of this limit being met you must split the program involved into two or more overlays, each of which is then linked as an independent program, as explained in the example of 6.1.11.

TOO MANY GLOBAL SYMBOLS
A program created by $LINK can employ a maximum of 250 global symbols. In the unlikely event of this limit being met you must split it into two or more overlays, each of which is then linked as an independent program, as explained in example 6.1.11.

PROGRAM NAME SAME AS PREVIOUS GLOBAL
The program name of the indicated module is the same as a global already defined in another module. This error usually results if you mistakenly attempt to link the same module twice. Repeat the linkage edit, taking more care when you specify the input files.

MULTIPLE GLOBAL DEFINITIONS FOUND
One or more global symbols are multiply defined in the program under construction. In this case detail messages of the form:

REDEFINITION OF *xxxxxx* FROM *ffff* TO *gggg*

appear on the link map listing immediately following the line naming each module which contains a second or subsequent definition of symbol xxxxxx. The quantities ffff and gggg are, respectively, the hexadecimal location at which the symbol was first defined and the hexadecimal location at which the redefinition would appear were it to be valid.

If the error is not obvious to you, repeat the link without the module (or modules) containing a duplicate definition. You will then be able to find the compilation containing the first definition from the part of the link map listing which reports on the globals that each module contains (this part is suppressed if you have multiple definitions).

Appendix C - Pre-V6.1 Symbolic Debugging Using $DEBUG

C.1 Introduction

This appendix describes the V6.0 debugging system that you will have to use if you are running a V6.0 operating system. The debugging system in the V5.1 and V5.2 operating systems is essentially the same as V6.0. The major difference from the post V6.1 systems is that it runs in the same partition as the program being debugged, overwriting locations 0 - #500. Therefore, if you want to be able to resume your program after a trap you must link it to leave this area free by using the DEBUG link option (see section 6.1.5).

This chapter begins by explaining the concepts underlying the system. A description of the individual instructions used in debugging follows, with related instructions grouped together in the same section so that it is easy to appreciate the scheme on a first reading.

C.1.1 Entering Debug from an Active Program

When the DBUG customisation instruction has been used to indicate that the symbolic debugging system rather than the diagnostics facility is to gain control following an error, the monitor loads debug following any program check, stop code, exit code, or trap, and one of the messages:

```
$91 PGM CHECK number AT location
$91 TERMINATED - STOP code
$91 TERMINATED - EXIT code
$91 TRAP AT location
$91 OVERLAY TRAP ON program-id
```

is displayed, followed by the debug prompt:

```
$50 DEBUG:
```

Debug is also entered when the operator interrupts the program using <CTRL W> and replies Y to the resulting break prompt:

```
<CTRL W>
$91 BREAK?:Y
$50 DEBUG:
```

The debug prompt enables you to key any of the instructions described later in the chapter. In some circumstances you may eventually be able to resume the interrupted program.

If DBUG customisation has not been used, then the $91 DIAGNOSTICS? prompt will be displayed instead of the debug prompt, but you can still enter the symbolic debugging system by replying D to this prompt, for example:

```
$91 DIAGNOSTICS?:D
$50 DEBUG:
```

Once you have entered $DEBUG in this way, if you resume execution of the program any subsequent errors or traps in that program will cause

the symbolic debugging system to be entered directly, without the diagnostics prompt appearing.

C.1.2 Running the $DEBUG Command

You can also enter the debugging system by running the debug command in the normal way from a ready prompt. For example:

```
GSM READY:$DEBUG
$50 DEBUG:
```

In this case instructions are provided to allow you to load the program to be tested, inspect and modify it, and eventually cause it to be executed. You can also set traps on selected GSM Cobol statements so that, when a statement is encountered, the debugging system is re-entered with the message:

$91 TRAP AT *location*

as described in section C.1.1. This enables you to test a program in a highly controlled manner since you can determine exactly those points at which debug is to be re-entered to allow you to inspect or modify storage, or even set further traps. (You can run a program under debug in this way even if the DBUG customisation has not been made).

C.1.3 Debug Instructions

Table C.1.3 summarises the format of the debug instructions described in detail in this chapter. Most of them take one or more operands, though in some cases these may be optional. When present a **single space** must separate the first operand from the instruction letter, and subsequent operands from each other.

Normally the first operand specifies the GSM Cobol location that is the target of the instruction. The following notation is used in the remainder of this section and the table to describe the various formats a location operand can assume:

hhhh hexadecimal format, i.e. a one, two, three or four character hexadecimal value such as E, 1A, 1AC, or 22A1.

name symbolic format, i.e. the short name of a symbol defined in the compilation identified by the previous N instruction. If the actual name is seven characters or more in length only the **first** six characters are significant during debugging.

xxxx indicates an operand which can assume either hexadecimal or symbolic format. If you key an operand which is a valid hexadecimal address and also a symbol in the named compilation (e.g. AA1), then symbolic format is assumed, and the message:

SYMBOL ASSUMED

is output on the same line as the instruction. Subject to the limitation that hexadecimal operands can consist of a maximum of four characters, a 0 prefix can be used

to indicate hexadecimal format (e.g. AA1 may be a symbol, but 0AA1 **must** be hexadecimal).

C.1.4 To Quit

To quit and return control to the monitor, when the problem that caused the error is fully understood, key Q to the debug prompt. Normally <ESCAPE> will return you to the monitor as well, but this will be treated as though you keyed <CR> (resulting in $06 INPUT REQUIRED), if the program under test has disabled ESCAPE key handling by setting system variable $$ESC to 1.

C.1.5 Debugging in Formatted Display Mode

When you are testing a program which is using a formatted display all messages and prompts used by GSM appear on the base line, including those from debug. A comma prompt is output so that you can read consecutive messages. This is so that you do not corrupt the formatted area of the screen and can resume the program should this prove possible.

If you do not mind overwriting the formatted area you can avoid the rather inconvenient base line working by keying the S instruction in response to the debug prompt:

```
$50 DEBUG:S
```

This will cause a SCROLL statement to be executed so that subsequent debug information can make full use of the screen by scrolling upwards and thus eventually pushing the formatted area off the top of the display. If you wish to resume the program under test you must key the S instruction again. This causes debug to execute a CLEAR statement to resume formatted working. The dialogue is thus of the form:

```
$50 DEBUG:S               (restore scrolling)
$50 DEBUG:etc
..........                (debugging using entire screen)
..........
$50 DEBUG:S               (restores formatted working)
$50 DEBUG:R               (resume execution of program)
```

C.1.6 The Debug Area

$DEBUG and the other commands of the debugging system all execute in the first 1280 (#500) bytes of the user area, which is known as the debug area. It is recommended that during testing at any rate you link your programs to avoid this area whenever possible. Indeed, the linker's default, when you do not supply a start address, is to begin each program at location #500, the first byte following the debug area. Even when a failing program occupies the debug area you can still of course use the D instruction and the others which allow you to inspect memory to see what went wrong, although the contents of any variables in the debug area will have been corrupted. There is not much point, of course, in modifying the program or trying to resume it, since it will have been corrupted by $DEBUG.

C.2 Establishing the Symbolic Debug Table for a Compilation

The N instruction described below allows debug to remember the short names and attributes of all symbols defined in a single compilation in

an internal symbolic debug table created temporarily in a free part of the Cobol memory region. You can then use any such symbol (rather than a hexadecimal location and attributes) as the location operand of the instructions which allow you to inspect and modify variables, display memory, clear and set traps, and so on.

The symbols introduced by the N instruction remain available either until another N instruction or the F instruction is used, or until control returns to the monitor, which automatically frees the symbols as part of the end of job processing.

The F instruction, which frees the space occupied by the current symbol table, must be used before running the program under test if it is likely to load an overlay or use the FREE$ system routine to dynamically acquire free storage, as described in the Global Cobol System Subroutines Manual.

Only the first six characters of each symbol are retained in the symbolic debug table so if you are testing a program which has been compiled using the "long names" option and several of its names begin with the six characters you key, this will be interpreted as a reference to the very first such name appearing in the compilation. If you supply a symbol as an operand and it is not present in the debug table the warning message:

```
INVALID - REINPUT
```

is output, followed by the debug prompt.

Global symbols only appear for the compilation in which they are defined, not for every compilation which references them.

C.2.1 N - Name the Compilation for Symbolic Debugging

To name the compilation in which any symbols to be used in subsequent B, C, H, I, M, P, R and T instructions are defined, you must key the instruction:

N *name*

where the name you supply is that used in the compilation's PROGRAM statement.

Following this instruction GSM will prompt you for the program-id of the program which contains the compilation. You may reply <CR> if this was the program last loaded.

You may **omit** the name in the N instruction, in which case debug uses the first six characters of the program-id. (This default option is useful when you wish to work with the main program of a program constructed according to the recommended naming conventions, under which the program-id and main program name are identical.)

C.2.2 N - Examples

The sales invoicing program (program-id SA100) is under test and you wish to examine data within its main program (program name SA100). This was the last program loaded (either by you running it from the

ready prompt or using the L instruction to bring it into memory under the control of debug).

You need only key:

```
$50 DEBUG:N PROGRAM-ID:<CR>
$50 DEBUG:
```

to establish its symbols in the symbolic debug table because in this, the most usual case, all the default conditions prevail.

Next you need to examine data within one of this program's subroutines (program name SASUBR). You key:

```
$50 DEBUG:N SASUBR PROGRAM-ID:<CR>
$50 DEBUG:
```

The symbols of SA100 are replaced in the symbolic debug table by those of SASUBR.

You now wish to continue execution of the program, and expect that it will load an overlay. Accordingly you key the F instruction to release the symbolic debug table space to ensure room to load the overlay. You use debug's E or R instruction (explained later) to continue executing the program and it loads the overlay SALESOVL, which terminates in error with a program check. SA100 is no longer the last program loaded - that is SALESOVL - so if you wish to look at data within SASUBR again you must key:

```
$91 PGM CHECK 11 AT 1E4C
$50 DEBUG:N SASUBR PROGRAM-ID:SA100
$50 DEBUG:
```

The very act of using the N instruction makes the program-id to which it refers count as the last loaded program-id, so if you now wanted to return to your examination of the main program (program name SA100 in program-id SA100) you would simply need to key:

```
$50 DEBUG:N PROGRAM-ID:<CR>
$50 DEBUG:
```

C.2.3 N - Operating Notes

When the N instruction is successful your reply to the PROGRAM-ID: prompt is immediately followed by a debug prompt to allow you to key the next instruction. In some cases, however, the N instruction will fail and one of the following warning messages will precede the debug prompt:

```
NO SYMBOLS
NO ROOM
LOAD ERROR
```

The **first** message means that no symbolic information was present for the compilation in the program specified. Either you have used the N instruction incorrectly, and the wrong program is being searched for the compilation, or no symbols are present in the program file. The latter will be the case if the module was compiled or linked using the NSD option, or if the symbols were eliminated later during library maintenance by using $LIB's NSD instruction.

The **second** message indicates that symbols are available, but there is insufficient free storage within the GSM memory region to hold the necessary symbolic debug table.

The **third** message is output if debug is unable to access the file containing the program for some reason. It will usually be preceded by the retry prompt or the program required prompt.

C.2.4 F - Free the Symbolic Debug Table

To release the symbolic debug table from memory, to enable you to resume or execute a program which uses overlays or dynamically acquires storage, simply key:

```
F
```

in response to the debug prompt:

```
$50 DEBUG:F
$50 DEBUG:
```

C.2.5 F - Operating Notes

If you fail to use the F instruction when it is required the program under test will be terminated with stop code 103 if it attempts to load an overlay. The results are unpredictable if it uses FREE$ to manage memory dynamically.

C.3 Setting the Program Base

When you successfully use the N instruction to name the compilation whose symbolic debug tables are to be loaded, GSM automatically establishes the program base to be the start address of the compilation thus identified. The first effect is to cause location information appearing in the diagnostic report, and the messages:

$91 TRAP AT *location*
$91 PGM CHECK *nn* AT *location*
$91 ADVANCED TO *location*

to be displayed in the form:

address (*base* + *offset*)

For example, if program SAPROG has been linked to start at location #500, and the base has been set to #500, then if an overflow program check takes place at location #0648, the message:

```
$91 PGM CHECK 11 AT 0648 (0500+0148)
```

appears. The importance of the **offset**, 0148 in this example, is that it relates to the location printed on the compilation listing and saves you having to perform hexadecimal subtractions to convert the addresses used at run-time to listing locations.

Whenever a base is established it is automatically added to any hexadecimal operand you may supply to the instructions which allow you to display or print memory, clear and set traps, inspect and modify variables, and so on. This allows you to use listing locations in place of addresses in these instructions once the correct base is

established, avoiding your having to perform hexadecimal additions to convert listing locations to GSM Cobol addresses.

In the instruction descriptions which follow we append the comment:

(relativised)

or:

(not relativised)

to any hexadecimal operands involved to indicate whether or not they are relativised by the addition of the current program base.

As well as the N instruction, the B, P and Z instructions described below can also be used to set the program base. Once established it remains in force until another instruction changes it, or it is reset to zero as part of the processing that precedes the monitor's displaying of the ready prompt or foreground terminated message.

The ? instruction which concludes this section calculates the **offset** relative to the current base of the symbol or hexadecimal value supplied as its operand.

C.3.1 B - Set Base Explicitly

To set the base explicitly you must key the instruction:

B

or:

B *xxxx* (*not relativised*)

in response to the debug prompt. The first form is only used to restore the base when temporarily overridden by a P instruction, as explained in section C.3.5 below.

In the second form xxxx is either a symbol or a hexadecimal value. The address of the symbol or the absolute hexadecimal value supplied will become the new program base. For example, to set the base to location #500, the address of the byte immediately following the debug area, the following dialogue is all that is necessary:

```
$50 DEBUG:B 500
$50 DEBUG:
```

C.3.2 B - Operating Notes

The B instruction is most useful when you are debugging programs where symbolic information is not available, because it allows you to work with the locations printed on the compilation listing rather than absolute addresses, which require hexadecimal conversion. Before you use the other debug instructions you should determine from your linkage edit map listing the starting location of the compilation in which you are interested. This is the value that appears in the LOCN column next to the program name in the PROGRAM column. The value is always hexadecimal 500 for main programs linkage edited to start at the first byte past the debug area.

You can avoid having to use the link map in this way by exploiting the symbolic debugging facility and relying on the N instruction to set the program base.

C.3.3 Z - Set Base to Partition Address

To set the base to a particular partition address you must key the instruction:

```
Z
or:
Z hhhh          (not relativised)
```

in response to the debug prompt. The first form sets the base to partition address 0, and is in fact equivalent to a special case of the second form, namely Z 0. In the second form, hhhh is of course the hexadecimal partition address which is to become the new base.

This instruction is provided for programmers responsible for developing machine code routines to run under GSM, using the technique described in the Global Interface Manual, which defines the term "partition address" in the programming notes to Appendix C.6. These programmers should note that following the Z instruction, in subsequent location information of the form:

    address (base + offset)

the base is the two's complement of the partition address of GSM Cobol location zero and the address is the GSM Cobol location corresponding to the partition address given by offset.

C.3.4 P - Use a Pointer Value as Temporary Base

To use the value of a pointer variable as a temporary base for subsequent instructions you key:

```
P xxxx          (relativised)
```

in response to the debug prompt. The temporary base is then set to the value contained in the two-byte pointer beginning at the hexadecimal location supplied or, if a symbol was keyed, beginning at the location of that symbol.

C.3.5 P - Operating Notes

The P instruction is provided to allow you swift access to data structures which are linked by pointers. It is also useful for handling linkage section or system area data if you are unable to refer to it symbolically. Any number of P instructions can be issued without destroying the permanent base established by the previous B, N or Z instruction. The permanent base is remembered by debug following a P instruction, although it is not used in subsequent relativisation. It can be restored at any time by simply keying B in response to the debug prompt.

C.3.6 P - Examples

The H instruction, explained in detail later, is keyed as:

```
H xxxx          (relativised)
```

and can be used to display an area of memory addressed by its operand as a string of hexadecimal words preceded by location information. We have introduced it here in order to provide a realistic example of the P instruction in use.

Subroutine SASUBR, linkage edited to begin at location 1CE4, is passed a parameter area which contains the field PAPTR. This in turn addresses a 16-byte data area which you wish to examine. The parameter area is, of course, defined in the linkage section.

If the symbols in SASUBR have been introduced by a previous N instruction, proceed as follows:

```
$50 DEBUG:P PAPTR
$50 DEBUG:H 0 :<CR>                 (display 16 bytes)
0648 (0648+0000) 4141 5320 0000 0102 0001 FFFF FFFF 00FF
$50 DEBUG:
```

Alternatively, when no symbolic information is available you will have to consult the compilation listing's symbol table for PAPTR. It appears as:

```
PAPTR 02 P 001A(01E4)
```

This indicates that the pointer to the linkage section group is at location 1E4 relative to the start of SASUBR and PAPTR is offset 1A bytes from the origin of that group. You can now examine the data area as follows:

```
$50 DEBUG:B 1CE4                    (base SASUBR)
$50 DEBUG:P 1E4                     (base group)
$50 DEBUG:P 1A                      (base data area)
$50 DEBUG:H 0 :<CR>                 (display 16 bytes)
0648 (0648+0000) 4141 5320 0000 0102 0001 FFFF FFFF 00FF
$50 DEBUG:B                         (restore permanent base)
$50 DEBUG:
```

These examples show that when you have symbolic information available there is no need to use the P instruction before accessing fields in the linkage section. This is because the debug table contains each linkage section field's pointer and offset, and GSM can therefore calculate its address without requiring you to key these attributes unnecessarily.

You should note that linkage section group pointers are set up at run-time by ENTRY or BASE statements, and therefore a linkage section item cannot be accessed using the P instruction until one of these statements has been executed for the linkage group which is to be examined. Similarly, a based area should not be referenced before its pointer has been initialised.

C.3.7 ? - Display Relative Address

To display the address of a symbol, or relativise a value, you key the instruction:

```
? xxxx                        (not relativised)
```

in response to the debug prompt. The required location is then displayed. For example:

```
$50 DEBUG:N PROGRAM-ID:<CR>
$50 DEBUG:? DIVISO 1248 (0500+D48)
$50 DEBUG:? C3E 0C3E (0500+073E)
```

The instruction provides you with a very easy way of relating symbols and values to listing locations.

C.4 The Diagnostic Report

The D instruction described in this section allows you to display a concise diagnostic report on the console, which is often sufficient to pin-point the cause of the error without further investigation. Alternatively, you can print the diagnostic report, together with selected dumps of main memory. The diagnostics facility, entered following a program check if the DBUG customisation has not been used, also produces a diagnostic report, and this can be followed by a dump of the whole of memory.

Figure C.4 describes the format of the diagnostic report and the photograph shows a report produced following an error in program SAPROG, which was deliberately constructed to fail by dividing by zero.

C.4.1 D - Produce Diagnostic Report and Dumps

To produce a diagnostic report (and possibly follow this by printing memory dumps) you must key the instruction:

```
D
```

or:

```
D unit-id
```

in response to the debug prompt. The first form simply causes the diagnostic report to be displayed, followed by a new debug prompt.

The second form of the instruction causes the diagnostic report to be written as a print file to the unit-id you have specified. For example, to write the report to the standard printer:

```
$50 DEBUG:D $PR
$50 DUMP:
```

The dump prompt that now appears allows you to specify the start address of the memory area you require to output, together with its length:

```
$50 DUMP:xxxx (relativised) LENGTH:length
```

For example:

```
$50 DUMP:SAREC LENGTH:100
$50 DUMP:
```

or:

```
$50 DUMP:EC0 LENGTH:#3C0
$50 DUMP:
```

These examples show how the first input you supply, xxxx, is either a symbol, or a relativised hexadecimal value, and the second input, the length, may be either specified as a decimal number, or a hexadecimal value introduced by a #-character. The printout below shows the dump produced as a result of the second example, when the program base is #500:

Once the information has been written a new dump prompt appears to output another area, should you so wish. This happens over and over again until you reply <CR> to the final dump prompt to close the print file and obtain a new debug prompt. For example:

```
$50 DEBUG:D $PR
$50 DUMP:SAREC LENGTH:100
$50 DUMP:EC0 LENGTH:#3C0
$50 DUMP:CB-DAT LENGTH:#8E
$50 DUMP:<CR>
$50 DEBUG:
```

Figure C.4 - The Diagnostic Report

```
check-type [CODE code] AT location-1
LAST PROGRAM LOADED program
LAST TRANSFER FROM location-2
[LAST FILE ACCESS operation ON file-id]
GSM version CALLED      RETURN ADDRESS
[name]      location-3  location-4
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
......      ..........  ..........
```

One such line for each level of control in the calling sequence at the point of failure. The deeper the level, the later it is listed

```
SYSTEM STACK MODULE TYP USER  ENTRY   SIZE
             module typ user  address size
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
```

One such line for each module in the system stack.

```
nnnn MODULES IN STACK space BYTES SPARE
SYSTEM STACK MODULE TYP USER  ENTRY   SIZE
             module typ user  address size
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
             ...... ... ....  ....... ....
```

One such line for each module in the user stack.

```
nnnn MODULES IN STACK space BYTES SPARE
```

Where:

check-type — is the type of program check that occurred, for example EXIT or OVERFLOW. See UM-C.

code — is the exit or stop code, and is only present following a $91 TERMINATED message. See UM-B or UM-C.

location-1 — is the address of the Cobol instruction at which the error was detected.

program — is the program-id or command name of the last module loaded.

location-2 — is the address of the last transfer of control instruction to have been executed before the error occurred.

operation file-id — indicates the last file processing statement executed (e.g. OPEN OLD, READ NEXT, CLOSE...TRUNCATE). The line is omitted if no such statement has been executed since the previous ready prompt or main menu display.

version — specifies the GSM version and release number.

name — is the first five characters of a CALLed entry name or PERFORMed section name appearing in the control path.

This will be blank if the module containing the entry name or section name was compiled with the NTR option.

location-3 is the address of an EXECed or CALLed entry point, or a PERFORMed section or paragraph.

location-4 is the address to which control would return were an EXIT statement issued at this level. That is it is the address of the statement following the EXEC, CALL or PERFORM that passed control to location-3.

module is the program-id of a module loaded onto a stack, either by the GSM LOAD customisation instruction, or by the LOAD$ system routine described in the Global Cobol Assembler Interface Manual. Modules on the user stack with names $JOBhhhh indicate that the program is running under job management.

typ is P for program modules, or D for data modules.

user is the number of the user who loaded the module onto the stack. Temporary entries on the user stack have the user number suffixed with [T]. If the module was loaded on the system stack as a result of LOAD customisation the word SYSTEM will appear. If the module has been unloaded, but the space cannot be released due to a subsequently loaded module, the word FREE will appear.

address is the address of the entry point of a module on a stack.

size is the size of a module on a stack.

space is the free space in bytes remaining on the user or system stack. For the user stack, this free space represents the currently unused part of the user area.

C.4.2 Operating Notes

Usually the dump is written to the standard printer, but it may be produced on a direct access device. In the latter case it is output to a file named D.$DEBUG.

When you key the D unit-id instruction any existing file named D.$DEBUG is deleted, and a new file with that name created. The file is allocated the maximum amount of contiguous space available on the volume. Any unused space is returned to the system for re-allocation when you reply <CR> to the dump prompt to obtain a new debug prompt.

The prompt:

```
$50 RESET?:
```

appears if an I/O error occurs when debug attempts to open the dump file in response to a D unit-id instruction. If you reply Y this will cause all files currently in use by the program under test to be closed, and the open operation to be retried. This may allow the dump to be produced successfully if the original error arose because the program under test already had the maximum allowable number of files

open for the type of unit in question. Typically, the problem is that you require to write a dump to the printer, which can only handle one file at a time, but it is already being used by the program under test. If you reply Y you may be able to produce the dump, but it will then not be possible to resume the program using debug's R instruction.

There are, of course, other possibilities for error if you attempt to write the dump to printer in a multi-user environment. You are likely to find it is in use by another program. You should normally try to produce the dump on a direct access spool unit or work unit, or your own private work unit.

If you reply N, <CR> (or any single character apart from Y), the dump request will be abandoned and the debug prompt will re-appear.

C.4.3 Notes on the Diagnostic Report

The last file access may be one performed using a system routine, for example CONV$.

The final part of the report, the stack listings, will normally only be of interest if you are loading modules onto one of the stacks, as described in the Global Cobol Assembler Interface Manual. It may also contain modules loaded by GSM. In particular, job management, $REMOTE, $TAPE and the shared memory system all involve the loading of modules onto the stack. If a stack is empty, then the listing is compressed into a single line of the form

SYSTEM STACK EMPTY *nnnnn* BYTES SPARE

as shown in the photograph.

C.5 Inspecting and Modifying Variables

The I and M instructions described in this section allow you to inspect and modify variables using their defined GSM Cobol format rather than a hexadecimal translation of their value. If you are using symbolic debugging you need only refer to a variable by name for debug to display it, and allow you to modify it, according to the picture clause information you supplied when the program containing it was compiled. In the case of groups, subgroups, file and map definitions, which have no picture clause, debug treats them as PIC X items whose length is the sum of their constituent fields. However, it is normally better to use the H instruction, also described in this section, to display the fields of such a group in hexadecimal.

If you attempt to inspect or modify a computational field containing too large a value to be displayed according to its picture clause, then instead of the value the word OVERFLOW is displayed. You will then have to use the H instruction to examine the hexadecimal contents of the field.

Items in the linkage section can be referred to symbolically, but since they are based dynamically it is essential that the base has been set up previously. For example, the operands of the USING clause of an ENTRY statement cannot be inspected until the ENTRY statement has been executed.

If you inspect an entry name, section name or paragraph name, then the first two bytes of the statement at that location are output (in hexadecimal). The first byte will be odd if the trap flag is set on the statement, and even otherwise.

C.5.1 I - Inspect a Variable

To inspect a variable you key the I instruction in response to the debug prompt. If you have established a symbolic debug table by identifying the relevant compilation in a previous N instruction, you need only key the short name of the variable you wish to examine in an instruction of the form:

I *name*

However, if symbolic debug information is not available you must choose one of the four different forms of the I instruction listed below depending on the type of variable involved:

I *hhhh*	pointer
I *hhhh* X*nn*	character or display numeric
I *hhhh* C*pp*	integral computational
I *hhhh* *Cpp,q*	fixed-point computational

In the non-symbolic instruction format:

hhhh specifies the hexadecimal location of the variable to be examined;

nn is a 1 or 2 digit decimal number giving the length in bytes of the character or display numeric variable to be inspected;

pp is a 1 or 2 digit decimal number giving the number of digits before the decimal point of the computational variable to be inspected;

q is a digit giving the number of places following the decimal point of a computational variable.

The symbol table of a Cobol compilation listing contains, in addition to the location, hhhh, of each symbol, a compressed form of picture clause which can be used to deduce nn, pp and q.

C.5.2 Examples

The base address of program SA100, linkage-edited to begin at the location immediately following the debug area, is #500. The compilation listing shows that:

- PNTR, a PIC PTR variable is at location 10;
- STRNG, a PIC X(8) variable is at location 12A;
- TOTAL, a PIC 9(6) COMP variable is at location 156;
- RATE, a PIC 9(4,2) COMP variable is at location 2AE.

If the symbols of SA100 have been established in the debug table by a previous N instruction, these variables can be examined using the following dialogue:

```
$50 DEBUG:I PNTR
132A (0500+0E2A)
$50 DEBUG:I STRNG
ABCDEFGH
$50 DEBUG:I TOTAL
 181204
$50 DEBUG:I RATE
23.09
```

Alternatively, when no symbolic information is available, you have to supply the hexadecimal locations and attributes:

```
$50 DEBUG:B 500
$50 DEBUG:I 10
132A (0500+0E2A)                (pointer)
$50 DEBUG:I 12A X8
ABCDEFGH                        (string)
$50 DEBUG:I 156 C6
181204                          (integer)
$50 DEBUG:I 2AE C4,2
23.09                           (fixed point)
```

Note how, whenever a pointer is inspected, its hexadecimal value is displayed followed by base and offset information in brackets.

C.5.3 M - To Modify a Variable

To modify a variable you key the M instruction in response to the debug prompt. If you have established a symbolic debug table by identifying the relevant compilation in a previous N instruction, you need only key the short name of the variable you wish to modify in an instruction of the form:

M *name*

However, if symbolic debug information is not available you must choose one of the four different forms of the M instruction listed below, depending on the type of variable involved:

M *hhhh*	pointer
M *hhhh* X*nn*	character or display numeric
M *hhhh* C*pp*	integral computational
M *hhhh* C*pp*,*q*	fixed-point computational

The instruction operates exactly as Inspect, described above, except that once the value has been displayed you are prompted with the modify prompt (a single colon):

```
:
```

You must then respond either by supplying a new value for the variable, as described below, or by keying <CR> to indicate that the existing, displayed value is to remain unchanged. Following your reply either the debug prompt is redisplayed, if all is well, or the modify prompt is repeated if the value you supplied is unacceptable.

To modify a **pointer**, simply key one to four hexadecimal digits giving the desired new value. (This facility can be used to set any two adjacent bytes of storage to any specific value).

To modify a **character** or **display numeric variable** simply key the ASCII characters required. If you key less characters than displayed previously, your input will be padded with rightmost blanks.

To modify a **computational variable**, key the new number as you require: It can be signed and have a maximum of 12 digits before the decimal point and up to 6 digits following it.

C.5.4 M - Example

Program SA100 has failed with a numeric conversion program check at location 1C62. It was linkage edited to start at location #500, the first byte following the area used by debug.

Examining the listing you find that a MOVE statement at listing location #1762 has failed. This is because of an erroneous value in the PIC 9(4) display numeric variable ALPHA whose listing location is 8E.

If the symbols in SA100 have been established in the debug table by a previous N instruction (which has the side effect of setting the base at #500) then the following dialogue shows how you might correct ALPHA:

```
$50 DEBUG:M ALPHA
AA**:1111                          (spurious AA** set to 1111)
$50 DEBUG:
```

Alternatively, when no symbolic information is available, you have to supply the hexadecimal locations and attributes of APLHA after having established the base:

```
$50 DEBUG:B 500
$50 DEBUG:M 8E X4
AA**:1111                          (spurious AA** set to 1111)
$50 DEBUG:
```

C.5.5 X - Index the Next I or M Instruction

The X instruction is provided to help you use the I and M instructions to operate on variables within tables or repeating groups established by means of the GSM Cobol OCCURS clause.

Note, however, that you cannot use the X instruction in conjunction with the symbolic form of the I and M instructions to index a variable which is greater than 255 bytes in length, or index a field within a repeating group whose entries are more than 255 bytes long. This limitation is due to symbolic debug table size restrictions. If you mistakenly attempt an X instruction in these circumstances the result will be unpredictable.

To select a particular occurrence of a variable, key the instruction:

```
X index
```

in response to the debug prompt. The index should be a decimal integer between 1 to 9999 inclusive. This will cause the next I or M instruction, which should define the first occurrence of the variable, to actually process the occurrence specified by the index. To select a range of occurrences, key the instruction:

```
X index1 index2
```

in response to the debug prompt. Both the indices specified must be decimal integers between 1 and 9999 inclusive, and index2 must be greater than index1. The next I or M instruction, which should define the next occurrence of the variable, will actually process the occurrences in the range defined by the indices.

In the case of an inspect instruction each entry value will be output on a new line and only when the entire range has been displayed will the next debug prompt appear.

In the case of the modify instruction each entry value will be displayed on a new line followed by a modify prompt. You can then either change the value as described in section C.5.3 or key <CR> to leave the value unaltered and proceed to the next entry. You will obtain the next debug prompt only when the entire range of values has been processed in this way.

The X instruction only affects the very next inspect or modify instruction. A new X instruction must therefore be keyed for every I or M that is to be indexed.

C.5.6 X - Example

A table within program SA100 is defined as follows:

```
01    TABLE OCCURS 20
  02  TANAME      PIC X(8)
  02  TACOUNT     PIC S9(4) COMP
```

The compilation listing shows that the first occurrence of TACOUNT is at location 8E relative to the start of the program. SA100 has been linkage-edited to begin at location #500, the byte immediately following the debug area.

The following examples show you how the index instruction might be used to assist you in modifying the 3rd, 5th and 6th occurrences of TACOUNT.

If the symbols in SA100 have been established in the debug table by the previous N instruction, proceed as follows:

```
$50 DEBUG:X 3 6                      (select items 3 to 6)
$50 DEBUG:M TACOUNT
-1104:108                            (3rd entry changed)
208:<CR>                             (4th entry unchanged)
13:0                                 (5th entry changed)
1287:1                               (6th entry changed)
$50 DEBUG:                           (range processed)
```

C.5.7 Y - Establish Length of Repeating Group

The Y instruction is provided to allow you to inspect variables inside a repeating group when no symbols are available.

To establish the length of a repeating group, key the instruction:

```
Y length
```

in response to the debug prompt. The length should be a decimal integer in the range 1 to 9999. The instruction must be keyed immediately following an X instruction, and immediately before an I or M instruction which specifies a hexadecimal (not symbolic) address. If used under any other circumstances it has no effect.

For example, if you wished to inspect TACOUNT in the example in C.5.6, but symbolic information was not available, you would have to use the following sequence of instructions:

```
$50 DEBUG:X 3 6
$50 DEBUG:Y 20
$50 DEBUG:M 8E C4
etc, etc.
```

Note that if you do not use the Y instruction in the above example, $DEBUG will have no way a telling the length of the group, and you will get erroneous results.

C.5.8 H - Display Memory in Hexadecimal and ASCII

To display an area of memory in hexadecimal, together with an ASCII character interpretation, you key the instruction:

```
H xxxx
```

in response to the debug prompt, to specify the starting location

of the area. You will then be prompted for the number of bytes to be displayed with a single colon:

```
:
```

to which you must reply with a decimal integer or <CR>. If <CR> is keyed 16 bytes will be displayed in hexadecimal, otherwise the number of bytes specified will be rounded up to a multiple of 16. 16 bytes are displayed on each line preceded by the location of the first byte (relativised according to the current base, in the normal way), and followed by the ASCII character equivalents of the bytes, with unprintable characters represented by periods.

C.5.9 H - Example

Program SA100 has suffered a program check whilst processing a record, and you suspect that the record read was not in the correct format and wish to examine the contents of the record area.

The record area is defined by a level 01 data item named SAREC. It is 24 bytes long and starts at location 114 relative to the start of SALES, which has been linkage edited to begin at location #500, the first byte after the debug area.

If the symbols in SA100 have been established in the debug table by a previous N instruction, proceed as follows:

```
$50 DEBUG:H SAREC :24
0614 (0500+0114) 4141 5320 0000 0102 0001 FFFF FFFF 00FF *AAS.............*
0624 (0500+0124) FFFF 3931 3232 FFFF 0000 0000 0000 0000 *..9122..........*
$50 DEBUG:
```

Alternatively, when no symbolic information is available, you have to supply the necessary starting location in hexadecimal, having established the base:

```
$50 DEBUG:B 500
$50 DEBUG:H 114 :24
0614 (0500+0114) 4141 5320 0000 0102 0001 FFFF FFFF 00FF *AAS.............*
0624 (0500+0124) FFFF 3931 3232 FFFF 0000 0000 0000 0000 *..9122..........*
$50 DEBUG:
```

The high values (FFFF) in the output probably indicate data corruption.

C.5.10 G - Get Location of String

To find the location of a particular string in memory, you key the instruction:

```
G hexadecimal-string
```

where the string may be any even number of hexadecimal digits from 2 to 18 inclusive. The whole of the partition, except for the user stack and debug area, is then searched for occurrences of the string, starting at the top of memory. Each time the string is found, its start address is displayed on a new line. For example, to find occurrences of the ASCII character string "123":

```
$50 DEBUG:G 313233
9316 (0500+8E16)
3121 (0500+2C21)
2AB5 (0500+25B5)
$50 DEBUG:
```

The instruction is particularly useful if you do not have an up-to-date listing of the program, or if you want to locate a data item allocated from free memory.

C.6 Executing and Resuming Programs from Debug

The load, execute and resume instructions allow you to use debug to bring a program into memory, modify or inspect it, and then cause it to be executed. If a break, trap, exit or program check occurs debug will be re-entered. You can then examine or alter the program again, and possibly restart it using the resume instruction. A typical sequence might be:

- Run $DEBUG;
- Load subject program using the L instruction;
- Use N to obtain symbolic information;
- Set traps at significant points using T;
- Modify data for testing purposes using M;
- Execute program using E;

(Debug is re-entered when a trapped instruction is encountered)

- Inspect variables using I, H etc.;
- Set additional traps using T;

- Resume subject program using A or R.

The L and E instructions are therefore used to allow you to change the code and data of a program to be tested before it is first executed. A and R are employed to resume the program which was active when debug was entered. You should note, however, that these instructions can only be used successfully if the program involved has been linkage edited to reserve the first 1280 bytes of the user area for debug itself.

C.6.1 L - Load a Program from Debug

To load a program from debug you must key the instruction:

```
L program-id
```

in response to the debug prompt, which will be redisplayed once the program identified by program-id has been satisfactorily loaded. For example:

```
GSM READY:$DEBUG
$50 DEBUG:L SA100
$50 DEBUG:
```

causes the program named SA100 to be loaded as the subject program for debug.

C.6.2 L - Operating Notes

The L instruction should not be issued if a symbolic debug table introduced by the N instruction is present in memory. In this case the table must be freed by the F instruction before L can be used. If you use L wrongly in this way the loader will terminate with stop code 103 and debug will be re-entered:

```
$91 TERMINATED - STOP 103
$50 DEBUG:
```

The L instruction also fails and the warning message:

```
LOAD ERROR
```

is output if there is an irrecoverable I/O error loading the program you have specified; or it cannot be found; or it is too large for the space available. In all but the last case other explanatory prompts (e.g. the retry or program required prompt) will be displayed before the load error warning appears.

C.6.3 E - Execute a Program from Debug

To execute the program which has been previously loaded by the L instruction you must key the instruction:

```
E
```

in response to the debug prompt. Execution of the program will then commence at its entry point. For example, the following dialogue loads SA100 as the subject program, modifies SA100, and then executes it:

```
GSM READY:$DEBUG
$50 DEBUG:L SA100
```

```
$50 DEBUG:etc          (instructions to modify, set traps and so on in SA100)
$50 DEBUG:etc
$50 DEBUG:E
(dialogue from SA100 which now receives control)
```

C.6.4 E - Operating Notes

If you key the E instruction by mistake, when there was no previous L instruction, your request will simply be ignored and the debug prompt will be redisplayed. You must use either the A or R instructions to resume a program following a break, trap or program check.

C.6.5 R - Resume a Program from Debug

To resume a program which has entered debug for any reason you must key the instruction:

```
R
```

or:

```
R xxxx                        (relativised)
```

in response to the debug prompt. The first format, R, is used to continue at the interrupted instruction following a trap or break.

The second format will normally only succeed in resuming the program successfully if a program check occurred because of invalid data and it is possible to correct this using the M (modify) instruction. If this is possible you must resume by specifying xxxx as the location of the failing statement. This location may not correspond exactly to the program check location because a single statement often expands into a number of internal instructions, any of which might fail. You must resume the program on a statement boundary at the same level of control as the failing instruction or unpredictable errors may arise.

C.6.6 R - Operating Notes

If you have removed an application volume to mount SYSRES so that debug itself can be loaded, you must replace your disk before executing the R instruction, or there is a risk that the system volume will be corrupted if the application writes records to the device occupied by the volume in question.

C.6.7 R - Examples

The following dialogue takes place when you set a trap at location 104E of program SA100 in order to inspect certain variables before continuing the program:

```
GSM READY:$DEBUG
$50 DEBUG:L SA100
$50 DEBUG:B 500
$50 DEBUG:T B4E                        (set trap, explained later)
$50 DEBUG:E                            (execute program previously loaded)
.........
......... (dialogue from SA100)
.........
$91 TRAP AT 104E (0500+0B4E)
$50 DEBUG: etc etc                     (instructions to inspect data)
$50 DEBUG:R
.........
......... (more dialogue from SA100)
.........
$91 PGM CHECK 11 AT 218C (0500+1C8C)
$50 DEBUG:
```

At the end of this you have unexpectedly entered debug again with program check 11 (overflow). If you can determine the cause of the problem you may be able to patch data fields and continue testing the program. For example, the following dialogue resumes program execution at location 2184 (listing offset 1C84) which you have determined to be the location of the failing statement, 8 bytes earlier than the instruction which caused the program check:

```
$50 DEBUG:R 1C84
```

Note that if the symbols in SA100 have been established in the debug table by a previous N instruction, then you can use a symbol as the operand of the R instruction. This would allow you to resume at a particular section or paragraph. For example:

```
$50 DEBUG:R BA-OPE
```

resumes the program at the beginning of the section named BA-OPEN-FILES.

C.6.8 A - Advance to a Transfer of Control Instruction

When a program has entered debug for any reason, you may advance its execution up to a subsequent transfer of control instruction by keying the instruction:

```
A
```

in response to the debug prompt. You will then receive a colon prompt, to which you may reply in one of four ways:

A :<CR> means that execution is to proceed to the next transfer of control instruction at the current level of nesting, or at the higher level following execution of an EXIT statement;

A :*nnnn* (1 < nnnn < 9999) means that execution is to proceed to the nnnn'th transfer of control instruction at the current, or a higher, level of nesting;

A :A means that execution is to proceed to the next transfer of control instruction at any level of nesting;

A :A*nnnn* (1 < nnnn < 9999) means that execution is to proceed to the nnnn'th transfer of control instruction at any level of nesting.

When the appropriate transfer of control instruction is reached, this is confirmed by a message of the form:

$91 ADVANCED TO *location*

For example:

```
$50 DEBUG:A :<CR>
$91 ADVANCED TO 0536 (0500 + 0036)
$50 DEBUG:
```

C.6.9 A - Operating Notes

Transfer of control instructions are generated by the following GSM Cobol statements:

AND	CALL	CHAIN	DO
ELSE	ENDDO	EXEC	EXIT
GO TO	IF	ON EXCEPTION	ON OVERFLOW
OR	PERFORM	RUN	STOP RUN

although not always as the first instruction of the statement. The location reported by debug identifies an address within the statement. The CALL, EXEC and PERFORM statements pass control to a lower nesting level, while the EXIT statement passes control to a higher level. The CHAIN and RUN statements pass control to a program executing at the current, or a higher, level of nesting. The STOP RUN statement returns control to the monitor, so if you advance past a STOP RUN statement you will receive a ready prompt. The other statements listed leave the nesting level unchanged.

Note that the conditional statements, AND, DO, IF, ON EXCEPTION, ON OVERFLOW and OR, are always treated as transfers of control even if program flow proceeds to the next sequential statement.

The following statements are equivalent to CALL statements as far as the debug A instruction is concerned:

ACCEPT	CLEAR	CLOSE	DISPLAY
EDIT	LOAD	LOCK	MAPCLEAR
MAPIN	MAPOUT	OPEN	READ
READ NEXT	RELEASE	RETURN	REWRITE
SCAN	SCROLL	SEARCH	SORT
SUSPEND	UNLOCK	WRITE	WRITE NEXT

Note however that some of these statements call monitor facilities and the A instruction ignores all transfer of control instructions within the monitor. Note also that the ACCEPT...NULL variant of the ACCEPT statement generates two transfer of control instructions.

C.6.10 A - Example

Your program contains a DO-loop and you wish to examine the twentieth iteration of the loop in detail. The start of the DO-loop is labelled START, and the loop count is held in COUNT. You proceed as follows:

```
GSM READY:$DEBUG
$50 DEBUG:L PROG                        (load program)
$50 DEBUG:N PROGRAM-ID:<CR>             (introduce symbols)
$50 DEBUG:T START                       (set trap at start of loop)
$50 DEBUG:E
$91 TRAP AT 0536 (0500 + 0036)          (trap at start of loop)
$50 DEBUG:I COUNT
1                                       (check first iteration)
$50 DEBUG:A :<CR>
$91 ADVANCED TO 053A (0500 + 003A)
$50 DEBUG:A :<CR>
$91 ADVANCED TO 0560 (0500 + 0060)
$50 DEBUG:A :<CR>
$91 ADVANCED TO 0536 (0500 + 0036)
$50 DEBUG:I COUNT
2                                       (check second iteration)
(You now know that three A instruction steps are needed
```

```
to advance one iteration of this DO-loop. Therefore a
further 54 steps are needed to advance to the start of
the twentieth iteration)
$50 DEBUG:A :54
$91 ADVANCED TO 0536 (0500 + 0036)
$50 DEBUG:I COUNT
20                                          (check 20th iteration)
$50 DEBUG:
```

Now you are at the start of the twentieth iteration and can start detailed debugging.

C.7 Traps

Debug's T and C instructions, explained below, may be used to set or clear the trap flag associated with any GSM Cobol procedure division statement. The trap flag may be set on any number of statements within the subject program.

The interpreter detects when a program attempts to execute an instruction with the trap flag set, and causes GSM to interrupt it with:

```
$91 TRAP AT location
$50 DEBUG:
```

You can then use whatever debug instructions you require, and normally resume the program at the indicated location, by means of the R instruction. A trap remains in force until either the subject program is reloaded, or it is explicitly cleared using the C instruction.

The O instruction described at the end of this section extends the concept of a trap by allowing you to request that debug is entered whenever a named overlay is loaded.

C.7.1 T - Trap a Statement

To set a trap on a statement you must key the instruction:

```
T xxxx                          (relativised)
```

in response to the debug prompt. The trap flag is then set and the debug prompt redisplayed.

By using the T instruction repeatedly, any number of traps may be set. For example, the dialogue:

```
$50 DEBUG:N PROGRAM-ID:<CR>
$50 DEBUG:T BA-OPE
$50 DEBUG:T 171A
$50 DEBUG:T 1C20
$50 DEBUG:
```

firstly names the currently loaded main program as the compilation whose symbols are to be used, and then sets a trap on BA-OPE, so that the first statement of the section named BA-OPEN-FILES is trapped. Subsequent T instructions are used to trap the statements at listing locations 171A and 1C20.

C.7.2 T - Operating Notes

If you set a trap on an ON OVERFLOW or ON EXCEPTION statement and the exception or overflow condition arises, then the message:

```
$91 PGM CHECK 11 AT location
```

or:

```
$91 TERMINATED - EXIT code
```

corresponding to that condition, rather than the normal $91 TRAP message, will appear immediately before entry to debug.

This indicates that the overflow or exception condition has been cleared which means that if you attempt to resume a program terminated in this way it will execute as though the condition had never occurred. However, if the ON OVERFLOW or ON EXCEPTION statement is met and no overflow or exception condition prevails, then the trap is handled in the normal way.

C.7.3 C - Clear the Trap Flag

To clear a trap on a statement you must key the instruction:

```
C
```

or:

```
C xxxx (relativised)
```

in response to the debug prompt. The trap flag is then cleared and the debug prompt redisplayed.

The first form of the instruction, C, clears the trap flag of the statement which last caused debug to be entered. The second form clears the trap on the statement at the location specified. By using the C instruction repeatedly, any number of traps may be cleared. For example, the following dialogue might take place after the traps introduced in the description of the T instruction in section C.7.1 had been set, and would have the effect of removing all of them:

```
$50 DEBUG:R                    (resume program with traps set)
$91 TRAP AT 1C1A (0500+171A)
$50 DEBUG:C BA-OPE             (clears T BA-OPE)
$50 DEBUG:C                    (clears T 171A)
$50 DEBUG:C 1C20               (clears T 1C20)
```

C.7.4 C - Operating Notes

When you use the R instruction to resume execution of the subject program following a trap, it will not cause the trap to be immediately repeated because the interpreter always ignores the trap flag when it is set on the very first instruction following a resume. This prevents you having to use the C instruction unnecessarily simply to proceed with your program.

C.7.5 O - Trap the Loading of an Overlay

To force a trap whenever a particular overlay is loaded (by CHAIN, EXEC, LOAD, RUN, CALL QLOAD$ or CALL LOAD$ statement) you key the instruction:

```
O program-id
```

in response to the debug prompt. Once the overlay trap has been set the debug prompt is redisplayed.

Only one program at a time can have an overlay trap set for it, and the latest O instruction determines the program affected. You may key the instruction:

```
O
```

with no parameter to clear the overlay trap. The trap is also automatically cleared by the monitor whenever the ready prompt appears.

The trap takes place as soon as the program identified by the O instruction has been loaded but before it is entered. This enables you to set additional traps using the T instruction. When you have finished setting traps, or otherwise modifying the program, you can resume execution by using the R instruction.

C.7.6 O - Example

Program SA100 invokes overlay SA103 by means of an EXEC statement. You wish to set traps in this overlay before it is executed. The following dialogue takes place:

```
GSM READY:$DEBUG
$50 DEBUG:O SA103
$50 DEBUG:L SA100
$50 DEBUG:E
$91 OVERLAY TRAP ON SA103
$50 DEBUG:
```

The L and E instructions are necessary because if you run SA100 normally from the ready prompt, the load trap you established on SA103 will have been cleared by the time the ready prompt appears.