GPL

User's Guide

DPS7000/XTA
NOVASCALE 7000

Languages: General

REFERENCE
47 A2 36UL 03

DPS7000/XTA
NOVASCALE 7000
GPL
User's Guide

Languages: General

July 1990

BULL CEDOC

357 AVENUE PATTON

B.P.20845

49008 ANGERS CEDEX 01

FRANCE

REFERENCE
47 A2 36UL 03

Trademarks and Acknowledgements

Preface

OBJECTIVES

This manual provides the information necessary to run GPL programs on the DPS 7 operating system. The GPL Reference Manual is a companion manual.

INTENDED READERS

This manual is intended for users of GCOS 7 who use GPL.

STRUCTURE

This manual is divided into two parts, Part A entitled "GPL Program Development", and Part B, entitled "Efficient Programming".

Part I

Section 1 is an introduction to the manual and gives an overview of the preparation of a GPL program.

Section 2 describes how the source text of a GPL program is entered and maintained.

Section 3 explains how MACPROC is used to expand the standard GPL system primitives, if they are used in the program.

Section 4 explains how to call the GPL compiler and gives an example of compiler output.

Section 5 explains how the Linker builds an executable load module from compile units.

Section 6 describes how the load module may be executed and debugged.

Part II

Sections 7 to 14 describe the features of GPL, literals and variables, addressability of data, declarations, references, expressions, statements and builtin functions, and how they can aid the user in making more efficient programs.

This manual also has three appendices.

Appendix A lists the limits of the GPL compiler.

Appendix B lists the messages which are displayed in the Job Occurrence Report by the GPL compiler.

Appendix C gives an example of a short GPL program.

ASSOCIATED DOCUMENTS

The following documents can be used in conjunction with this manual:

- For more information about GPL programming

GPL Reference Manual.......................................47 A2 35UL
GPL System Primitives.......................................47 A2 37UL

- For more information about the MACPROC macro

MACPROC Reference Manual.......................................47 A2 70UL
MACPROC User's Guide.......................................47 A2 71UL

- Error Message and Return Codes Message Directory 47 A2 10UJ

JCL Reference Manual.......................................47 A2 11UJ
JCL User Guide.......................................47 A2 12UJ

- For GCOS7 interactive (GCL) functions

IOF Programmer's Manual.......................................47 A2 05UJ
IOF System Administrator's Manual.......................................47 A2 06UJ

IOF Terminal User's Reference Manual (GCOS7-V3):
Part I : Introduction to IOF.......................................47 A2 01UJ
Part II : GCOS Command Language.......................................47 A2 02UJ
Part III : Processor commands.......................................47 A2 03UJ
Part IV : Appendices.......................................47 A2 04UJ

IOF Terminal User's Reference Manual (GCOS7-V5):
Part I : Introduction to IOF.......................................47 A2 21UJ
Part II : GCL Commands (VBO).......................................47 A2 22UJ
Part II : GCL Commands (FBO).......................................47 A2 23UJ
Part III : Directives and General Processor commands.......................................47 A2 24UJ
Part IV : Appendices.......................................47 A2 25UJ

- For manipulations during compilation and linking

Library Maintenance Reference manual.......................................47 A2 01UP
Library Maintenance User's Guide.......................................47 A2 02UP
Linker User's Guide.......................................47 A2 10UP

- For an overview of the system

System Overview.......................................47 A2 04UG
System Administrator's Manual.......................................47 A2 01US

SYNTAX NOTATION

The commands use the following syntax:

ITEM	An item in upper case is a name or keyword and is entered literally as shown. The upper case is merely a convention; in practice you can specify the item in upper or lower case.
item	An item in lower case indicates that a user-supplied value is expected. In most cases it gives the type and maximum length of the value:
char105	a string of up to 105 alphanumeric characters
name31	a name of up to 31 characters
lib78	a library name of up to 78 characters
file78	a file name of up to 78 characters
	In some cases, it gives the format of the value:
	a means a single alphabetic character
	nnn means a 3-digit number
	hh.mm means a time in hours and minutes

In other cases, it is simply descriptive of the value:

device-class
condition
any-characters

$\left\{\begin{array}{l}\text{item}\\\text{item}\\\text{item}\end{array}\right\}$ A list of items enclosed in braces indicates a choice of value. Only one can be selected. Sometimes the list is presented horizontally, with each item separated by a vertical bar, i.e.
{ item | item | item }

[item]	An item enclosed in square brackets is optional.
$\underline{\text{ITEM}}$	An underlined item is a default value. It is the value assumed if none is specified.
<item>	Angle brackets indicate a single key on the micro computer.
= , $ * / \ .	Enter these special non-alphabetic characters as shown.

NOTE: This document uses the following symbols in the left margin:

!	This indicates danger.
-->	This relates to performance.

(1) $$\left[\text{WHEN} \;=\; \left\{\begin{array}{l}\underline{\text{IMMED}}\\ \left[\text{dd.mm.yy.}\right]\ \text{hh.mm}\\ +\ \text{nnnn}\ \left\{\ \text{W}\ \middle|\ \text{D}\ \middle|\ \text{H}\ \middle|\ \text{M}\ \right\}\ \text{item}\end{array}\right\}\right]$$

This means you can specify:

1. Nothing at all, in which case WHEN=IMMED applies.
2. WHEN=IMMED (the same as nothing at all).
3. WHEN=22.30 to specify a time (and today's date).
4. WHEN=10.11.87.22.30 to specify a date and time.
5. WHEN=+0002W to specify 2 weeks from now.
6. WHEN=+0021D to specify 21 days from now.
7. WHEN=+005H to specify 5 hours from now.
8. WHEN=+0123M to specify 123 minutes from now.

(2) PAGES={dec4 **|** (dec4[-dec4][,dec4]...)}

Indicates that PAGES must be specified. Valid entries are a single value or a list of values, enclosed in parentheses. The list can consist of single values seperated by a comma, a range of values separated by a hyphen, or a combination of both. For example:

PAGES=(2,4,10-25,33-36,78,83)

(3) <enter> refers to the return key (the enter key) on the alphanumeric keypad

<transmit> refers to the transmission key on the numeric keypad

Table of Contents

Appendices

Illustrations

1. Introduction

The GCOS Programming Language (GPL) is a PL/1-like language which is suitable for writing system software while also providing the facilities of a high-level language. GPL consists of:

- a subset of PL/1 in accordance with the ANSI 1976 standard.
- certain extensions to the PL/1 subset. These include additional features to aid structured programming and specifically for programming in the DPS 7 GCOS environment.

The GPL programmer accesses the GCOS facilities, task management, file handling, segment management, etc., via system primitives. If these primitives are used in a program, the program must be processed before compilation by the MACPROC utility. MACPROC expands each primitive into source code and inserts the source code into the program. This expansion can then be compiled by the GPL compiler. The steps in a GPL job are:

- MACPROC. Processes primitives, produces an expansion. (If there are no primitives in the program, this step is unnecessary). The use of MACPROC is described fully in Section III of this manual.
- GPL. Compiles the expansion (or the original source program, if MACPROC was not used) and produces a compile unit.
- LINKER. Links the compile unit(s) and produces a load module. See the note below.
- Execution. The load module is loaded and executed.

NOTE: The user can call GPL procedures from within other programs. That is, you can write the main program in, say, COBOL and call procedures written in GPL from within it. The converse is also true: you can call procedures written in other languages from within a GPL program. If this is the case, at linkage time there will be compile units from different compilers to be linked. This process is shown in Figure 1-1.

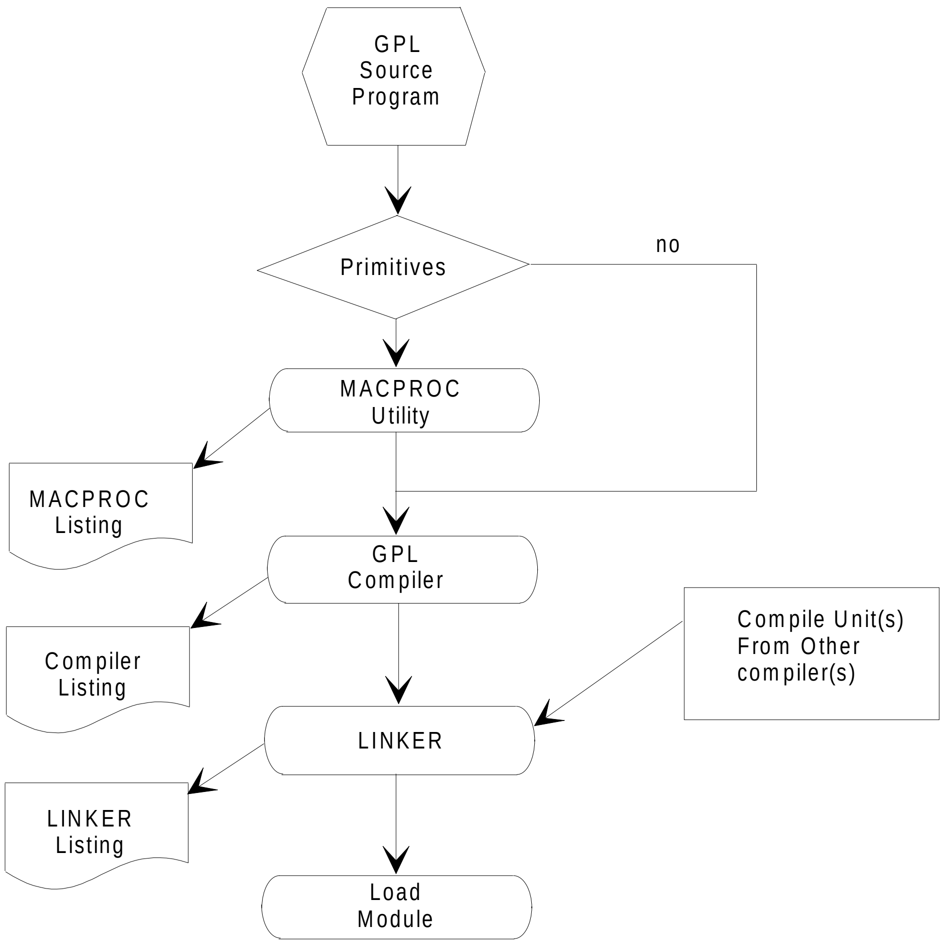


Figure 1-1. Overview of Program Processing

2. Input And Maintenance Of Source Programs

The GPL compiler and the MACPROC utility accept input from an input enclosure or from a library member. An input enclosure must be part of a Batch job; a library member, however, may be used in either Batch mode or interactive mode. Library members are created and updated using the Library Maintenance facilities. The use of input enclosures and libraries for source programs is discussed briefly in the following paragraphs.

2.1 INPUT ENCLOSURES

In the following example an input enclosure called GPROG is used as input to the GPL compiler; the compiler will store its output, the compile unit, in the library RES.CULIB (the library member will be called GPROG).

```
   $JOB...
GPL SOURCE = *GPROG  CULIB = RES.CULIB;   $INPUT GPROG  TYPE = DATASSF;
      GPROG : PROC;      .      .      END;   $ENDINPUT;   $ENDJOB;
```

If the source program contains primitives then the MACPROC utility must be used before compilation; that is, the JCL statement MACPROC is required before the JCL statement GPL. An example is given below.

```
  $JOB...
MACPROC SOURCE = *GPROG  PRTLIB = SL.PRT;
    GPL SOURCE = *GPROG  CULIB = RES.CULIB  PRTLIB = SL.PRT   $INPUT GPROG TYPE = DATASSF;
      GPROG : PROC;
      .
      .
      END;
   $ENDINPUT;
   $ENDJOB;
```

MACPROC reads its input from the input enclosure GPROG, and stores its output, the expansion, in a temporary source library TEMP.SLLIB (the library member will be called GPROG). The listing is stored as a member of the library SL.PRT; the member will be called GPROG_J. The GPL compiler reads its input from the member GPROG in the temporary source library TEMP.SLLIB and stores its output in the library RES.CULIB; the compile unit name will be GPROG. The listing produced by GPL is stored as a member of the library SL.PRT; the member will be called GPROG_L. The JCL statements MACPROC and GPL are described in Sections III and IV of this manual respectively.

2.1.1 Source Libraries

If a program is part of a JCL step, in an input enclosure, it is generally more useful to store the program in a library in order to permit the program to be edited, printed etc..

2.1.2 Creating A Library Member Interactively

To create a library member while working in interactive mode, use the Text Editor (EDIT) or the Full Screen Editor (FSE) under the control of the Interactive Operation Facility (IOF). An example using EDIT at LIBMAINT command level is given and explained below.

```
S: LMN SL LIB = OPER.SLLIB;
>>> 11:25 LMN  50.00 21
C: EDIT;
R: A
I: SFIRST : PROC;
  .
  .
I: /
R: W (GPL) SFIRST
R: /
C: RENUMBER SFIRST;
C: /
<<< 11.28
S:
```

The prompt S: is output by the system; it indicates that the user can enter any JCL statement at job enclosure level. When the Library Maintenance statement is entered, Library Maintenance outputs a heading (>>> etc.) containing the time of day. This is followed by the C: prompt which invites the user to enter a Library Maintenance command. The user then enters EDIT, after which Library Maintenance outputs the R: prompt - this invites the user to enter an EDIT request. The user enters an Append Data request (A), after which Library Maintenance outputs the I: prompt - this invites the user to enter data until the escape sequence / is encountered. The user then enters the source program. When it encounters the escape sequence, Library Maintenance again outputs the R: prompt and waits for a request. The response "W (GPL) SFIRST" requests that the source program just entered be written to a library member named SFIRST. The (GPL) option used in the W request gives the language type value, see "SOURCE PROGRAM FORMAT" below. The line sequence numbers are generated after input by means of the RENUMBER command, as shown in the example. This command generates a sequence number in the SSF header of each record, but does not insert a sequence number into the GPL text. '/' terminates the EDIT session; Library Maintenance outputs the C: prompt and waits for a new command. The RENUMBER command is used to generate line numbers, and then the Library Maintenance session is terminated by '/'. Library Maintenance outputs a termination line (<<< etc); then the S: prompt indicates that the terminal is once more ready to accept GCL statements.

You can also create a GPL program using FSE as follows:

```
S: FSE LIB=OPER.SLLIB;
 >>> 11:15  FSE  20.00
 F: MODIFY NEW=SFIRST LANG=GPL
```

We recommend that, where possible, the same name is used throughout program development for the following:

- input enclosure name
- program name (procedure name of the external procedure)
- compile unit name (taken by the compiler from the program-name)
- load module name (procedure name of the main external procedure)
- expansion name (if MACPROC is used).

This minimizes any confusion that might arise from having different names for the same program at various stages of development.

2.2 UPDATING THE SOURCE MEMBER

A source member can be updated in a Batch job using EDIT. This utility allows the user to insert, replace, or delete specific lines; the lines are identified by their line sequence numbers. EDIT and FSE can be used to modify programs interactively. EDIT is described in the Text Editor User's Guide. FSE is described in the Full Screen Editor User's Guide.

2.3 SOURCE PROGRAM FORMAT

2.3.1 Interactive Line Format

When a library member containing a GPL source program is created interactively, the language type parameter (GPL) should be specified when creating or modifying the source member using the editor (EDIT or FSE). An example of this is given in the paragraph "Creating a Library Member Interactively" above. The specification of the language type parameter ensures the following:

- It causes the language type, GPL, to be written into the SSF control record.
- The member is stored in SSF format (explained below).

Each line is written into the "text" part of an SSF record exactly as it is entered at the terminal. The line number field in the 8-byte SSF header remains blank until sequence numbers are entered by the Library Maintenance command RENUMBER.

2.3.2 System Standard Format

System Standard Format (SSF) is a standard record format used by GCOS. Briefly, storing a GPL source program in SSF format implies two things:

- The first record of each member is a control record, known as a type 101 control record. This control record is created by the system and, as far as MACPROC processing and GPL compilation is concerned, it need not concern the user.
- The user's source code is stored in data records, each of which consists of two parts: an 8-byte record header part and a text part. The record header contains information about the record. This information includes the line number, as explained above.

For a detailed description of the structure of SSF records see the Section entitled "System Standard Format Usage" in the GPL Primitives manual.

2.4 THE IND REQUEST

IND is an FSE request which may be used to make a GPL program more readable. IND rearranges the layout of the program so that, when it is printed, each related sequence of statements (e.g., from a DO or BEGIN to the matching END statement) is equally indented. Comments and the first attributes of DECLARE statements are lined up in a specified column. Indenting a program does not affect its meaning. A full description of the IND request is given in the FSE User's Guide. You may also use the LIBMAINT command INDENT. The same remarks apply.

3. Using Macproc

MACPROC may be called either in batch mode or in interactive mode.

3.1 BATCH MODE

In batch mode, MACPROC is called via the Job Control Language (JCL) statement "MACPROC". JCL is described fully in the "JCL Reference Manual".

3.1.1 The MACPROC JCL Statement

The MACPROC statement is an extended JCL statement. It constitutes a job step in itself and therefore must not appear inside a step enclosure. An example of the syntax for the MACPROC statement is given below. Only the most important and essential parameters are included. The MACPROC statement is described fully in the MACPROC User's Guide.

Note that the underlined parameters are the default values.

MACPROC

$$
\left\{\begin{array}{l}
\text{SOURCE}=*\text{input\_encolure\_name}\\
\text{SOURCE}=\left\{\begin{array}{l}\text{member - name}\\ (\text{member\_name}[\text{member\_name}]\ldots)\\ (\text{star\_name}[\text{star\_name}]\ldots)\end{array}\right\}\\
\qquad\qquad\left[\left\{\begin{array}{r}\left\{\begin{array}{l}\text{INLIB}\\ \\ \text{LIB}\end{array}\right\}=(\text{input\_library})\\ \text{INLIB1}\\ \text{INLIB2}\\ \text{INLIB3}\end{array}\right\}\right]
\end{array}\right\}
$$

$$[\ \text{OUTLIB} = \{\ (\text{output\_library})\ |\ \underline{\text{TEMP}}\ \}\ ]$$

$$[\ \text{PRTLIB} = \{\ (\text{print\_library})\ |\ \underline{\text{TEMP}}\ \}\ ]$$

$$[\ \underline{\text{LIST}}\ |\ \text{NLIST}\ ]$$

$$[\ \underline{\text{XREF}}\ |\ \text{BXREF}\ |\ \text{NXREF}\ ]$$

$$[\ \underline{\text{OBSERV}}\ |\ \text{NOBSERV}\ ]$$

$$[\ \underline{\text{WARN}}\ |\ \text{NWARN}\ ]$$

The following symbolic names refer to standard parameter groups and are described in the JCL Reference Manual: input_library output_library print_library

A detailed description of each parameter may be found in the subsection entitled "PARAMETER DESCRIPTION".

3.2 INTERACTIVE MODE

MACPROC can be executed from an IOF terminal. The JCL statement MACPROC can be used for this purpose when interactive JCL mode is selected. If the interactive mode is GCL, since GPL programs quite often invoke the GPL system primitives, the GCL command "GPL" invokes MACPROC to expand the primitives before calling the GPL Compiler. The "GPL" command is fully described in Section IV of this manual.

3.2.1 Execution Of Macproc

MACPROC executes as in Batch mode with the following exceptions:

- The source text must be held in a library member or in a sequential file. IOF does not use input enclosures.
- If the PRTLIB parameter is not used, the listing is stored in a member of the temporary source library TEMP.SLLIB instead of in the standard SYSOUT file. The listing may be examined using SCANNER or MAINTAIN_LIBRARY.
- If the SILENT parameter is not used, the error diagnostics, with the corresponding source lines, are output on the screen, as well as in the listing.
- All messages sent to the Job Occurrence Report (JOR), including the processing summary, are also sent to the terminal.
- If a break is detected, the current processing (single or serial processing) is terminated immediately.

3.2.2 Interactive Jcl

An example of the interactive use of MACPROC in JCL mode, first without and then with the SILENT parameter, is given in the following lines:

```
S: MACPROC  SOURCE = SHORT_EX  INLIB = MAC.SLLIB;
>>>17:43 MACPROC  50.00
   JUN 06, 1986 17:43:16 X284.2  PROCESSING OF MAC.SLLIB: SHORT_EX
         5  $INDEX ('ABC','A',4);
   *     ERROR 38  SEVERITY 1 DETECTED ON LINE 5
         OBSERV- THE STARTING POINT OF SEARCH IS BEYOND THE
         LIMITS OF THE STRING - THE RETURNED VALUE IS 0.

         $INDEX(ABC,A,4);
   MAC00(50.00) SUMMARY FOR SHORT_EX: *:1 --> OUTPUT PRODUCED  <<<17:43

S: MACPROC  SOURCE = SHORT_EX  INLIB = MAC.SLLIB  SILENT;
   MAC00(50.00) SUMMARY FOR SHORT_EX: *:1 --> OUTPUT PRODUCED
```

3.3 PARAMETER DESCRIPTION

The following paragraphs describe the most important parameters which may be used in the MACPROC JCL statement.

3.3.1 Source, Inlib, Lib And Inlibn

These parameters are used to specify the name and the location of the text or texts to be processed. A series of texts can be processed during a single execution of MACPROC. This is known as serial processing. Note that the source text must always be in SSF format (System Standard Format). When using these parameters, the simplest case is when the source text is held in an "input enclosure" contained in the same job enclosure.

Example:

```
MACPROC SOURCE=*T;
$INPUT T, TYPE=DATASSF;
text
$ENDINPUT;
```

Because the source text must be in SSF format, "TYPE=DATASSF" is mandatory in the JCL statement $INPUT.

- If the source text is held in a library, the name of this library may be specified with the INLIB (alias LIB) parameter, as in the following example:

```
MACPROC  SOURCE = text_mb  LIB = my_sllib;
```

  where "text_mb" is the name of a member in the catalogued library named "my_sllib".

- Up to three libraries may also be specified in separate JCL statements as follows:

```
LIB SL   INLIB1 = input_library_1
       [ INLIB2 = input_library_2
       [ INLIB3 = input_library_3 ]];
MACPROC  SOURCE = text_mb;
```

  The JCL statement LIB defines a "search path" for MACPROC. The source text called "text_mb" is first searched for in input_library_1, specified by INLIB1, then in the library specified by INLIB2, and finally in the library specified by INLIB3, if INLIB2 and INLIB3 are specified. The first member found with the specified name will be processed. Any other member with the same name will be ignored.

- Using the same JCL statement LIB as above, the standard search path may be overridden by the INLIBn parameter, as follows:

```
MACPROC  SOURCE = text_mb  INLIB2;
```

  In this case, the source text is only searched for in the INLIB2 library, and any other member called "text_mb" in the INLIB1 or INLIB3 libraries will be ignored.

- The last three methods of specifying a member name and a library name (cases (b) to (d)), may also be used when a series of source texts is to be processed in a single execution of MACPROC. In this case, the SOURCE parameter must specify a list of member names:

```
MACPROC  SOURCE = (text_mb1, text_mb2, text_mb3 ) ...
```

- As an alternative to specifying a list of member names in the SOURCE parameter, a range of member names can be specified with a "star name". Using the star convention it is possible to select in a specified library, (or in the INLIB1 library when neither the INLIB (alias LIB) nor INLIBn parameter is used), all members whose names have a common characteristic. Conversely, all members whose names have a common characteristic may be excluded from processing:

```
MACPROC  SOURCE = (abc*, bc*z)  INLIB = my_sllib;
MACPROC  SOURCE = ^x*  INLIB2;
MACPROC  SOURCE = ^*z$>bc$<f
```

  The asterisk "*" may match any occurrence of characters, including none, within a name:

  abc* matches all names that begin with "abc"
  bc*z matches all names that begin with "bc" and end with "z"
  The not sign "^" means "all but those ...":

  ^x* matches all names except those that begin with "x"

  The qualifiers "$>" and "$<" restrict the matched names to those that fall alphabetically between, and including, specified values.

  ^*z$>bc$<f matches all those names which do not end with "z, and which begin with any of the letters between "bc" and "f" inclusive.

  Parentheses are not mandatory for one star name, unless the name begins with an asterisk. Otherwise the star name would be interpreted as an input enclosure name.

  Note that the JCL statement "GPL" does not accept the star name convention. (See Section IV for more explanation).

3.3.2 Outlib

The OUTLIB parameter specifies the source library into which the expanded members are to be written. To each source text successfully processed corresponds an expanded member with the same name. As this member replaces any previous member of the same name, the OUTLIB library must be different from the source text library, in order to protect this source text.

The default parameter specification is "OUTLIB = TEMP": the expanded texts are written in members of the temporary source library TEMP.SLLIB.

3.3.3 Prtlib

PRTLIB specifies the source library to which the MACPROC listing is to be written. For each source text processed, successfully or not, there is a listing whose name is the source text name suffixed by "_J". This member replaces any previous member with the same name. The default value depends on the execution mode of MACPROC:

- In Batch mode, MACPROC writes the listings to the SYSOUT file, which will be printed at the end of job execution.
- In interactive mode, the default is "PRTLIB = TEMP": the listings are written to members of the temporary source library TEMP.SLLIB.

3.3.4 List And Nlist

The LIST parameter specifies that a listing of the source text is to be produced. This is the default value. The NLIST parameter specifies that a listing of the source text is not to be produced. However, lines associated with the production of error diagnostics will be produced.

3.3.5 Xref, Bxref And Nxref

The XREF parameter specifies that a cross reference listing of the GPL system primitives encountered during the source text processing is to be produced. This is the default. The primitive names are listed in alphabetical order.

The BXREF parameter specifies that the cross reference listing is to be produced in brief form.

The NXREF parameter specifies that no cross reference listing is to be produced.

3.3.6 Silent And Nsilent

The SILENT and NSILENT parameters are effective only in interactive mode. The SILENT parameter specifies that only the summary is to be printed on the screen. The error diagnostics with the corresponding source lines will only be printed in the listing. The NSILENT parameter specifies that banners, source identification, the summary and the error diagnostics with the corresponding source lines are to be printed on the screen. This is the default.

3.3.7 Observ, Nobserv, Warn And Nwarn

The OBSERV parameter specifies that all observation messages, errors of severity 1, are to be reported in the listing. This is the default. The NOBSERV parameter suppresses the printout of all observation messages. The WARN parameter specifies that all warning messages, errors of severity 2, are to be included in the listing. This is the default. The NWARN parameter suppresses the printout of all warning messages.

3.4 THE MACPROC LISTING

The listing produced by MACPROC consists of four main sections:

- The banner.
- A listing of the source program with error messages; this listing is not produced if NLIST is specified.
- A cross reference listing of the GPL system primitives, in alphabetical order. This listing is not produced if NXREF is specified.
- A summary of errors and a message indicating whether MACPROC processing terminated normally or abnormally; a message to this effect is also printed in the Job Occurrence Report (JOR).
- The listing given here as an example is mainly self-explanatory, but a few points need to be noted here:
- The error messages refer only to errors in the coding of primitives; the rest of the code is checked by the GPL compiler.
- The line numbers were not inserted by the user; they were created by the RENUMBER command when the members were written to the library.
- The last line of the first page of the listing gives the contents of the SSF header record (see subsection 2.3 of this manual). The abbreviations have the following meanings:

CD	Creation Date
CT	Creation Time
MD	Modification Date
MT	Modification Time
SL	Source Language
MN	Modification (Version) Number

The line of information printed at the top of each page in the listing is as follows:

Utility name and version number
Run Occurrence Number (RON)
Job Identifier
User
Project
Account
Time of day and date
Page number

3.5 USE OF THE $ CHARACTER IN SOURCE PROGRAMS

Every primitive within a source program must begin with the character $ (dollar), for example, $H_GET or $H_CHKPT. MACPROC uses the character $ to identify primitives within the source code. In some cases however, a $ character is not intended to identify a primitive. For example:

```
DCL  A  CHAR(3);
     .
     .
A = "A$B";
```

In such a case, the $ character should be immediately followed by a semicolon. MACPROC will translate "A$;B" in the source program into "A$B" in the expanded source program. Note however that if the source to be compiled is passed directly to the GPL Compiler, this pattern must not be used.

```
*********************************************************************************************
*********************************************************************************************
**** GCOS7                                                                               ****
****                                   M A C P R O C                                     ****
****                                                    VERSION:50.00  DATED: JAN 06, 1986 ****
*****ERROR_J************************************************ 40  -1*************************
*********************************************************************************************

ADDITIONAL  INFO:    38



ACTIVE OPTIONS OF MACPROC ARE:
LIST, NEXPLIST, XREF, NCASEQ, NTRACEON, WARN, OBSERV

MAY 28, 1990 10:16:31 X9315.2  PROCESSING OF LSFY.DOC.SLLIB: ERROR
CD=10/30/84 CT=12:49    MD=11/06/84 MT=15:04     SL=GPL MN=24
 1  Error : PROC (Message);
 2  DCL  Message CHAR (*) INPUT;
 3  %REPLACE Buffer_max_length BY 100;
 4  %REPLACE Max_number_of_words BY 200;
 5  %REPLACE Max_word_length BY 20;
 6  DCL 1 Input_interface EXTERNAL STATIC,
 7    2 Buffer_length FIXED BIN (15),
 8    2 Buffer CHAR (Buffer_max_length),
 9    2 End_of_file BIT (1) INIT ("0"b) ;
10  DCL  First_time BIT (1) INIT ("1"b) STATIC;
11  DCL  Line CHAR (80);
12  DCL  No_ref_so_not_alloc AUTO PTR; /* Level1 never ref'd so not alloc */
13  DCL Number_of_words FIXED BIN (15) EXTERNAL STATIC;
14  DCL 1 Word (Max_number_of_words) EXTERNAL STATIC,
15      2 Name CHAR (Max_word_length),
16      2 Counter FIXED BIN (15);
17            $H_FD INPUT ACTUAL;
18            $H_FD OUTPUT ACTUAL SYSOUT;
19            $H_OPEN OUTPUT PMD = OU;
20            IF $H_TESTRC ^DONE;
21            THEN $H_ABTSK;
22            Line = Message;
23            $H_PUT OUTPUT WA = Line ALN = 'MEASURE (Line)';
24            $H_ABTSK;
25  Read_in_buffer : ENTRY;
26            IF First_time
27            THEN DO;
28                 $H_OPEN INPUT PMD = IN;
29                 IF $H_TESTRC ^DONE;
30                 THEN CALL Error ("Cannot open INPUT");
31                 First_time = "0"b;
32            END;
33            $H_GET INPUT WA = Buffer ALN = 'MEASURE (Buffer)' OUTLEN = Buffer_length;
34            SELECT;
35                 WHEN ($H_TESTRC DATALIM;) End_of_file = "1"b;
36                 WHEN ($H_TESTRC ^DONE;) CALL Error ("Cannot get INPUT");
37                 OTHER;
38            END;
39            RETURN;
40  Write_word_array : ENTRY;
41  DCL i FIXED BIN (15);
42            $H_CLOSE INPUT;
43            $H_OPEN OUTPUT PMD = OU;
44            IF $H_TESTRC ^DONE;
45            THEN $H_ABTSK;
46                 Line = "  Statistics on the text :";
47                 $H_PUT OUTPUT WA = Line ALN = 'MEASURE (Line)';
48                 IF $H_TESTRC ^DONE;
49                 THEN $H_ABTSK;
50            DO i = 1 TO Number_of_words;
51                 Line = "  " !! Word (i).Name !! CHAR (Word (i).Counter);
52                 $H_PUT OUTPUT WA = Line ALN = 'MEASURE (Line)';
53                 IF $H_TESTRC ^DONE;
54                 THEN $H_ABTSK;
55            END;
56            $H_CLOSE OUTPUT;
57       END Error;

            ( * INDIRECT REF, + MULTIPLE REFS)                                CROSS_REF_MAP
NAME                                    TYPE           DATE_TIME         ORIGIN
```

```
             REFERENCES
H_ABTSK                                 EXTERNAL MACRO   07/29/86 16:22 FROM SYS.GPL.MACLIB
             21  24  45  49  54
H_CLOSE                                 EXTERNAL MACRO   07/29/86 16:25 FROM SYS.GPL.MACLIB
             42  56
H_FD                                    EXTERNAL MACRO   07/29/86 16:36 FROM SYS.GPL.MACLIB
             17  18
H_GET                                   EXTERNAL MACRO   07/29/86 16:39 FROM SYS.GPL.MACLIB
             33
H_OPEN                                  EXTERNAL MACRO   07/29/86 16:41 FROM SYS.GPL.MACLIB
             19  28  43
H_PUT                                   EXTERNAL MACRO   07/29/86 16:42 FROM SYS.GPL.MACLIB
             23  47  52
H_TESTRC                                EXTERNAL MACRO   07/29/86 16:48 FROM SYS.GPL.MACLIB
             20  29  35  36  44  48  53


          MACPROC PROCESSING : NO ERROR DETECTED
          OUTPUT PRODUCED

*****ERROR_J***************************M*A*C*P*R*O*C****************************************
```

Figure 3-1. MACPROC Listing

4. Using The GPL Compiler

This section explains how to use the GPL Compiler. The JCL statement "GPL", the GCL command "GPL", and the printed output produced by the Compiler are described in detail.

4.1 THE JCL STATEMENT GPL

The JCL statement GPL, which is used to call the compiler, is an extended statement. It constitutes a job step in itself and so must not appear inside a step enclosure. GPL compiles the user's source code and produces a compile unit (object code) which can then be input to the LINKER utility. The format of the GPL statement is shown below.

```
GPL
 ⎧ SOURCE = * input_enclosure_name                                        ⎫
 ⎨          ⎧nb_name                      ⎫                                ⎬
 ⎩ SOURCE = ⎨ALL                          ⎬ INLIB = { (input_library) }    ⎭
            ⎩(mb_name [ ,mb_name ]...)    ⎭         { TEMP          }

 [ CULIB = { (output_library) } ]
           { TEMP             }

 [ OBJ | NOBJ ]  [ WARN | NWARN ]

 [ OBSERV | NOBSERV ]  [ MAP | NMAP ]

 [ OPTIMIZE = { 0 | 1 | 2 | 3 | 4 } ]

 [ DEBUG | NDEBUG ]  [ XREF | NXREF ]

 [ DEBUGMD | NDEBUGMD ]  [ DCLXREF | NDCLXREF ]

 [ LIST | NLIST ]  [ LEVEL = {GPL | PL1} ]  [ CASEQ | NCASEQ ]

 [ BRIEF ]  [ ILN | XLN ]

 [ SILENT ]  [ CODE = { OBJA | OBJCD } ]

 [ DUMP = { NO | DATA } ]

 [ STEPOPT = ( step_parameters ) ]

 [ ⎧PRTFILE = ( print_file )          ⎫ ]
 [ ⎨PRTLIB  = ⎧( print_library )⎫     ⎬ ]
 [ ⎩          ⎩TEMP             ⎭     ⎭ ]
```

Note that TEMP may be specified for PRTLIB only under IOF, in which case it is the default.

4.2 DESCRIPTION OF PARAMETERS

The table below describes the parameters which may be used in the GPL statement. Note that the following symbolic names used in the statement description above refer to standard parameter groups which are described in the JCL Reference Manual.

input_library	= input-library-description
output_library	= output-library-description
print_file	= print-file-description
print_library	= print-library-description

Table 4-1. GPL Statement Parameters (1/2)

Parameter Value	Notes
mb_name	Indicates the name of the member in the library defined by the INLIB parameter. The number of mb_names given in a list is limited to 90.
ALL	All the members of the library defined by the INLIB parameter will be compiled.
INLIB = TEMP	Specifies a temporary library created by a preceding step.
CULIB	Specifies the library into which CUs are to be placed (if OBJ option is ON).
PRTLIB print_library	Specifies the library to which the output listing will be written. The member name will be the main procedure name suffixed by "_L".
PRIFILE print_file	The listing will be written to a sequential file.
	Note that if neither PRTLIB nor PRIFILE is present, the listing will be written to the standard SYSOUT for printing at job termination.
OBJ	Causes the compiler to generate a compile unit in the library specified by the CULIB parameter.
NOBJ	No compile unit is generated
WARN	Diagnostics defined as WARNING messages are output.
NWARN	Diagnostics defined as WARNING messages are not output. This implies NOBSERV.
OBSERV	Diagnostics defined as observation messages are output.
NOBSERV	Diagnostics defined as observation messages are not output.
MAP	A data_map listing is issued.
NMAP	No data_map listing is issued.
DCLXREF	A cross reference listing sorted by line of declaration is issued.
NDCLXREF	Opposite of DCLXREF parameter.
XREF	A cross reference listing sorted by variable name in alphabetical order is issued.
NXREF	Opposite of WREF parameter.
LIST	A listing of the original source program is issued. Diagnostic messages are written in the margins of the lines which contain errors.
NLIST	Only the lines with errors are listed.
DEBUG	Causes the data base for PCF to be produced.
NDEBUG	The data base for PCF is not produced.

Table 4-1. GPL Statement Parameters (2/2)

Parameter Value	Notes
DEBUGMD	The text enclosed between % DEBUG and % END_DEBUG will be compiled.
LEVEL	Checks the language level.
BRIEF	Non-level1 names which are not referenced will not be cross referenced.
XLN	The line number given in the cross reference and in the line location map will be the external line number.
ILN	The line number given in the cross reference and in the line location map will be the internal line number given by the compiler.
SILENT	Only the summary line and the system errors, if any, will be reported to the console during an interactive compilation. Ineffective in Batch mode.
CASEQ	Lower case letters are converted to upper case letters everywhere except within character string literals.
NCASEQ	Lower case letters are not converted. The words belonging to the language (even if not reserved) must be in upper case.
OPTIMIZE 0 1 2 3 4	No optimization Statement optimization Local optimization Global optimization Enhanced global optimization, with automatic in-line procedure insertion and loop unrolling.

4.2.1 The Code Parameter

The CODE parameter specifies the class of the target computer for which code will be generated. The different classes are:

• Class A	: DPS 7/X5, X07 and 64/DPS
• Class C	: DPS 7/X0, X17, X27 and DPS 7000
• Class D	: DPS 7/1017, 1027

If CODE=OBJA is used, the program can be run on a class A or C computer.

If CODE=OBJCD is used, the program can be run on a class C or D computer. When the GPL compiler runs under GCOS 7-V3A, the default value is OBJA. When the GPL compiler runs under GCOS 7-V3B or V5, the default value is OBJCD.

If no value is given, the default value OBJA is chosen when compiling under GCOS7-V3A system version.

If a program is compiled with CODE=OBJA and executed on a class D computer, there is a loss of precision on floating point results, and the program is rejected by the system.

A program compiled with CODE=OBJCD and executed on a class A computer may stop with the error message: "ILLEGAL FIELD IN INSTRUCTION".

The LIST command of the Library Maintenance processor may be used to get information on the compatibility class of a compiled unit. It should be interpreted as follows:

CU CLASS	A-C COMPATIBLE	C-C COMPATIBLE
0 or none	yes	yes
1	yes	no
2	no	yes
3	no	no
4	unknown	unknown

4.3 GCL MODE

When in GCL mode in the IOF domain, use the GPL command to call the GPL compiler.

The parameters are similar to those explained for JCL mode, but with the usual GCL conventions, for example, namely boolean values, and file literals.

Menus and helps can be requested if guidance is needed.

Full details about the GCL command GPL can be found in Part II of the IOF Terminal User's Reference Manual.

NOTE: The GPL procedure includes a call to MACPROC prior to compilation.

The following Table contains a transcript of a call to this procedure through a menu. GPL

```
Compile GPL program(s)
SOURCE                + source program names
my_prog
INLIB                 input library (def. is #SLIB-<>#HSINLIB)
my_lib
CULIB                 output library (default is #CLIB)
MLIST                 list source text?                          1
LIST                  list expanded text?                        0
MAP                   produce a data map?                        0
XREF                  produce a cross reference table?           0
DCLXREF               cross references in DCL order?             0
BRIEF                 brief cross references?                    0
DEBUG                 produce PCF information?                   0
DEBUGMD               Debug mode?                                0
WARN                  report warnings                            1
OBSERV                report observations?                       1
LEVEL                 GPL, PL1                                   GPL

PRTLIB                listing library (default is #PRTLIB)
RETRIEVE              retrieve libs. (default #BLIB -<>#BINLIB)
OPTIMIZE              level from 0 to 4
```

Note that all these parameters are described in detail in the entry for the GPL command in Volume II of the IOF Terminal User's Reference Manual.

4.4 COMPILER OUTPUT

The following paragraphs give a brief description of the output produced by the GPL compiler. The output is described in the order in which it is printed, under the following headings:

- Banner Page
- Source Program Listing
- Data Maps
- Cross-Reference Listing
- Summary Page

4.4.1 Banner Page

A sample banner page is shown in Appendix C. The name of the listing file is formed by adding the suffix "_L" to the name of the main procedure. The compiler always produces a banner page.

4.4.2 Source Program Listing

A listing of the user's source program is produced, unless the parameter NLIST is specified. See Appendix C for examples of source program listings produced by the compiler. The source listing is mainly self-explanatory, but some explanation is necessary for line numbers, primitives and diagnostic messages.

4.4.2.1 Line Numbers

The line numbers printed on the left of the source lines are respectively the external line numbers (XLN) and the internal line numbers (ILN). The XLN is taken from the source input file; if no numbers were generated in the SSF header, the XLN for each line is zero (but see 'PRIMITIVES' below). The ILN is generated by the compiler, which uses it to identify the line. The XLN and ILN are completely independent of each other.

4.4.2.2 Primitives

The source listing below was created when GPL compiled the expansion produced by MACPROC from a procedure containing one primitive.

```
FIRST_PROC
           SOURCE
      0       1 FIRST_PROC:PROC;
      0       2    DCL AA  CHAR(250);
      0       3
  .0002       4
  .0003       5 /* $H_PUTACT  AA,LENGTH=30;  *V3*/
  .0004       6
      0      16 RETURN;
      0      17 END;
     XLN     ILN
```

The expansion code is not printed; its existence is, however, indicated in the listing. the example above, the ILN jumps from 6 to 16: the omitted lines contain the expanded code. Blank lines are inserted on either side of the primitive statement, and the statement itself appears inside comment symbols (i.e. /* and */). The level of the GCOS operating system is also indicated inside the comment symbols. The XLN for the comment line and the blank lines above and below it is prefixed with a dot. This indicates that the line was inserted by the text editor or generated by MACPROC.

4.4.2.3 Diagnostic Messages In The Source Listing

When incorrect or inconsistent code is detected during compilation, a diagnostic message is printed after the offending line. The message is of the form:

```
aaaa   order-no  code  message-text
```

Example:

```
 ***   1  D3  THIS VARIABLE IS NOT EXPLICITLY DECLARED IN
       THE PROGRAM
```

where:

`aaaa` is one, two, three or four asterisks, indicating the severity of the message as follows:

```
   *  observation
  **  warning
 ***  serious error
****  fatal error
```

An observation message indicates the action taken by the compiler when this may not be clear from the source code. Observation messages may be suppressed by specifying the NOBSERV or NWARN parameter in the JCL statement GPL.

A warning message indicates a possible error. The statement is compiled, but the results may be unexpected. Warning messages may be suppressed by specifying the NWARN parameter in the JCL statement GPL. A serious error message indicates a major error in the program. The compiler continues to check the source code but does not produce a compile unit. The message "NO CU PRODUCED" is printed in the summary page and in the JOR. A fatal error message indicates that an error has occurred which prevents the compiler from continuing its analysis or from generating object code. This could be a system error, a compiler error, compiler limit exceeded, user error, use of a feature not included in the level of compilation being used, etc.. No compile unit is produced and a message to this effect is printed in the summary page and in the JOR.

order-no	If there is more than one error in a line, this number indicates the order in which the errors occurred.
code	is the code number of the error which has occurred.
message-text	is a short explanation of the error.

4.4.3 Data Maps

Data maps are only produced if the MAP parameter is specified; NMAP is the default. The MAP parameter produces the following three maps:

- SYMDEF map
- SYMREF map
- line location map

A SYMDEF (symbolic definition) is an entry point or data entity within the compile unit, which may be referred to from another compile unit. A SYMREF (symbolic reference) is a reference to another compile unit. SYMDEFs and SYMREFs are generated at compilation time and matched at linkage time. See the LINKER User's Guide for more information.

4.4.3.1 Symdef Map

The SYMDEF map specifies how many SYMDEFs are generated, their names, and whether they are procedure or data SYMREFs. The address specified gives the internal segment number (ISN) and the displacement within the segment. The address is given in the form:

```
/internal-segment-number              displacement/
```

In Figure 4-1, the address /0 08/ indicates that the procedure descriptor for ERROR is in segment 0 and starts at byte 8. This information is useful if you wish to locate the procedure in a memory dump. To find the start of the procedure in the dump, use the displacement in conjunction with the segment number obtained from the linker listing. It is however preferable to use the Program Checkout Facility for this purpose. PCF is described briefly in subsection 6.3 of this manual, and in the Program Checkout Facility User's Guide.

4.4.3.2 Symref Map

The SYMREF map specifies how many SYMREFs were generated, their names and whether they are procedure or data SYMREFs. The addresses specified are for use by the Service Center. They are of no direct interest to the user.

4.4.3.3 Segment Map

The segment map specifies how many segments have been generated. The information is given in the form:

```
          usage     /isn   displacement/     size    name
```

where:

usage	can be LINKAGE_SECTION, CODE SEGMENT or DATA SEGMENT.
isn	is the internal segment number given by the GPL compiler.
displacement	is the offset from the beginning of the segment.
size	is the size of the segment in bytes. It is specified in decimal and hexadecimal in parentheses.
name	is the name of the segment, if it has one.

The first value for isn 0 is the value of register BR7 at the beginning of the procedure, which is the offset of the linkage section.

4.4.3.4 Line Location Map

The line location map specifies the segment and location of the instructions generated from each line of source code. This information is useful when an exception message which contains an address has been output. You can trace on the line location map the line of source code where the exception occurred. Note that the address given is the byte at which the instructions begin. In the example listing, Figure 4-4, if the exception message specified location 36A, this would indicate line 43 of the source code. The line number specified is the ILN. Note that the instructions generated from one line of source code may occupy a large number of bytes. Also, one line of source code may have more than one entry in the line location map. The data maps in Figures 4-1 to 4-4 were produced by the example program given in Section V.

4.4.4 Cross Reference Listing

A cross-reference listing is produced if either the XREF or DCLXREF parameter is specified. The only difference between the two parameters is that with XREF the variable names are given in alphabetical order, and with DCLXREF, they are given in order of declaration. The example cross-reference map shown in Figure 4-5 is produced by XREF. It gives the following information:

- Level number (if applicable)
- Name of variable
- Address (base register and offset)
- Data type
- Storage class
- Number of the line on which the variable was declared
- Number of each line on which the variable is referenced.

The address may be used for debugging purposes, in the DUMP command of the Program Checkout Facility, however it is preferable to use the symbolic addressing facilities of PCF. See the PCF User's Guide for details. Note that for internal procedures, the address given is that which contains the return address. A separate cross-reference map is produced if the compile-time statement %REPLACE is used in the program (and XREF or DCLXREF is specified). An example is given in Appendix C.

4.4.5 Summary Page

The final page of the compilation listing is the summary page. This states whether the compilation produced object code and if any error messages were generated, gives a summary of the messages and their line numbers. A successful compilation which generates object code and caused no error messages to be produced will have a summary page which simply says:

```
                         NO ERROR MESSAGES
                        OBJECT CODE PRODUCED
```

In the example below, one serious error was detected at line 29 and no object code was produced.

```
                    NUMBER OF ERROR MESSAGES

                         *          0

                       * *          0

                     * * *          1

                   * * * *          0


          ERRONEOUS LINES
    29 ***
          NO OBJECT PRODUCED
```

4.5 NAMING CONVENTIONS

Let S be the source member name. Let P be the program name. Then:

- S_J contains the report listing of MACPROC in the PRTLIB defined for MACPROC.
- S contains the expansion created by MACPROC in OUTLIB.
- P_L contains the report listing of GPL in the PRTLIB defined for GPL.
- P contains the compile unit created by GPL in the CULIB.

4.6 COMPILER MESSAGES IN THE JOR

For every program input to the compiler, the following message is printed in the JOR:

```
    GPL00    (vv.nn)     SUMMARY FOR   program-name
                       error-summary     [NO] CU PRODUCED
```

In the example given below, the program NICE_ONE compiled successfully with no error messages:

```
   GPL00     (72.00)     SUMMARY FOR  NICE_ONE
                           NO ERRORS          CU PRODUCED
```

The message below was produced by the program TWO_BAD, whose compilation terminated abnormally with one serious error. No compile unit was produced:

```
   GPL00     (72.00)     SUMMARY FOR  TWO_BAD
                           ***1           NO CU PRODUCED
```

The same information is given for each program in the summary page of the compilation listing. (The terms "compile unit" and "object code" are synonymous). When a compilation terminates abnormally, there will usually be explanatory messages in the source listing and/or further messages in the JOR. Appendix B contains an annotated list of all the JOR messages produced by the GPL compiler.

```
ERROR

         SOURCE
    SYMDEF                         ADDRESS

ERROR                            /  0     08/                  PROCEDURE
READ_IN_BUFFER                   /  0     10/                  PROCEDURE
WRITE_WORD_ARRAY                 /  0     18/                  PROCEDURE
INPUT_INTERFACE                  /  1     00/                  DATA
INPUT                            /  2     00/                  DATA
H_S_INPUT                        /  3     00/                  DATA
OUTPUT                           /  4     00/                  DATA
H_S_OUTPUT                       /  5     00/                  DATA
   8 SYMDEFS GENERATED:    5 REFERENCE DATA.       3 REFERENCE PROCEDURES.
THE ADDRESSES ABOVE REFER TO INTERNAL SEGMENT NUMBERS (ISN'S) WHICH ARE MAPPED
INTO
SEGMENT TABLE NUMBERS (STN'S) AND SEGMENT TABLE ENTRIES (STE'S) BY THE STATIC
LINKER.
```

Figure 4-1. SYMDEF Data Map

```
ERROR

         SOURCE
    SYMDEF                  ADDRESS

                          /  0     0C/=  /  0     20/   SEGMENT NUMBER
ERROR                     /  0     20/                  PROCEDURE
                          /  0     14/=  /  0     20/   SEGMENT NUMBER
READ_IN_BUFFER            /  0     24/                  PROCEDURE
                          /  0     1C/=  /  0     20/   SEGMENT NUMBER
WRITE_WORD_ARRAY          /  0     28/                  PROCEDURE
INPUT_INTERFACE           /  0     2C/                  DATA
NUMBER_OF_WORDS           /  0     30/                  DATA
WORD                      /  0     34/                  DATA
H_STND2_TDF1              /  0     38/                  DATA
INPUT                     /  0     3C/                  DATA
H_S_INPUT                 /  0     40/                  DATA
OUTPUT                    /  0     44/                  DATA
H_S_OUTPUT                /  0     48/                  DATA
H_DFPRE_UOPF              /  0     4C/                  PROCEDURE
H_TASKM_UABT              /  0     50/                  PROCEDURE
H_DFPRE_UCFM              /  0     54/                  PROCEDURE
                          /  0     58/=  /  6     00/   SEGMENT NUMBER
                          /  0     08/= E/  0     BC/   SEGMENT NUMBER
                          /  0     10/= E/  0    1FC/   SEGMENT NUMBER
                          /  0     18/= E/  0    336/   SEGMENT NUMBER
H_S_INPUT                 /  2     00/                  DATA
H_STND2_TDF1              /  3     01/                  DATA
H_S_OUTPUT                /  4     00/                  DATA
H_STND2_TDF1              /  5     01/                  DATA

25 SYMREFS GENERATED:
12 REFERENCE DATA. 6 REFERENCE PROCEDURES. 7 SEGMENT NUMBERS.

THE ADDRESSES ABOVE REFER TO INTERNAL SEGMENT NUMBERS (ISN'S) WHICH ARE MAPPED
INTO SEGMENT TABLE NUMBERS (STN'S) AND SEGMENT TABLE ENTRIES (STE'S) BY THE
STATIC LINKER.
```

Figure 4-2. SYMREF Data Map

```
ERROR
          SEGMENT MAP
     USAGE                /BEGINNING /          SIZE
NAME

LINKAGE SECTION           /  0     20/      9C     (     156)
CODE SEGMENT              /  0     BC/     51E     (    1310)
DATA SEGMENT              /  1     00/      67     (     103)
DATA SEGMENT              /  2     00/      08     (       8)
DATA SEGMENT              /  3     00/      4D     (      77)
DATA SEGMENT              /  4     00/      08     (       8)
DATA SEGMENT              /  5     00/      4D     (      77)
DATA SEGMENT              /  6     00/      01     (       1)
```

Figure 4-3. Segment Map

```
LINE:LOC LINE:LOC LINE:LOC LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC
LINE:LOC

ISN:   0
  1:BC   141:CC   141:D4   142:DC    143:E4    144:EC    145:F4    146:108   153:110
172:118
172:124  172:12C  176:15C  235:16E   236:176   237:17E   238:186   239:18A   240:190
241:198
258:1B4  258:1C0  258:1C8  263:204   302:20E   302:216   303:21E   304:226   305:22E
306:236
307:24A  314:252  323:25A  324:286   385:28C   386:294   387:29C   388:2A8   389:2B0
390:2B6
391:2BE  392:2C6  399:2E2  409:2F6   419:304   422:334   452:33E   453:346   454:34E
455:356
456:36A  500:372  500:37A  501:382   502:38A   503:392   504:39A   505:3AE   512:3B6
531:3BE
531:3CA  531:3D2  535:402  594:412   595:41A   596:422   597:42A   598:42E   599:434
600:43C
607:458  626:460  626:46C  626:474   630:4A4   631:4BE   690:504   691:50C   692:514
693:51C
694:524  695:52A  696:532  703:54E   722:556   722:562   722:56A   726:59A   754:5A4
755:5AC
756:5B4  757:5BC  758:5D0  765:5D8


LOC:LINE LOC:LINE LOC:LINE LOC:LINE LOC:LINE  LOC:LINE  LOC:LINE  LOC:LINE  LOC:LINE
LOC:LINE

ISN:   0
 BC:1     CC:141   D4:141   DC:142    E4:143    EC:144    F4:145   108:146   110:153
118:172
124:172  12C:172  15C:176  16E:235   176:236   17E:237   186:238   18A:239   190:240
198:241
1B4:258  1C0:258  1C8:258  1F8:259   204:263   20E:302   216:302   21E:303   226:304
22E:305
236:306  24A:307  252:314  25A:323   286:324   28C:385   294:386   29C:387   2A8:388
2B0:389
2B6:390  2BE:391  2C6:392  2E2:399   2F6:409   304:419   334:422   33E:452   346:453
34E:454
356:455  36A:456  372:500  37A:500   382:501   38A:502   392:503   39A:504   3AE:505
3B6:512
3BE:531  3CA:531  3D2:531  402:535   412:594   41A:595   422:596   42A:597   42E:598
434:599
43C:600  458:607  460:626  46C:626   474:626   4A4:630   4BE:631   504:690   50C:691
514:692
51C:693  524:694  52A:695  532:696   54E:703   556:722   562:722   56A:722   59A:726
5A4:754
5AC:755  5B4:756  5BC:757  5D0:758   5D8:765
```

Figure 4-4. Line Location Data Map

```
BUFFER_MAX_LENGTH            INTEGER      100                                        3      8
MAX_NUMBER_OF_WORDS          INTEGER      200                                        4     14
MAX_WORD_LENGTH              INTEGER      20                                         5     15

2  BUFFER(INPUT_INTERFACE)   ?BR7.C->2        CHAR(100)           EXT STATIC    DCL: 8
                                                                        387
2  BUFFER_LENGTH(INPUT_INTERFACE)  ?BR7.C->0 FIXED BIN(15)       EXT STATIC   DCL: 7
                                                                        388
2  COUNTER(WORD)             ?BR7.14->14      ARRAY FIXED BIN(15) EXT STATIC   DCL: 16
                                                                        631
2  END_OF_FILE(INPUT_INTERFACE)  ?BR7.C->66  BIT(1)              EXT STATIC   DCL: 9i
                                                                        409m
   ERROR                     ?BR7.0           PROCEDURE RECURSIVE EXT          DCL: 1
                                                                        323   419
   FIRST_TIME                ?BR7.38->0       BIT(1)              INT STATIC   DCL: 10i
                                                                        263   324m
2  H_BRA_PTR(H_EVA)          /EXP_PTR/-> 30   POINTER             BASED        DCL: 215  NO
REF
2  H_BRA_PTR(H_EVA)          /EXP_PTR/-> 30   POINTER             BASED        DCL: 364  NO
REF
2  H_BRA_PTR(H_EVA)          /EXP_PTR/-> 30   POINTER             BASED        DCL: 574  NO
REF
2  H_BRA_PTR(H_EVA)          /EXP_PTR/-> 30   POINTER             BASED        DCL: 670  NO
REF
2  H_CL_PROC(H_EVA)          /EXP_PTR/-> 4    ENTRY RECURSIVE     BASED        DCL: 193  NO
REF
2  H_CL_PROC(H_EVA)          /EXP_PTR/-> 4    ENTRY RECURSIVE     BASED        DCL: 342  NO
REF
2  H_CL_PROC(H_EVA)          /EXP_PTR/-> 4    ENTRY RECURSIVE     BASED        DCL: 552  NO
REF
2  H_CL_PROC(H_EVA)          /EXP_PTR/-> 4    ENTRY RECURSIVE     BASED        DCL: 648  NO
REF
   H_DFPRE_UCFM              ?BR7.34          ENTRY RECURSIVE     EXT          DCL: 433
                                                            455
   H_DFPRE_UCFM              ?BR7.34          ENTRY RECURSIVE     EXT          DCL: 735
                                                            757
   H_DFPRE_UOPF              ?BR7.2C          ENTRY RECURSIVE     EXT          DCL: 112
                                                            145
   H_DFPRE_UOPF              ?BR7.2C          ENTRY RECURSIVE     EXT          DCL: 273
                                                            306
   H_DFPRE_UOPF              ?BR7.2C          ENTRY RECURSIVE     EXT          DCL: 471
                                                            504
2  H_DL_PROC(H_EVA)          /EXP_PTR/-> 24   ENTRY RECURSIVE     BASED        DCL: 209  NO
REF
2  H_DL_PROC(H_EVA)          /EXP_PTR/-> 24   ENTRY RECURSIVE     BASED        DCL: 358  NO
REF
2  H_DL_PROC(H_EVA)          /EXP_PTR/-> 24   ENTRY RECURSIVE     BASED        DCL: 568  NO
REF
2  H_DL_PROC(H_EVA)          /EXP_PTR/-> 24   ENTRY RECURSIVE     BASED        DCL: 664  NO
REF
1  H_EVA                     H_EXEVA_PTR-> 0  STRUCTURE           BASED        DCL: 188  NO
REF
1  H_EVA                     H_EXEVA_PTR-> 0  STRUCTURE           BASED        DCL: 337  NO
REF
1  H_EVA                     H_EXEVA_PTR-> 0  STRUCTURE           BASED        DCL: 547  NO
REF
1  H_EVA                     H_EXEVA_PTR-> 0  STRUCTURE           BASED        DCL: 643  NO
REF
2  H_EXACCMODE(H_EXUCA_ON)   ?BR1.7B          CHAR(1)             AUTO         DCL: 131  NO
REF
2  H_EXACCMODE(H_EXUCA_ON)   ?BR1.7B          CHAR(1)             AUTO         DCL: 292  NO
REF
2  H_EXACCMODE(H_EXUCA_ON)   ?BR1.7B          CHAR(1)             AUTO         DCL: 490  NO
REF
2  H_EXAVAILSP(H_EXUCA_PT)   ?BR1.90          POINTER             AUTO         DCL: 233  NO
REF
2  H_EXAVAILSP(H_EXUCA_PT)   ?BR1.90          POINTER             AUTO         DCL: 592  NO
REF
2  H_EXAVAILSP(H_EXUCA_PT)   ?BR1.90          POINTER             AUTO         DCL: 688  NO
REF
2  H_EXCKPT(H_EXUCA_CL)      ?BR1.70          POINTER             AUTO         DCL: 448  NO
REF
2  H_EXCKPT(H_EXUCA_CL)      ?BR1.70          POINTER             AUTO         DCL: 750  NO
REF
2  H_EXCKPT2(H_EXUCA_CL)     ?BR1.74          POINTER             AUTO         DCL: 449  NO
REF
2  H_EXCKPT2(H_EXUCA_CL)     ?BR1.74          POINTER             AUTO         DCL: 751  NO
REF
```

```
2  H_EXCNT(H_EXUCA_CL)      ?BR1.60          POINTER              AUTO          DCL: 444  NO
REF
2  H_EXCNT(H_EXUCA_CL)      ?BR1.60          POINTER              AUTO          DCL: 746  NO
REF
2  H_EXEFN(H_EXUCA_ON)      ?BR1.6A          POINTER              AUTO          DCL: 126  NO
REF
2  H_EXEFN(H_EXUCA_ON)      ?BR1.6A          POINTER              AUTO          DCL: 287  NO
REF
2  H_EXEFN(H_EXUCA_ON)      ?BR1.6A          POINTER              AUTO          DCL: 485  NO
REF
2  H_EXEVA_PTR(H_EXFCB_C)   /EXP_PTR/-> 1    POINTER              BASED         DCL: 186
                                                            188d  241b
2  H_EXEVA_PTR(H_EXFCB_C)   /EXP_PTR/-> 1    POINTER              BASED         DCL: 335
                                                            337d  392b
2  H_EXEVA_PTR(H_EXFCB_C)   /EXP_PTR/-> 1    POINTER              BASED         DCL: 545
                                                            547d  600b
2  H_EXEVA_PTR(H_EXFCB_C)   /EXP_PTR/-> 1    POINTER              BASED         DCL: 641
                                                            643d  696b
2  H_EXEXLEN(H_EXUCA_PT)    ?BR1.88          FIXED BIN(15)        AUTO          DCL: 230  NO
REF
2  H_EXEXLEN(H_EXUCA_GT)    ?BR1.70          FIXED BIN(15)        AUTO          DCL: 379  NO
REF
2  H_EXEXLEN(H_EXUCA_PT)    ?BR1.88          FIXED BIN(15)        AUTO          DCL: 589  NO
REF
2  H_EXEXLEN(H_EXUCA_PT)    ?BR1.88          FIXED BIN(15)        AUTO          DCL: 685  NO
REF
2  H_EXEXTMASK(H_EXUCA_ON)  ?BR1.7C          BIT(32)              AUTO          DCL: 132  NO
REF
2  H_EXEXTMASK(H_EXUCA_ON)  ?BR1.7C          BIT(32)              AUTO          DCL: 293  NO
REF
2  H_EXEXTMASK(H_EXUCA_ON)  ?BR1.7C          BIT(32)              AUTO          DCL: 491  NO
REF
2  H_EXEXTPTR(H_EXUCA_PT)   ?BR1.8C          POINTER              AUTO          DCL: 232  NO
REF
2  H_EXEXTPTR(H_EXUCA_GT)   ?BR1.74          POINTER              AUTO          DCL: 381  NO
REF
2  H_EXEXTPTR(H_EXUCA_PT)   ?BR1.8C          POINTER              AUTO          DCL: 591  NO
REF
2  H_EXEXTPTR(H_EXUCA_PT)   ?BR1.8C          POINTER              AUTO          DCL: 687  NO
REF
1  H_EXFCB_C                H_EXFCB_PTR-> 0  STRUCTURE            BASED         DCL: 333  NO
REF
1  H_EXFCB_C                H_EXFCB_PTR-> 0  STRUCTURE            BASED         DCL: 543  NO
REF
1  H_EXFCB_C                H_EXFCB_PTR-> 0  STRUCTURE            BASED         DCL: 639  NO
REF
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.94          POINTER              AUTO          DCL: 115
                                                            141m  146
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.60          POINTER              AUTO          DCL: 218
                                                            184d  188b  235m  241b
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.94          POINTER              AUTO          DCL: 276
                                                            302m  307
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.80          POINTER              AUTO          DCL: 367
                                                            333d  337b  385m  392b
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.80          POINTER              AUTO          DCL: 436
                                                            452m  456
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.94          POINTER              AUTO          DCL: 474
                                                            500m  505
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.60          POINTER              AUTO          DCL: 577
                                                            543d  547b  594m  600b
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.60          POINTER              AUTO          DCL: 673
                                                            639d  643b  690m  696b
2  H_EXFCB_PTR(H_EXUCA)     ?BR1.80          POINTER              AUTO          DCL: 738
                                                            754m  758
2  H_EXFILEATTR(H_EXUCA_ON) ?BR1.64          POINTER              AUTO          DCL: 124  NO
REF
2  H_EXFILEATTR(H_EXUCA_ON) ?BR1.64          POINTER              AUTO          DCL: 285  NO
REF
2  H_EXFILEATTR(H_EXUCA_ON) ?BR1.64          POINTER              AUTO          DCL: 483  NO
REF
2  H_EXGT_RFU(H_EXUCA_GT)   ?BR1.7A          CHAR(2)              AUTO          DCL: 383  NO
REF
2  H_EXILNVALUE(H_EXUCA)    ?BR1.A4          FIXED BIN(15)        AUTO          DCL: 118  NO
REF
2  H_EXILNVALUE(H_EXUCA)    ?BR1.70          FIXED BIN(15)        AUTO          DCL: 221
                                                            238m
2  H_EXILNVALUE(H_EXUCA)    ?BR1.A4          FIXED BIN(15)        AUTO          DCL: 279  NO
REF
2  H_EXILNVALUE(H_EXUCA)    ?BR1.90          FIXED BIN(15)        AUTO          DCL: 370
                                                            390m
```

```
2  H_EXILNVALUE(H_EXUCA)    ?BR1.90          FIXED BIN(15)       AUTO          DCL: 439  NO
REF
2  H_EXILNVALUE(H_EXUCA)    ?BR1.A4          FIXED BIN(15)       AUTO          DCL: 477  NO
REF
2  H_EXILNVALUE(H_EXUCA)    ?BR1.70          FIXED BIN(15)       AUTO          DCL: 580
                                                            597m
2  H_EXILNVALUE(H_EXUCA)    ?BR1.70          FIXED BIN(15)       AUTO          DCL: 676
                                                            693m
2  H_EXILNVALUE(H_EXUCA)    ?BR1.90          FIXED BIN(15)       AUTO          DCL: 741  NO
REF
2  H_EXINADDR(H_EXUCA_ON)   ?BR1.88          POINTER             AUTO          DCL: 135  NO
REF
2  H_EXINADDR(H_EXUCA_PT)   ?BR1.7C          POINTER             AUTO          DCL: 227  NO
REF
2  H_EXINADDR(H_EXUCA_ON)   ?BR1.88          POINTER             AUTO          DCL: 296  NO
REF
2  H_EXINADDR(H_EXUCA_GT)   ?BR1.64          POINTER             AUTO          DCL: 376  NO
REF
2  H_EXINADDR(H_EXUCA_ON)   ?BR1.88          POINTER             AUTO          DCL: 494  NO
REF
2  H_EXINADDR(H_EXUCA_PT)   ?BR1.7C          POINTER             AUTO          DCL: 586  NO
REF
2  H_EXINADDR(H_EXUCA_PT)   ?BR1.7C          POINTER             AUTO          DCL: 682  NO
REF
2  H_EXINKEY(H_EXUCA_PT)    ?BR1.78          POINTER             AUTO          DCL: 226  NO
REF
2  H_EXINKEY(H_EXUCA_GT)    ?BR1.60          POINTER             AUTO          DCL: 375  NO
REF
2  H_EXINKEY(H_EXUCA_PT)    ?BR1.78          POINTER             AUTO          DCL: 585  NO
REF
2  H_EXINKEY(H_EXUCA_PT)    ?BR1.78          POINTER             AUTO          DCL: 681  NO
REF
2  H_EXINKEYID(H_EXUCA_GT)  ?BR1.78          FIXED BIN(15)       AUTO          DCL: 382  NO
REF
2  H_EXLBADDR(H_EXUCA_ON)   ?BR1.72          POINTER             AUTO          DCL: 128  NO
REF
2  H_EXLBADDR(H_EXUCA_ON)   ?BR1.72          POINTER             AUTO          DCL: 289  NO
REF
2  H_EXLBADDR(H_EXUCA_CL)   ?BR1.6A          CHAR(5)             AUTO          DCL: 446  NO
REF
2  H_EXLBADDR(H_EXUCA_ON)   ?BR1.72          POINTER             AUTO          DCL: 487  NO
REF
2  H_EXLBADDR(H_EXUCA_CL)   ?BR1.6A          CHAR(5)             AUTO          DCL: 748  NO
REF
2  H_EXNUMBLK(H_EXUCA_ON)   ?BR1.6E          POINTER             AUTO          DCL: 127  NO
REF
2  H_EXNUMBLK(H_EXUCA_ON)   ?BR1.6E          POINTER             AUTO          DCL: 288  NO
REF
2  H_EXNUMBLK(H_EXUCA_CL)   ?BR1.64          CHAR(6)             AUTO          DCL: 445  NO
REF
2  H_EXNUMBLK(H_EXUCA_ON)   ?BR1.6E          POINTER             AUTO          DCL: 486  NO
REF
2  H_EXNUMBLK(H_EXUCA_CL)   ?BR1.64          CHAR(6)             AUTO          DCL: 747  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.A0          POINTER             AUTO          DCL: 117  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.6C          POINTER             AUTO          DCL: 220  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.A0          POINTER             AUTO          DCL: 278  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.8C          POINTER             AUTO          DCL: 369
                                                            388m
2  H_EXOLNPTR(H_EXUCA)      ?BR1.8C          POINTER             AUTO          DCL: 438  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.A0          POINTER             AUTO          DCL: 476  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.6C          POINTER             AUTO          DCL: 579  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.6C          POINTER             AUTO          DCL: 675  NO
REF
2  H_EXOLNPTR(H_EXUCA)      ?BR1.8C          POINTER             AUTO          DCL: 740  NO
REF
2  H_EXORGCHECK(H_EXUCA_ON) ?BR1.8E          BIT(8)              AUTO          DCL: 137  NO
REF
2  H_EXORGCHECK(H_EXUCA_ON) ?BR1.8E          BIT(8)              AUTO          DCL: 298  NO
REF
2  H_EXORGCHECK(H_EXUCA_ON) ?BR1.8E          BIT(8)              AUTO          DCL: 496  NO
REF
2  H_EXOUTADDR(H_EXUCA_PT)  ?BR1.84          POINTER             AUTO          DCL: 229  NO
REF
```

```
2  H_EXOUTADDR(H_EXUCA_GT)  ?BR1.6C          POINTER             AUTO         DCL: 378  NO
REF
2  H_EXOUTADDR(H_EXUCA_PT)  ?BR1.84          POINTER             AUTO         DCL: 588  NO
REF
2  H_EXOUTADDR(H_EXUCA_PT)  ?BR1.84          POINTER             AUTO         DCL: 684  NO
REF
2  H_EXOUTKEY(H_EXUCA_PT)   ?BR1.80          POINTER             AUTO         DCL: 228  NO
REF
2  H_EXOUTKEY(H_EXUCA_GT)   ?BR1.68          POINTER             AUTO         DCL: 377  NO
REF
2  H_EXOUTKEY(H_EXUCA_PT)   ?BR1.80          POINTER             AUTO         DCL: 587  NO
REF
2  H_EXOUTKEY(H_EXUCA_PT)   ?BR1.80          POINTER             AUTO         DCL: 683  NO
REF
2  H_EXPARMASK(H_EXUCA)     ?BR1.98          BIT(32)             AUTO         DCL: 116
                                                          144m
2  H_EXPARMASK(H_EXUCA)     ?BR1.64          BIT(32)             AUTO         DCL: 219
                                                          240m
2  H_EXPARMASK(H_EXUCA)     ?BR1.98          BIT(32)             AUTO         DCL: 277
                                                          305m
2  H_EXPARMASK(H_EXUCA)     ?BR1.84          BIT(32)             AUTO         DCL: 368
                                                          391m
2  H_EXPARMASK(H_EXUCA)     ?BR1.84          BIT(32)             AUTO         DCL: 437
                                                          454m
2  H_EXPARMASK(H_EXUCA)     ?BR1.98          BIT(32)             AUTO         DCL: 475
                                                          503m
2  H_EXPARMASK(H_EXUCA)     ?BR1.64          BIT(32)             AUTO         DCL: 578
                                                          599m
2  H_EXPARMASK(H_EXUCA)     ?BR1.64          BIT(32)             AUTO         DCL: 674
                                                          695m
2  H_EXPARMASK(H_EXUCA)     ?BR1.84          BIT(32)             AUTO         DCL: 739
                                                          756m
2  H_EXPGID(H_EXUCA_ON)     ?BR1.60          POINTER             AUTO         DCL: 123  NO
REF
2  H_EXPGID(H_EXUCA_ON)     ?BR1.60          POINTER             AUTO         DCL: 284  NO
REF
2  H_EXPGID(H_EXUCA_ON)     ?BR1.60          POINTER             AUTO         DCL: 482  NO
REF
2  H_EXPMD(H_EXUCA_ON)      ?BR1.68          CHAR(2)             AUTO         DCL: 125
                                                          142m
2  H_EXPMD(H_EXUCA_ON)      ?BR1.68          CHAR(2)             AUTO         DCL: 286
                                                          303m
2  H_EXPMD(H_EXUCA_ON)      ?BR1.68          CHAR(2)             AUTO         DCL: 484
                                                          501m
2  H_EXPREASONS(H_EXUCA_CL) ?BR1.7C          POINTER             AUTO         DCL: 451  NO
REF
2  H_EXPREASONS(H_EXUCA_CL) ?BR1.7C          POINTER             AUTO         DCL: 753  NO
REF
2  H_EXRECOV(H_EXUCA_CL)    ?BR1.78          POINTER             AUTO         DCL: 450  NO
REF
2  H_EXRECOV(H_EXUCA_CL)    ?BR1.78          POINTER             AUTO         DCL: 752  NO
REF
2  H_EXRECTYPE(H_EXUCA_PT)  ?BR1.94          BIT(16)             AUTO         DCL: 234  NO
REF
2  H_EXRECTYPE(H_EXUCA_GT)  ?BR1.7C          POINTER             AUTO         DCL: 384  NO
REF
2  H_EXRECTYPE(H_EXUCA_PT)  ?BR1.94          BIT(16)             AUTO         DCL: 593  NO
REF
2  H_EXRECTYPE(H_EXUCA_PT)  ?BR1.94          BIT(16)             AUTO         DCL: 689  NO
REF
2  H_EXREFMODE(H_EXUCA_ON)  ?BR1.7A          CHAR(1)             AUTO         DCL: 130  NO
REF
2  H_EXREFMODE(H_EXUCA_ON)  ?BR1.7A          CHAR(1)             AUTO         DCL: 291  NO
REF
2  H_EXREFMODE(H_EXUCA_ON)  ?BR1.7A          CHAR(1)             AUTO         DCL: 489  NO
REF
2  H_EXRESTART(H_EXUCA_ON)  ?BR1.84          POINTER             AUTO         DCL: 134  NO
REF
2  H_EXRESTART(H_EXUCA_ON)  ?BR1.84          POINTER             AUTO         DCL: 295  NO
REF
2  H_EXRESTART(H_EXUCA_ON)  ?BR1.84          POINTER             AUTO         DCL: 493  NO
REF
2  H_EXRFLDEF(H_EXUCA_ON)   ?BR1.80          POINTER             AUTO         DCL: 133  NO
REF
2  H_EXRFLDEF(H_EXUCA_ON)   ?BR1.80          POINTER             AUTO         DCL: 294  NO
REF
2  H_EXRFLDEF(H_EXUCA_ON)   ?BR1.80          POINTER             AUTO         DCL: 492  NO
REF
2  H_EXSECIDX(H_EXUCA_ON)   ?BR1.8C          FIXED BIN(15)       AUTO         DCL: 136  NO
REF
```

```
2  H_EXSECIDX(H_EXUCA_ON)   ?BR1.8C          FIXED BIN(15)       AUTO          DCL: 297  NO
REF
2  H_EXSECIDX(H_EXUCA_ON)   ?BR1.8C          FIXED BIN(15)       AUTO          DCL: 495  NO
REF
2  H_EXSELECT(H_EXUCA_ON)   ?BR1.8F          CHAR(1)             AUTO          DCL: 138  NO
REF
2  H_EXSELECT(H_EXUCA_ON)   ?BR1.8F          CHAR(1)             AUTO          DCL: 299  NO
REF
2  H_EXSELECT(H_EXUCA_ON)   ?BR1.8F          CHAR(1)             AUTO          DCL: 497  NO
REF
2  H_EXSPECPTR(H_EXUCA)     ?BR1.A8          POINTER             AUTO          DCL: 119
                                                          141m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.74          POINTER             AUTO          DCL: 222
                                                          236m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.A8          POINTER             AUTO          DCL: 280
                                                          302m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.94          POINTER             AUTO          DCL: 371
                                                          386m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.94          POINTER             AUTO          DCL: 440
                                                          453m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.A8          POINTER             AUTO          DCL: 478
                                                          500m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.74          POINTER             AUTO          DCL: 581
                                                          595m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.74          POINTER             AUTO          DCL: 677
                                                          691m
2  H_EXSPECPTR(H_EXUCA)     ?BR1.94          POINTER             AUTO          DCL: 742
                                                          755m
2  H_EXSRCHLEN(H_EXUCA_PT)  ?BR1.8A          FIXED BIN(15)       AUTO          DCL: 231  NO
REF
2  H_EXSRCHLEN(H_EXUCA_GT)  ?BR1.72          FIXED BIN(15)       AUTO          DCL: 380  NO
REF
2  H_EXSRCHLEN(H_EXUCA_PT)  ?BR1.8A          FIXED BIN(15)       AUTO          DCL: 590  NO
REF
2  H_EXSRCHLEN(H_EXUCA_PT)  ?BR1.8A          FIXED BIN(15)       AUTO          DCL: 686  NO
REF
2  H_EXSUBFILE(H_EXUCA_ON)  ?BR1.76          POINTER             AUTO          DCL: 129  NO
REF
2  H_EXSUBFILE(H_EXUCA_ON)  ?BR1.76          POINTER             AUTO          DCL: 290  NO
REF
2  H_EXSUBFILE(H_EXUCA_ON)  ?BR1.76          POINTER             AUTO          DCL: 488  NO
REF
2  H_EXSUBREF(H_EXUCA)      ?BR1.A6          FIXED BIN(15)       AUTO          DCL: 118  NO
REF
2  H_EXSUBREF(H_EXUCA)      ?BR1.72          FIXED BIN(15)       AUTO          DCL: 221
                                                          239m
2  H_EXSUBREF(H_EXUCA)      ?BR1.A6          FIXED BIN(15)       AUTO          DCL: 279  NO
REF
2  H_EXSUBREF(H_EXUCA)      ?BR1.92          FIXED BIN(15)       AUTO          DCL: 370
                                                          389m
2  H_EXSUBREF(H_EXUCA)      ?BR1.92          FIXED BIN(15)       AUTO          DCL: 439  NO
REF
2  H_EXSUBREF(H_EXUCA)      ?BR1.A6          FIXED BIN(15)       AUTO          DCL: 477  NO
REF
2  H_EXSUBREF(H_EXUCA)      ?BR1.72          FIXED BIN(15)       AUTO          DCL: 580
                                                          598m
2  H_EXSUBREF(H_EXUCA)      ?BR1.72          FIXED BIN(15)       AUTO          DCL: 676
                                                          694m
2  H_EXSUBREF(H_EXUCA)      ?BR1.92          FIXED BIN(15)       AUTO          DCL: 741  NO
REF
2  H_EXTYPE(H_EXFCB_C)      /EXP_PTR/-> 0    CHAR(1)             BASED         DCL: 185  NO
REF
2  H_EXTYPE(H_EXFCB_C)      /EXP_PTR/-> 0    CHAR(1)             BASED         DCL: 334  NO
REF
2  H_EXTYPE(H_EXFCB_C)      /EXP_PTR/-> 0    CHAR(1)             BASED         DCL: 544  NO
REF
2  H_EXTYPE(H_EXFCB_C)      /EXP_PTR/-> 0    CHAR(1)             BASED         DCL: 640  NO
REF
1  H_EXUCA                  ?BR1.94          STRUCTURE           AUTO          DCL: 114
                                                          145r
1  H_EXUCA                  ?BR1.60          STRUCTURE           AUTO          DCL: 217
                                                          241r
1  H_EXUCA                  ?BR1.94          STRUCTURE           AUTO          DCL: 275
                                                          306r
1  H_EXUCA                  ?BR1.80          STRUCTURE           AUTO          DCL: 366
                                                          392r
1  H_EXUCA                  ?BR1.80          STRUCTURE           AUTO          DCL: 435
                                                          455r
1  H_EXUCA                  ?BR1.94          STRUCTURE           AUTO          DCL: 473
                                                          504r
```

```
1  H_EXUCA                  ?BR1.60          STRUCTURE            AUTO         DCL: 576
                                                           600r
1  H_EXUCA                  ?BR1.60          STRUCTURE            AUTO         DCL: 672
                                                           696r
1  H_EXUCA                  ?BR1.80          STRUCTURE            AUTO         DCL: 737
                                                           757r
1  H_EXUCA_CL               ?BR1.60          STRUCTURE            AUTO         DCL: 441
                                                           453
1  H_EXUCA_CL               ?BR1.60          STRUCTURE            AUTO         DCL: 743
                                                           755
2  H_EXUCA_CL01(H_EXUCA_CL) ?BR1.6F          CHAR(1)              AUTO         DCL: 447  NO
REF
2  H_EXUCA_CL01(H_EXUCA_CL) ?BR1.6F          CHAR(1)              AUTO         DCL: 749  NO
REF
1  H_EXUCA_GT               ?BR1.60          STRUCTURE            AUTO         DCL: 372
                                                           386
1  H_EXUCA_ON               ?BR1.60          STRUCTURE            AUTO         DCL: 120
                                                           141
1  H_EXUCA_ON               ?BR1.60          STRUCTURE            AUTO         DCL: 281
                                                           302
1  H_EXUCA_ON               ?BR1.60          STRUCTURE            AUTO         DCL: 479
                                                           500
1  H_EXUCA_PT               ?BR1.78          STRUCTURE            AUTO         DCL: 223
                                                           236
1  H_EXUCA_PT               ?BR1.78          STRUCTURE            AUTO         DCL: 582
                                                           595
1  H_EXUCA_PT               ?BR1.78          STRUCTURE            AUTO         DCL: 678
                                                           691
2  H_EXUTILITY(H_EXUCA_ON)  ?BR1.90          POINTER              AUTO         DCL: 139  NO
REF
2  H_EXUTILITY(H_EXUCA_ON)  ?BR1.90          POINTER              AUTO         DCL: 300  NO
REF
2  H_EXUTILITY(H_EXUCA_ON)  ?BR1.90          POINTER              AUTO         DCL: 498  NO
REF
2  H_EXWAPTR(H_EXUCA)       ?BR1.9C          POINTER              AUTO         DCL: 117
                                                           143m
2  H_EXWAPTR(H_EXUCA)       ?BR1.68          POINTER              AUTO         DCL: 220
                                                           237m
2  H_EXWAPTR(H_EXUCA)       ?BR1.9C          POINTER              AUTO         DCL: 278
                                                           304m
2  H_EXWAPTR(H_EXUCA)       ?BR1.88          POINTER              AUTO         DCL: 369
                                                           387m
2  H_EXWAPTR(H_EXUCA)       ?BR1.88          POINTER              AUTO         DCL: 438  NO
REF
2  H_EXWAPTR(H_EXUCA)       ?BR1.9C          POINTER              AUTO         DCL: 476
                                                           502m
2  H_EXWAPTR(H_EXUCA)       ?BR1.68          POINTER              AUTO         DCL: 579
                                                           596m
2  H_EXWAPTR(H_EXUCA)       ?BR1.68          POINTER              AUTO         DCL: 675
                                                           692m
2  H_EXWAPTR(H_EXUCA)       ?BR1.88          POINTER              AUTO         DCL: 740  NO
REF
2  H_FO_PROC(H_EVA)         /EXP_PTR/-> 2C   ENTRY RECURSIVE      BASED        DCL: 213  NO
REF
2  H_FO_PROC(H_EVA)         /EXP_PTR/-> 2C   ENTRY RECURSIVE      BASED        DCL: 362  NO
REF
2  H_FO_PROC(H_EVA)         /EXP_PTR/-> 2C   ENTRY RECURSIVE      BASED        DCL: 572  NO
REF
2  H_FO_PROC(H_EVA)         /EXP_PTR/-> 2C   ENTRY RECURSIVE      BASED        DCL: 668  NO
REF
2  H_GT_PROC(H_EVA)         /EXP_PTR/-> 10   ENTRY RECURSIVE      BASED        DCL: 199  NO
REF
2  H_GT_PROC(H_EVA)         /EXP_PTR/-> 10   ENTRY RECURSIVE      BASED        DCL: 348
                                                           392
2  H_GT_PROC(H_EVA)         /EXP_PTR/-> 10   ENTRY RECURSIVE      BASED        DCL: 558  NO
REF
2  H_GT_PROC(H_EVA)         /EXP_PTR/-> 10   ENTRY RECURSIVE      BASED        DCL: 654  NO
REF
2  H_IN_PROC(H_EVA)         /EXP_PTR/-> 28   ENTRY RECURSIVE      BASED        DCL: 211  NO
REF
2  H_IN_PROC(H_EVA)         /EXP_PTR/-> 28   ENTRY RECURSIVE      BASED        DCL: 360  NO
REF
2  H_IN_PROC(H_EVA)         /EXP_PTR/-> 28   ENTRY RECURSIVE      BASED        DCL: 570  NO
REF
2  H_IN_PROC(H_EVA)         /EXP_PTR/-> 28   ENTRY RECURSIVE      BASED        DCL: 666  NO
REF
   H_LENGTH                 ?BR1.80          FIXED BIN(15)        AUTO         DCL: 171
                                                           172m  172r
   H_LENGTH                 ?BR1.80          FIXED BIN(15)        AUTO         DCL: 257
                                                           258m  258r
   H_LENGTH                ?BR1.80          FIXED BIN(15)       AUTO          DCL: 530
                                                          531m  531r
   H_LENGTH                ?BR1.80          FIXED BIN(15)       AUTO          DCL: 625
                                                          626m  626r
   H_LENGTH                ?BR1.80          FIXED BIN(15)       AUTO          DCL: 721
                                                          722m  722r
   H_NAME                  ?BR1.60          CHAR(32)            AUTO          DCL: 170
                                                          172m  172r
   H_NAME                  ?BR1.60          CHAR(32)            AUTO          DCL: 256
                                                          258m  258r
   H_NAME                  ?BR1.60          CHAR(32)            AUTO          DCL: 529
                                                          531m  531r
   H_NAME                  ?BR1.60          CHAR(32)            AUTO          DCL: 624
                                                          626m  626r
   H_NAME                  ?BR1.60          CHAR(32)            AUTO          DCL: 720
                                                          722m  722r
2  H_NT_PROC(H_EVA)        /EXP_PTR/-> 18   ENTRY RECURSIVE     BASED         DCL: 203  NO
REF
2  H_NT_PROC(H_EVA)        /EXP_PTR/-> 18   ENTRY RECURSIVE     BASED         DCL: 352  NO
REF
2  H_NT_PROC(H_EVA)        /EXP_PTR/-> 18   ENTRY RECURSIVE     BASED         DCL: 562  NO
REF
2  H_NT_PROC(H_EVA)        /EXP_PTR/-> 18   ENTRY RECURSIVE     BASED         DCL: 658  NO
REF
2  H_ON_PROC(H_EVA)        /EXP_PTR/-> 0    ENTRY RECURSIVE     BASED         DCL: 191  NO
REF
2  H_ON_PROC(H_EVA)        /EXP_PTR/-> 0    ENTRY RECURSIVE     BASED         DCL: 340  NO
REF
2  H_ON_PROC(H_EVA)        /EXP_PTR/-> 0    ENTRY RECURSIVE     BASED         DCL: 550  NO
REF
2  H_ON_PROC(H_EVA)        /EXP_PTR/-> 0    ENTRY RECURSIVE     BASED         DCL: 646  NO
REF
2  H_PN_PROC(H_EVA)        /EXP_PTR/-> 8    ENTRY RECURSIVE     BASED         DCL: 195  NO
REF
2  H_PN_PROC(H_EVA)        /EXP_PTR/-> 8    ENTRY RECURSIVE     BASED         DCL: 344  NO
REF
2  H_PN_PROC(H_EVA)        /EXP_PTR/-> 8    ENTRY RECURSIVE     BASED         DCL: 554  NO
REF
2  H_PN_PROC(H_EVA)        /EXP_PTR/-> 8    ENTRY RECURSIVE     BASED         DCL: 650  NO
REF
2  H_PO_PROC(H_EVA)        /EXP_PTR/-> 1C   ENTRY RECURSIVE     BASED         DCL: 205  NO
REF
2  H_PO_PROC(H_EVA)        /EXP_PTR/-> 1C   ENTRY RECURSIVE     BASED         DCL: 354  NO
REF
2  H_PO_PROC(H_EVA)        /EXP_PTR/-> 1C   ENTRY RECURSIVE     BASED         DCL: 564  NO
REF
2  H_PO_PROC(H_EVA)        /EXP_PTR/-> 1C   ENTRY RECURSIVE     BASED         DCL: 660  NO
REF
2  H_PT_PROC(H_EVA)        /EXP_PTR/-> 14   ENTRY RECURSIVE     BASED         DCL: 201
                                                          241
2  H_PT_PROC(H_EVA)        /EXP_PTR/-> 14   ENTRY RECURSIVE     BASED         DCL: 350  NO
REF
2  H_PT_PROC(H_EVA)        /EXP_PTR/-> 14   ENTRY RECURSIVE     BASED         DCL: 560
                                                          600
2  H_PT_PROC(H_EVA)        /EXP_PTR/-> 14   ENTRY RECURSIVE     BASED         DCL: 656
                                                          696
2  H_PX_PROC(H_EVA)        /EXP_PTR/-> 20   ENTRY RECURSIVE     BASED         DCL: 207  NO
REF
2  H_PX_PROC(H_EVA)        /EXP_PTR/-> 20   ENTRY RECURSIVE     BASED         DCL: 356  NO
REF
2  H_PX_PROC(H_EVA)        /EXP_PTR/-> 20   ENTRY RECURSIVE     BASED         DCL: 566  NO
REF
2  H_PX_PROC(H_EVA)        /EXP_PTR/-> 20   ENTRY RECURSIVE     BASED         DCL: 662  NO
REF
1  H_S_INPUT               ?BR7.20->0       STRUCTURE           EXT STATIC    DCL: 28
                                                          25d
1  H_S_OUTPUT              ?BR7.28->0       STRUCTURE           EXT STATIC    DCL: 71
                                                          68d
2  H_SK_PROC(H_EVA)        /EXP_PTR/-> C    ENTRY RECURSIVE     BASED         DCL: 197  NO
REF
2  H_SK_PROC(H_EVA)        /EXP_PTR/-> C    ENTRY RECURSIVE     BASED         DCL: 346  NO
REF
2  H_SK_PROC(H_EVA)        /EXP_PTR/-> C    ENTRY RECURSIVE     BASED         DCL: 556  NO
REF
2  H_SK_PROC(H_EVA)        /EXP_PTR/-> C    ENTRY RECURSIVE     BASED         DCL: 652  NO
REF
   H_STND2_TDF1            ?BR7.18->0       CHAR(52)            EXT STATIC    DCL: 22
                                                          31d   74d
   H_TASKM_UABT            ?BR7.30          ENTRY RECURSIVE     EXT           DCL: 169
                                                          172
   H_TASKM_UABT             ?BR7.30          ENTRY RECURSIVE     EXT          DCL: 255
                                                          258
   H_TASKM_UABT             ?BR7.30          ENTRY RECURSIVE     EXT          DCL: 528
                                                          531
   H_TASKM_UABT             ?BR7.30          ENTRY RECURSIVE     EXT          DCL: 623
                                                          626
   H_TASKM_UABT             ?BR7.30          ENTRY RECURSIVE     EXT          DCL: 719
                                                          722
   I                        ?BR1.5C          FIXED BIN(15)       AUTO         DCL: 424
                                                          630m  631   631
1  INPUT                    ?BR7.1C->0       STRUCTURE           EXT STATIC   DCL: 24   NO
REF
1  INPUT_INTERFACE          ?BR7.C->0        STRUCTURE           EXT STATIC   DCL: 6    NO
REF
2  INPUTFCB_PTR(INPUT)      ?BR7.1C->0       POINTER             EXT STATIC   DCL: 25i
                                                          302   307m  385   452   456m
2  INPUTRFUUCA(INPUT)       ?BR7.1C->4       POINTER             EXT STATIC   DCL: 26   NO
REF
   LINE                     ?BR1.C           CHAR(80)            AUTO         DCL: 11
                                                          176m  237   535m  596   631m  692
   MESSAGE                  ?BR1.4->0        CHAR(*)             PARAM        DCL: 2
                                                          176
2  NAME(WORD)               ?BR7.14->0       ARRAY CHAR(20)      EXT STATIC   DCL: 15
                                                          631
   NO_REF_SO_NOT_ALLOC      **NOT ALLOC**    POINTER             AUTO         DCL: 12   NO
REF
   NUMBER_OF_WORDS          ?BR7.10->0       FIXED BIN(15)       EXT STATIC   DCL: 13
                                                          630
1  OUTPUT                   ?BR7.24->0       STRUCTURE           EXT STATIC   DCL: 67   NO
REF
2  OUTPUTFCB_PTR(OUTPUT)    ?BR7.24->0       POINTER             EXT STATIC   DCL: 68i
                                                          141   146m  235   500   505m  594
690   754   758m
2  OUTPUTRFUUCA(OUTPUT)     ?BR7.24->4       POINTER             EXT STATIC   DCL: 69   NO
REF
   READ_IN_BUFFER           ?BR7.4           ENTRY RECURSIVE     EXT          DCL: 262  NO
REF
1  WORD                     ?BR7.14->0       ARRAY STRUCTURE     EXT STATIC   DCL: 14   NO
REF
   WRITE_WORD_ARRAY         ?BR7.8           ENTRY RECURSIVE     EXT          DCL: 423  NO
REF
        + + + NO ERROR MESSAGES + + +
             OBJECT CODE PRODUCED
```

Figure 4-5. Cross Reference Listings

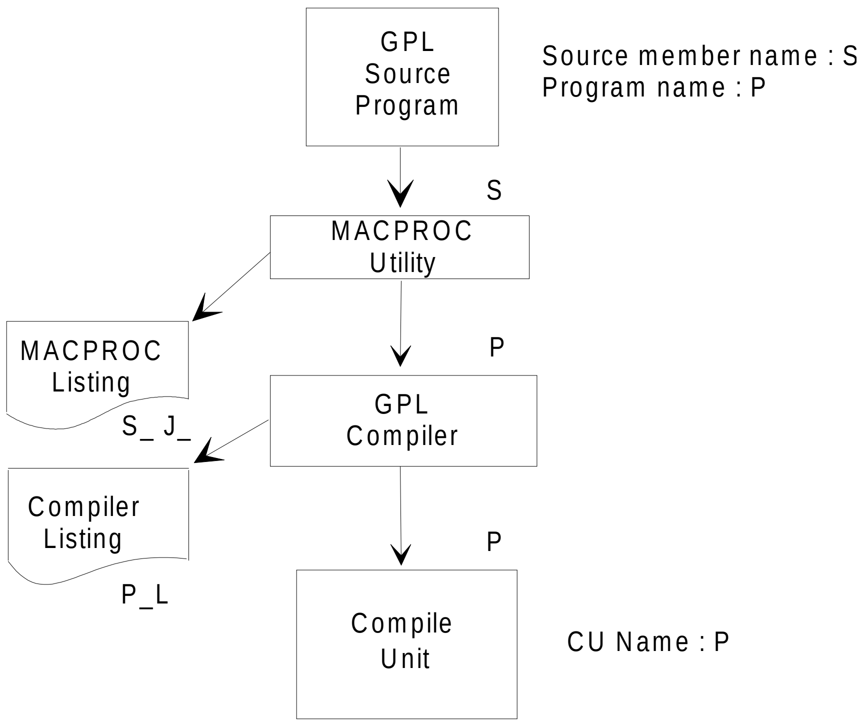


Figure 4-6. Naming Conventions

5. Linking and Communication

Every compile unit must be processed by the LINKER utility before execution. The LINKER builds an executable load module from a compile unit or set of compile units. It is possible to link compile units which are produced by different compilers, for example, the source programs of which were written in different languages.

5.1 USING THE LINKER

5.1.1 Linking In Batch

The LINKER is called by the JCL statement LINKER. The following example illustrates the simple use of this statement:

```
$JOB...
 LIB CU INLIB1 = CU.LIB;
 LINKER GPROG1 OUTLIB = LM.LIB;
$ENDJOB;
```

The LIB statement sets up a search path for LINKER. LINKER will produce a load module called GPROG1 (the entry-name is assumed to be also GPROG1) which will be stored in the library LM.LIB. For a complete description of the JCL statement LINKER see the LINKER manual. The GPL programmer should have a copy of this manual: in addition to the JCL for LINKER, it gives a detailed description of the printed output produced by LINKER and of the advanced LINKER commands. It contains a considerable amount of background information which is of interest to the GPL user. An example of a LINKER listing is given as part of the example GPL program in Appendix C. The LINKER statement format is given in Figure 5-1 as a memory aid only.

```
LINKER      load module name
                       *
           [INLIB  =  ( input - library - description ]
                       ( output - library - description )
            OUTLIB  =  TEMP
            COMFILE  =  ( sequential - input - file - description )
            COMMAND  =  ' command  [ command ] ...'
            ENTRY  =  entry - name [ COMFAC ]
            PRTFILE  =  ( print - file - description )
            PRTLIB  =  ( print - library - description )
           [ STEPOPT  =  ( step - parameters ) ] ;
```

Figure 5-1. LINKER Statement Format

5.1.2 Interactive Linking

In interactive mode, the LINKER may be called using the GCL statement LINK_PG. LINK_PG is described fully in Volume II of the IOF Terminal User's Reference Manual.

The remainder of this section, which should be read in conjunction with the LINKER manual, is concerned with multitasking and associated topics. It covers data sharing, segmentation and procedure references.

5.2 DATA SHARING

5.2.1 Data Sharing Between Procedures

Data which is to be shared by different procedures must be declared with storage class STATIC and scope EXTERNAL in each of the sharing procedures. For example, if the declaration: DCL A CHAR(20) STATIC EXTERNAL; is used in two procedures, then the same character variable A can be referenced by both procedures. The variable must be declared by using exactly the same declaration statement in each procedure. STATIC EXTERNAL data is written in the BLANK segment, unless a WITHIN clause which specifies a segment name is used (see below). Data declaration and the concepts of storage class and scope are explained in the GPL Reference Manual.

5.2.2 Data Sharing Between Tasks

The above method of sharing data applies only to procedures which are part of the same task (a program which does not use the multitasking facilities is by default a monotask program). In a multitask program, data which is to be shared by different tasks must be written in a common segment by means of the WITHIN clause in the declaration statement. The clause: WITHIN X (TT(2)); specifies that all data associated with X is to be stored in the same segment. Since the segment is to be shared, it is declared with the attribute Table Type 2 which corresponds to SHRLEVEL=2 in the LINKER command MSEGAT. (Segment sharability is discussed in the LINKER manual). The segment name is known to the LINKER and, provided that the segment is similarly declared in each task, the data within the segment may be shared. If no segment name is specified, the data is allocated in the BLANK segment. Note that the data which is to be shared must be declared as STATIC EXTERNAL. So, if the character variable A is to be shared by different tasks, it is declared as follows: DCL A CHAR(20) STATIC EXTERNAL WITHIN X (TT(2)); Any segment attributes specified in the WITHIN clause may be overridden at linkage time by the LINKER command MSEGAT. The MSEGAT command is described in the LINKER manual; the WITHIN clause is described in the GPL Reference Manual.

5.2.3 Communication Between Different Languages

5.2.3.1 Data Types And Other Compilers

When a GPL program calls, or is called by, a program written in another language, and data is to be shared between the two programs, the programmer must take into account the data storage conventions used by the different compilers. The programmer must ensure that the data attributes used represent the same data storage. The table below gives a comparison of GPL, COBOL, PASCAL, C and FORTRAN data formats.

Table 5-1. Comparison of Data Formats

GPL	C (a)	COBOL-74	FORTRAN 77 (e)	PASCAL
FIXED BIN (15)	short int	COMP-1	INTEGER *2	enumerators
FIXED BIN (31)	int	COMP-2	INTEGER	INTEGER
FLOAT BIN (21)	float	COMP-9	REAL	-
FLOAT BIN (53)	double	COMP-10	DOUBLE PRECISION	REAL
FLOAT BIN (109)	-	-	QUADRUPLE PRECISION	-
CHAR (1)	char (b)	PIC X	-	CHAR
CHAR (K) (c)	-	PIC X (K)	-	PACKED ARRAY [1..K] OF CHAR
CHAR (*)	-	-	CHARACTER* (*)	-
POINTER	pointer	- (f)	-	-
ENTRY	-	-	SUBROUTINE	PROCEDURE
ENTRY RETURNS	function	-	FUNCTI0N	FUNCTION
BIT (1) BYTE	-	-	-	BOOLEAN
BIT (8) BYTE	-	-	LOGICAL*1 (d)	-
LOGBIN (32) BYTE	unsigned	-	-	-
FIXED DEC (K) [PACKED] (c)	-	PIC 9(K) USAGE COMP-8	-	-
FIXED DEC (K) UNPACKED	-	PIC 9(K)	-	-

- (a) Note that in C, parameters are always passed by value, unless the function is declared with the special symbol '&'. For example, extern int F(&).
- (b) A LINKER warning will be issued.
- (c) K denotes a constant integer.
- (d) Only certain values are allowed.
- (e) A LINKER warning may be issued.
- (f) See the paragraph below on passing pointers from a Cobol program.

For further information, refer to the language User Guides.

5.2.4 Passing Pointers From A Cobol Program

It is possible, when calling a GPL procedure from a COBOL program, to pass a pointer as a parameter of the CALL statement. The pointer is set up in the COBOL program by means of the ADDRESS OF option in the USING phrase of the CALL statement. (There is no equivalent facility in FORTRAN). The following example shows a COBOL program calling a GPL procedure. Two parameters are passed; the first is a pointer, the second is a halfword binary variable.

```
  .
  .
WORKING-STORAGE SECTION.
  .
  .
01   GROUP.
  02 AGE  PIC S999 COMP-1.
  02 NAME PIC X(31).
  02 AMOUNT  PIC $99.99.
  .
  .
77   CODE    PIC S9 COMP-1.
  .
  .
PROCEDURE DIVISION.
  .
  .
   CALL HPROC USING ADDRESS OF GROUP, CODE.
  .
  .
```

The GPL procedure may be as follows:

```
HPROC:PROC(P,D);
DCL  1  G BASED P,
  2  A FIXED BIN(15),
  2  B CHAR(31),
  2  C CHAR(6);
DCL  D  FIXED BIN(15);
DCL  P  PTR;
A = D + 12;
B = "LITERAL";
C = "$12.34";
 .
 .
RETURN;
END HPROC;
```

5.3 PROCEDURE REFERENCES

5.3.1 Procedure References In A Monotask Program

A GPL program consists of one or more external procedures. The source code for each external procedure is held in a unique input enclosure or library member, which is compiled to produce a compile unit. At linkage time, the LINKER utility resolves all references between compile units and builds an executable load module. Each procedure name (from a PROC statement) or entry name (from an ENTRY statement) is known to the LINKER. For example, if the statement:

```
X : PROC;  or  X : ENTRY;
```

is used, then the entry point X is known to LINKER. The code which is entered through X can be called in another procedure by means of a declaration statement and a CALL statement:

```
DCL X ENTRY (...
CALL X (...
```

GPL procedures can call, or be called by, programs written in other languages (e.g., COBOL or FORTRAN). A GPL procedure calls such a program in the manner shown above: the non-GPL program is called in the same way as another GPL procedure would be called. The only difference is that the programmer must take into account the relative data storage conventions used by the two compilers in order to ensure the compatibility of any data items which are passed as parameters in the CALL statement. See Table 5-1 above. Table 5-2 gives a comparison of the call and entry statements for GPL, FORTRAN and COBOL.

Table 5-2. Comparison of CALL and ENTRY Statements

Language	Call	Entry
GPL	CALL EXTPRO(A,B)	EXTPRO: PROC(C,D)
FORTRAN	CALL EXTPRO(A,B)	SUBROUTINE EXTPRO (C,D)
COBOL	CALL EXTPRO USING A,B	PROGRAM-ID. EXTPRO. . .
		PROCEDURE DIVISION USING C,D.(C and D must be defined at level 01 in the linkage section).

In the statements in Table 5-2, the variables A and B are passed to the called procedure in which they are known as C and D. When the called procedure terminates, the values of C and D are passed back to the calling program as A and B.

5.3.2 Procedure References In A Multitask Program

The above method of calling procedures applies only to monotask programs. In a multitask program, code which is to be called by different tasks should be written in a common segment by using the WITHIN clause in the PROC or ENTRY statement. The segment must be created with Table Type 2 (equivalent to SHRLEVEL=2 in the LINKER command MSEGAT). For example:

```
  P : PROC WITHIN (TT(2));
```

specifies that the code in procedure P is to be written in the same segment, which is to have Table Type (share level) 2. A segment name may be specified, but is not necessary for procedure reference purposes, since the code will be called by procedure name (or entry point name), not by segment name.

Any segment attributes specified in the WITHIN clause may be overridden at linkage time by the LINKER command MSEGAT. If the code segment to be referenced by different tasks is created by a program written in a source language that does not have facilities for specifying segment attributes, SHRLEVEL=2 must be specified in the LINKER command MSEGAT. The MSEGAT command is described in the LINKER manual; the WITHIN clause is described in the GPL Reference Manual.

6. Execution and Debugging

6.1 THE JCL STATEMENT STEP

The user requests that a load module be executed by specifying the load module name in a STEP statement. A simple example is given below:

```
$JOB ...
 STEP GPROG1 LM.MYLIB;
ENDSTEP;
$ENDJOB;
```

The STEP statement specifies the load module name and the name of the library file on which it is stored: ENDSTEP indicates the end of the step specifications and requests execution of the named load module.

See the JCL Reference Manual for a complete description of the STEP JCL statement and of the other JCL statements (e.g., ASSIGN and DEFINE) which may be specified between the STEP and ENDSTEP statements.

The remainder of this section is concerned with facilities for debugging GPL programs and with job execution messages.

6.2 THE GCL STATEMENT EXEC_PG

The previous example can be written in GCL as follows:

```
EXEC_PG GPROG1 LM.MYLIB
```

See the IOF Terminal User's Reference Manual for further details.

6.3 DEBUGGING CODE

If the user inserts debugging code in the program, the code should be enclosed in the compile-time statements:

```
%DEBUG  and  %END_DEBUG
```

The code is compiled only if the DEBUGMD parameter is specified in the JCL statement "GPL", or in the GCL command "GPL". Compile-time statements are described in the GPL Reference Manual.

In most cases, however, the programmer will find it simpler and more efficient to debug his programs using the Program Checkout Facility (outlined below) than to write and insert his own debugging code.

6.4 PROGRAM CHECKOUT FACILITY

Program Checkout Facility (PCF)

The Program Checkout Facility (PCF) is a diagnostic facility which can be executed in parallel with a user program. PCF may be used to monitor the user program in the following ways:

- The flow of program control can be traced through a specified point (or points) in the program.
- The values of specified data items can be dumped when control reaches specified points within the program.
- The values of specified data items can be changed when control reaches specified points within the program.
- PCF commands can be made conditional upon the value of specified data items.
- Commands can be applied to selected procedures or blocks.

PCF can be used in Batch mode or in interactive mode. The programmer specifies the type of monitoring to be done by PCF by means of PCF commands. In Batch mode these commands must be stored in a file which is assigned to the job step and which has the internal file name H_DB; in interactive mode the PCF commands are entered at the interactive device. PCF is requested by specifying the DEBUG parameter in the STEP statement of the user program.

PCF is described in detail in the Program Checkout Facility Reference Manual, and is not described any further in this manual. The paragraphs below are concerned only with the following aspects of PCF usage which are specific to GPL:

- The JCL required at compilation time for the subsequent use of PCF in either symbolic or effective addressing mode.
- The use of PCF with primitives and multitask programs.

6.4.1 Symbolic Addressing And Effective Addressing

If the Program Checkout Facility is to be used in symbolic addressing form, the program must be compiled with the DEBUG parameter specified in the JCL statement GPL . This instructs the compiler to produce tables giving the mapping of source program elements (i.e., line numbers, labels and variables) on main memory locations and to store these tables in the compile unit. Specifying the DEBUG parameter does not affect the subsequent linking and execution of the program.

The user refers to source program elements by their symbolic names in PCF statements. For example: DUMP VARA AT LB2; specifies that the value of variable VARA should be displayed when control reaches label LB2. It is recommended that the Program Checkout Facility be used in symbolic addressing form: it is more efficient and simpler for the programmer. PCF can be used on a program which was compiled without the DEBUG parameter, but it can only be used in effective addressing form or semi-symbolic form, that is, source program elements must be addressed by their memory locations. The addresses of data items are given in the Cross Reference Variables Map produced by the compiler when the MAP parameter is specified in the GPL statement.

The address of a source line is constructed from information given in the following two listings:

- the Line Location Map (also produced by specifying the MAP parameter in the GPL statement

- the Group Information listing produced by LINKER.

The use of effective addressing in PCF is described in detail in the Program Checkout Facility User's Guide.

6.4.2 Primitives And Pcf

The expansion code produced by MACPROC is not printed on any listing (though its presence can be inferred from the internal line numbers in the source listing produced by the compiler). Similarly, PCF does not list the occurrences of a source program element within the expansion code. For example, the PCF command:

```
DUMP A AT EACH-REF A;
```

specifies that the value of the variable A is to be displayed at each reference to the name A in the program. If, however, A is passed as a parameter to a primitive there may be one or more references to A within the expansion code. PCF does not list these references.

In the vast majority of cases, the invisibility of the expansion code does not matter for the purposes of debugging: this code is very unlikely to contain errors. The user himself, however, may introduce an error condition into the expansion by passing a bad parameter in the primitive statement. For example, the sequence:

```
10 MYPROC : PROC;
20 DCL  A(1:10)...
.
.
70 I = 20;
80 $H PRIM A(I)...
```

causes an array-out-of-bounds error when the variable A(I) is referenced in the expansion code. In such a case the line number returned by PCF will be of the form:

n.xxxx (for example 80.4)

where:

n	is the external line number of the executable line immediately preceding the expansion code,
xxxx	is the number of the line within the expansion code.

The dot before the expansion line number indicates that the line was inserted by the text editor or generated by the MACPROC utility.

If PCF returns a line number of this form and indicates that an error has been detected at that line, then the user should check the parameter(s) passed in the primitive statement. If the parameters are all correct, the problem should be reported to the Service Center.

It is also possible that, during an interactive PCF session, the user can "break" into program execution while expansion code is being executed; in such a case the message returned will be of the form:

```
... PCF AT LINE n.xxxx IN MYPROC
```

6.4.3 Multitask Programs And Pcf

The current version of PCF does not support multitask programming. The user who wishes to monitor a multitask program should insert his own debugging code.

6.5 JOB EXECUTION MESSAGES

The general format of messages output by the system in the Job Occurrence Report is as follows:

```
ccnn.text
```

where:

cc	is a two letter classification code.
nn	is the number of the message within its class.

The messages are classified according to the nature of the system function which generated the message. Some of the more common classification codes and corresponding system functions are:

CK Checkpoint/Restart
DV Device Management
EX Exception Handling
FP File Open/Close

Depending on the error class, the text following the code may be a brief explanation of the cause of the error or else a further numerical classification followed by a return code specification. A complete list of classification codes, messages and return codes is given in the Error Messages and Return Codes Manual.

6.5.1 Exception Messages Specific To GPL Programs

The exception messages listed and explained below are specific to GPL programs; for other messages, see the Error Messages and Return Codes Manual.

6.5.1.1 Hardware Exceptions

```
EXCEPTION 03-03 ILLEGAL SEMAPHORE TYPE
```

This results from an attempt to perform an operation restricted to a "semaphore with message" or a "semaphore without message", or vice versa. For example, a CALL SEV builtin function was issued from a semaphore declared as "without message".

```
EXCEPTION 03-07 ILLEGAL SEMAPHORE SEGMENT
```

In a builtin function specific to semaphores, the pointer specified as an argument does not define a semaphore.

```
EXCEPTION 04-03 SEMAPHORE COUNT OUT OF RANGE
```

This results from the execution of a V-type operation which would increment the semaphore counter above the maximum value declared for this semaphore.

```
EXCEPTION 06-00 ACCESS OUT OF SEGMENT BOUNDS
```

This usually results from the misuse of a based variable and its associated pointer: the pointer has been updated so that it points outside the limits of the segment.

```
EXCEPTION 06-01 ILLEGAL SEGMENT NUMBER
```

This results from an illegal pointer value: the STE field is greater than the maximum segment number in the segment table.

```
EXCEPTION 06-02 UNAVAILABLE SEGMENT
```

Similar to 06-01, illegal pointer value; but in this case the STE field does not correspond to a valid segment in the segment table.

```
EXCEPTION 06-03 ILLEGAL SEGMENT TABLE NUMBER
```

Similar to 06-01, illegal pointer value; but in this case the STN field does not correspond to a valid segment table number.

```
EXCEPTION 09-08 ILLEGAL FIELD IN INSTRUCTION
```

The code was generated with the option CODE=OBJCD, but it is being run on a class A machine. See paragraph 3.2.1. of this manual.

```
EXCEPTION 09-10 ILLEGAL OVERLAP IN TRANSLATE INSTRUCTION
```

This results from an overlap, in the TRANSLATE builtin function, between the string to be translated and the translation table.

```
EXCEPTION 10-09 ILLEGAL GATE SEGMENT
```

This results from an attempt to call a procedure to be executed with a maximum ring of 2 from a procedure executed in ring 3, when the called procedure is not gated. (No GATE command at linkage time with CMRN=3).

```
EXCEPTION 12-00 READ RIGHT VIOLATION
```

This results from an attempt to read, in a procedure executed in ring 3, a variable located in a segment in which read operations are allowed in ring 2 only.

```
EXCEPTION 12-01 WRITE RIGHT VIOLATION
```

Similar to 12-00, but for write operations.

```
EXCEPTION 17-02 SUBSCRIPT OUT OF ARRAY RANGE
```

This results from a reference to an array element, the subscript of which is outside the bounds of the array. This exception occurs only when the SUBSCRIPTRANGE condition prefix applies.

6.5.1.2 Software Exceptions

Software exceptions correspond to run-time checks generated by the compiler. They have the form:

```
UNEXPECTED RETURN CODE (RC= F000xxxx -> USER 0, yyyyyyyy)
           GOT IN TASK MAIN AT ADDRESS <address>
```

The values are:

xxxx	yyyyyyyy	Meaning
1032	CASUNKN	No matching value in SELECT statement without OTHERWISE clause.
1807	LNERR	Illegal length specified as argument to VERIFY builtin function.
1816	SNDARERR	Illegal syntax specified to BINARY builtin function.
1889	SEQERR	A function is left and no value is returned. (No RETURN (x) statement.)

7. Literals And Variables

The following sections are intended for the user who wishes to minimize the "cost" of GPL constructs in terms of the size and execution time of the generated code. It is assumed that the reader is familiar with the contents of the GPL Reference Manual and has a basic knowledge of the GCOS 7 operating system.

It is also assumed that the user programs will have been properly structured, i.e. organized into a number of modules, each module being a procedure or a BEGIN block.

NOTE: This document uses symbols in the left margin to indicate danger or performance. The symbols are as follows:

Symbol	Meaning
!	Indicates danger
-->	Relates to performance

7.1 LITERAL VALUES

7.1.1 Types

The GPL language allows the programmer to describe 8 kinds of literals belonging to the 5 data types, namely:

```
. true FIXED BINARY such as       11001B
. FIXED BINARY in decimal base    25
. FIXED DECIMAL                   25.
. FLOAT BINARY                   .25E01
. BIT STRING (binary)            "11001"B
. BIT STRING (hexadecimal)       "19"X
. CHARACTER STRING               "AB"
. CHARACTER STRING (hexadecimal) "C108"H
```

7.1.2 Logbin Type

It is not possible to describe a literal of type LOGICAL BINARY, or LOGBIN. FIXED BINARY literals must be used, and these will be converted to LOGBIN if necessary by the context. For example:

```
DCL L LOGBIN (8) INIT (2);
    L = L + 1;
```

The literals 2 and 1 will be converted to LOGBIN. Be aware of this when looking which conversion rule will apply in an expression.

7.1.3 About Syntax

! Some literal descriptions have a similar syntax but different types, for instance

```
 11001B  (FIXED BINARY) vs "11001"B  (BIT STRING)
"A2"H   (CHAR STRING)  vs   "A2"X   (BIT STRING)
 12     (FIXED BINARY) vs    12.    (FIXED DECIMAL)
```

Remember that the "E" is mandatory for FLOAT BINARY literals, as is the final decimal point for FIXED DECIMAL literals.

7.1.4 Precision Of Arithmetic Literals

The precision of a literal is computed from the number of digits or which it consists. (See the GPL Reference Manual)

This is particularly important in the following cases:

- implicit conversions
- conversions through builtin functions (if not specified)
- FLOAT BINARY values handling

For example, 1 is a literal FIXED BINARY with precision p=5 since p= 1 + ceil (3.32 x l) where l=1. Therefore, if B8 is a BIT(8) data item, the statement B8=1; will lead to a conversion to a BIT STRING of length 5 padded to the right with zeroes, so that B8 will contain "00001000"B.

Leading zeroes, although non-significant, will be taken into account when computing the precision. Example:

```
BINARY(12)   will return a FIXED BINARY(8)  value, as l=2
BINARY(0012) will return a FIXED BINARY(15) value, as l=4
```

The internal representation for a FLOAT BINARY literal depends on its precision. As a reminder:

1 to 6 digits in mantissa	:short representation
7 to 15 digits in mantissa	:long representation
16 or more digits in mantissa	:extended representation

Therefore avoid extended FLOAT BINARY literals if you have performance problems because their handling is more expensive.

Example:

Use

```
1.00000000000000E0
```

instead of

```
1.000000000000000E0
```

This is because the second version has 16 digits and will then be implemented as extended.

If a value has an infinite representation in decimal, try to give the maximum number of significant digits according to the representation you choose. For example, the following value is the most accurate representation for one third.

```
0.33333333333333E0
```

7.1.5 Use Of Symbolic Literals

This is an important means to increase program readability.

As a reminder, this is achieved by using the REPLACE compile-time statement:

```
   %REPLACE identifier BY literal;
```

This is particularly interesting in the following cases:

- When it is an implementation value

```
   %REPLACE MAX_NUMBER_OF_USERS BY 17;
   %REPLACE EPSILON BY 1E-10;
```

- When the name is better known than the value

```
   %REPLACE PI BY 3.1415926535E0;
   %REPLACE BELL BY "2F"H;
```

7.1.6 Literals Versus Constants

The symbolic representation of values may also be achieved using the CONSTANT attribute for declarations. The main differences between the two features are that CONSTANTs are always allocated and not pooled.

A memory access is thus needed, when literals may be incorporated in machine instructions and therefore not allocated.

If a literal must be allocated because it cannot be used directly, a new copy is actually allocated only if its internal representation does not already exist in the constant section.

Example:

```
X ="ZA";
Y ="BX";
Z ="AB";
```

In this case, only a string of length 4 will be allocated: "ZABX".

7.2 VARIABLES

7.2.1 Arithmetic Data

FIXED and FLOAT BINARY data are represented internally with a precision that is predefined according to the precision specified by the user.

For FIXED DECIMAL and LOGICAL BINARY data the actual precision is used.
If you have some performance constraints, note that:

- Generally speaking FIXED BINARY is more efficient than LOGBIN which is more efficient than FIXED DECIMAL.
- For LOGBIN data the precision is all important. 8,16 and 32 are the most efficient, followed by 1 and 24, and then the other values.
- For FIXED DECIMAL, performance is directly related to precision. In particular, note that:
- A general register is required to handle the precision when it becomes higher then 16.
- PACKED is more efficient than UNPACKED.
- For FLOAT BINARY data, beware of extended representation, as this is less efficient than the others. This representation takes effect as soon as the precision exceeds 53.

7.2.2 String Data

Manipulation of BIT STRINGs is more efficient if the string does not share a byte with other data.

Example:

```
DCL 1 *,
    2 B1 BIT(1),
    2 B2 BIT(1);
DCL   BB1 BIT(1);
DCL   BB2 BIT(1);
```

Access to BB1 and BB2 will be more efficient than to B1 and B2 as B1 and B2 are both in the same byte. Note however that this form is more compact.

Bit strings of length 1, 8, 16, 32 and 64 are handled better than other lengths.

Character strings of length 1, 2 or 4 are particularly efficient.

Strings of length up to 256 or strings of variable length with the SHORT attribute are more efficient.

7.2.3 Program Control

This involves pointer, entry and label data. Remember that label variables are not allowed.

7.2.3.1 Pointer Handling

The use of specific constructs is preferable when handling pointers.

Pointers allow the user to manipulate memory locations. Remember that the programmer is responsible for the validity of the address value.

Moreover the NULL() builtin function returns a unique pointer value that do not identify any location.

7.2.3.2 Entry Variables

entry variables

The use of entry variables is the only way to communicate a context dynamically between modules.

Example:

We have two modules M1 and M2. According to some event which occurs for M1, M1 gives control to M2 by selecting a specific function of M1 (one of the procedures of M1).

This kind of structure may be achieved through an ENTRY VARIABLE common to both M1 and M2.

Let P1, P2 and P3 be the procedures of M1 which can be used by M2, and let EV be the entry variable.

In M1 one must have EV = P1;, EV = P2; or EV = P3;

A call to EV inside M2 calls P1, P2 or P3 depending on the last assignment to EV.

Consequently different procedures may be called using one name EV which insures independence between M1 and M2.

An internal procedure may be called externally in the way shown below:

```
P : PROC ;
 ...
P1 : PROC ; ... END P1 ;
P2 : PROC RECURSIVE ; ... END P2 ;
P3 : PROC ; ... END P3 ;
DCL EV ENTRY VARIABLE EXT ;
EV = P2 ;
  ...
END P ;
```

EV may be declared and used in another external procedure Q.

P2 will be called in the example. As P2 is an internal procedure of P the context of P is accessible.

Note the example shown below:

```
P : PROC ;
DCL EV ENTRY VARIABLE EXT ;
  ...
EV = P1 ;
LAB : ...
P1 : PROC RECURSIVE; GO TO LAB ; END P1; END P ;
```

If P calls Q which calls R and so on, a call to EV will produce a transfer of control to the last activation of P i.e to the label LAB. Q, R etc. will be erased from the stack.

Note that if such an internal procedure has not got the RECURSIVE attribute, the compiler forces it and a message is issued to this effect.

7.2.4 Structuring Data

This is achieved by building structures or arrays,which correspond to homogeneous and non-homogeneous aggregates respectively. Both structures and arrays can be mixed. In this latter case non-connected arrays can be obtained where two elements are not contiguous in memory.

Example:

```
DCL 1 S(3)
    2 A FIXED BIN(31),
    2 B PTR;
```

The representation of A in memory is as follows:

A(1)	//////	A(2)	//////	A(3)	//////

Some restrictions of use apply to such aggregates:

- They cannot be globally assigned; (T1=T2 is not allowed if T1 or T2 is a non-connected array).

- They cannot be used as arguments for builtin functions.

Because of alignment constraints fillers may occur in certain structures. (Refer to Appendix C of the GPL Reference Manual). Be careful when handling such structures, especially when comparing them.

8. Storage Control

8.1 ADDRESSING OF DATA

The GPL compiler establishes the addressing of data according to storage class.

- B0 points to the communication area of the current stack frame.
- B1 points to the local area of the current stack frame.
- B7 points to the linkage section of the procedure.

The registers need not be reloaded once the prologue of the procedure has been executed.

8.1.1 Constant Data

There are two types of constant; data declared with the CONSTANT data attribute and literals. The constants declared with the CONSTANT attribute are allocated in the linkage section and are referenced directly through B7. They do not share their storage with any other data. Literals that can be manipulated using immediate instructions are not allocated.

Example:

X="A"; generates an efficient instruction.

A literal with a length greater than 8 bytes is allocated in the current code segment and must be referenced by a supplementary instruction. Other literals are allocated in the linkage section and are directly referenced through B7.

When allocated, two literals can share the same memory.

Example:

```
    X = "ZA"
    Y = "BX"
    Z = "AB"
```

Only "ZABX" is allocated.

--> Programs containing INTERNAL STATIC data used as constants may be greatly improved by using the CONSTANT attribute which will avoid the loading of a base register and possibly a segment.

8.1.2 Static Data

Static data, INTERNAL or EXTERNAL, is allocated in segments.

Each name with the EXTERNAL scope attribute has a pointer which points to that data in the linkage section of the procedure.

INTERNAL static data is grouped together by the compiler into one or more segments. Each segment is addressed by a pointer in the linkage section of the procedure. Thus, INTERNAL and EXTERNAL data are addressed in a similar way.

Example:

```
P : PROC ;
 .
 .
 .
DCL A EXTERNAL ....
DCL B INTERNAL STATIC....
 .
 .
 .
```

leads to the following linkage section:

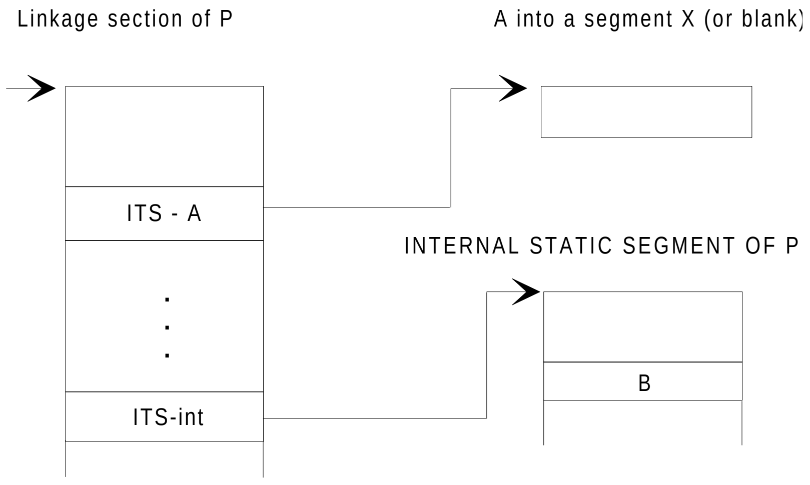


Figure 8-1. Static Data Linkage Section

To address A or B, a base register with the corresponding ITS must be loaded.

8.1.3 Automatic Data

AUTOMATIC variables are allocated in the hardware stack. The stack frame consists of two areas; the fixed area, which is addressed directly, and the variable area, which requires one level of indirect addressing via pointers in the fixed area. Short AUTOMATIC variables (i.e. <= 1024 bytes), are allocated in the fixed area; long AUTOMATIC variables (i.e. > 1024 bytes) and adjustable AUTOMATIC variables are allocated in the variable area.

Note that all scalar POINTER and FIXED BINARY data is short, and is therefore allocated in the fixed area. For this initial allocation, an implicit base register exists which is B1.

A stack frame allocation may be viewed as follows:

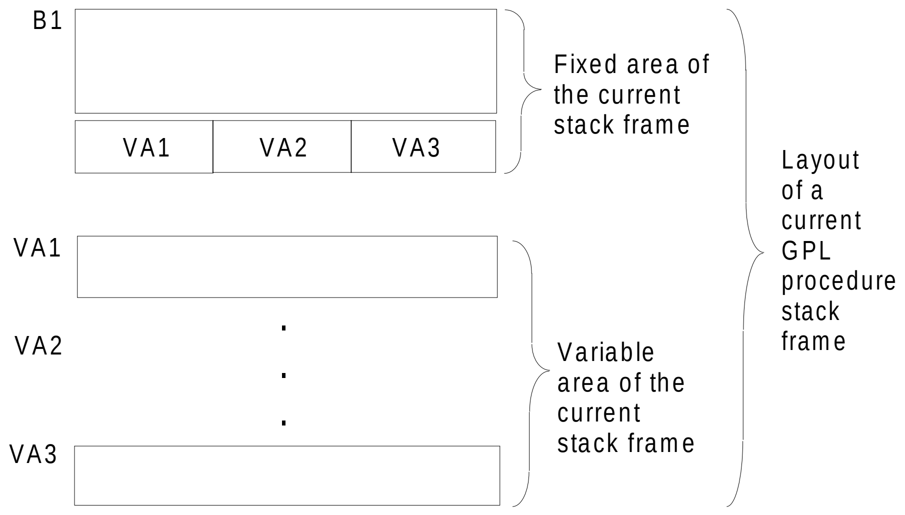

B1
VA1
VA2
VA3
Fixed area of the current stack frame
VA1
VA2
VA3
Variable area of the current stack frame
Layout of a current GPL procedure stack frame

Figure 8-2. Automatic Data - Stack Frame Allocation

Each box of the variable area is allocated on entering a BEGIN or internal procedure block.

The addressing of the variable area is obtained through "DOPE" information allocated in the fixed area and initialized during the prologue of the procedure. If the procedure contains internal procedures with the RECURSIVE attribute, for example:

```
P : PROC;
  .
  .
  .
Q : PROC RECURSIVE
  .
  .
  .
END Q, END P;
```

the stack frame allocated for Q is linked back to the stack frame of P. The layout of the stack during execution may be as shown in the following figure.

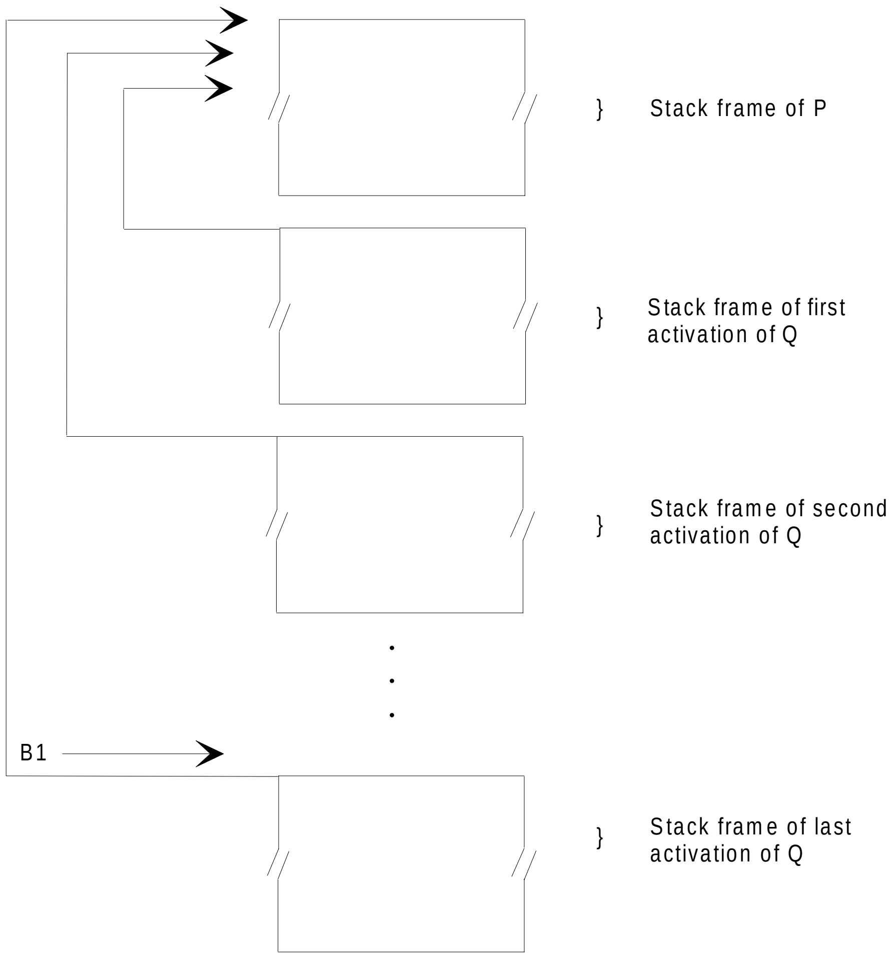

Stack frame of P
Stack frame of first activation of Q
Stack frame of second activation of Q
B1
Stack frame of last activation of Q

Figure 8-3. Automatic Data - Stack Layout

However, automatic variables declared in P and in the scope of Q may be referenced in Q. It is possible to address P during the activation of Q through a link set during the execution of the prologue of Q.

A new stack frame is created each time an internal procedure with the RECURSIVE attribute or an external procedure is entered. In the first case, when an external procedure contains a recursive internal procedure, the new stack frame which is allocated for the internal procedure is linked back to the stack frame of the external procedure. Variables which are declared in the external procedure are within the scope of the internal procedure. They can be referenced in the in the internal procedure. However, referencing variables in a non-current stack frame involves indirect addressing, as follows:

- Variables allocated in the fixed area of the non-current stack frame require one level of indirect addressing.
- Variables allocated in the variable area of the non-current stack frame require two levels of indirect addressing.

8.1.4 Parameter Data

Like AUTOMATIC data, the parameters are allocated in the hardware stack.

Parameters of external procedures or internal procedures with the RECURSIVE attribute are referenced through the BO register.

Parameters of internal non-recursive procedures are referenced through the B1 register.

Parameters of an OPTIONS(VARIABLE) ENTRY that are not named in the parameter list, CHAR(*) parameters and unaligned BIT parameters are referenced through a data descriptor, called a DOPE. To avoid two base register loadings, the DOPEs of CHAR(*) and unaligned BIT parameters are moved from the caller's stack frame to the fixed area of the current stack frame when the procedure block is activated. Unnamed parameters are referenced via the ARG_PTR builtin function.

If a dummy argument is passed (* in the argument list) a NULL pointer is set in the parameter area (parameter without DOPE) or in the first word of the DOPE (parameter with DOPE). For example :

```
P:PROC(U,V,W,X)OPTIONS(VARIABLE);
DCL(U,V)CHAR(*),(W,X)FIXEDBIN(31);
Q:PROC(Y,Z);DCL(Y,Z)FIXED BIN(31);
:
END Q;
:
CALL Q(D,E);
:
END P;
```

Let the call to the external procedure P be:

```
CALL P(*,A,*,B,C);
```

This example has the following stack frame layout.

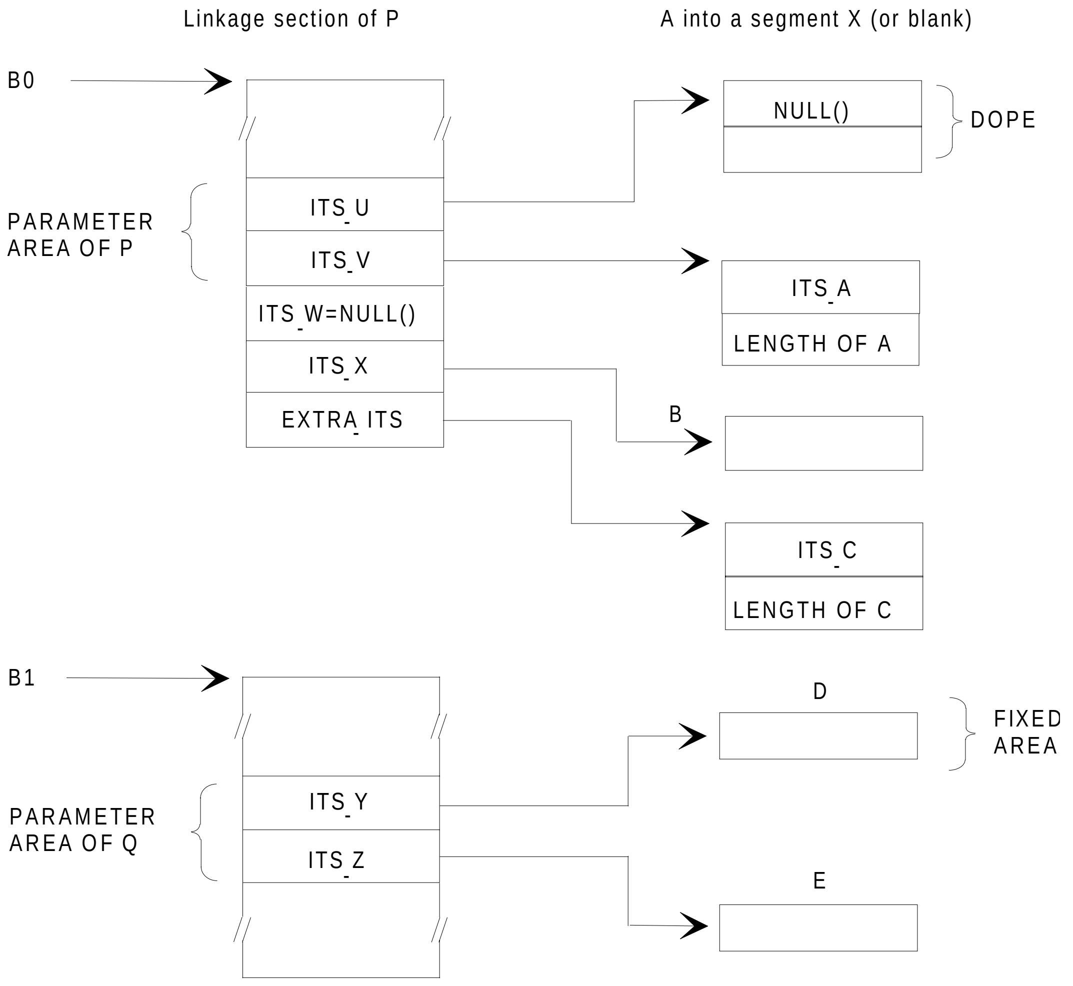


Figure 8-4. Parameter Data - Stack Frame Layout

If the parameters are in a previous stack frame, addressing is established through the link array but this implies one indirection more.

Example:

```
P : PROC (X, Y);
Q : PROC RECURSIVE;
END Q; END P;
```

has the following stack frame layout:

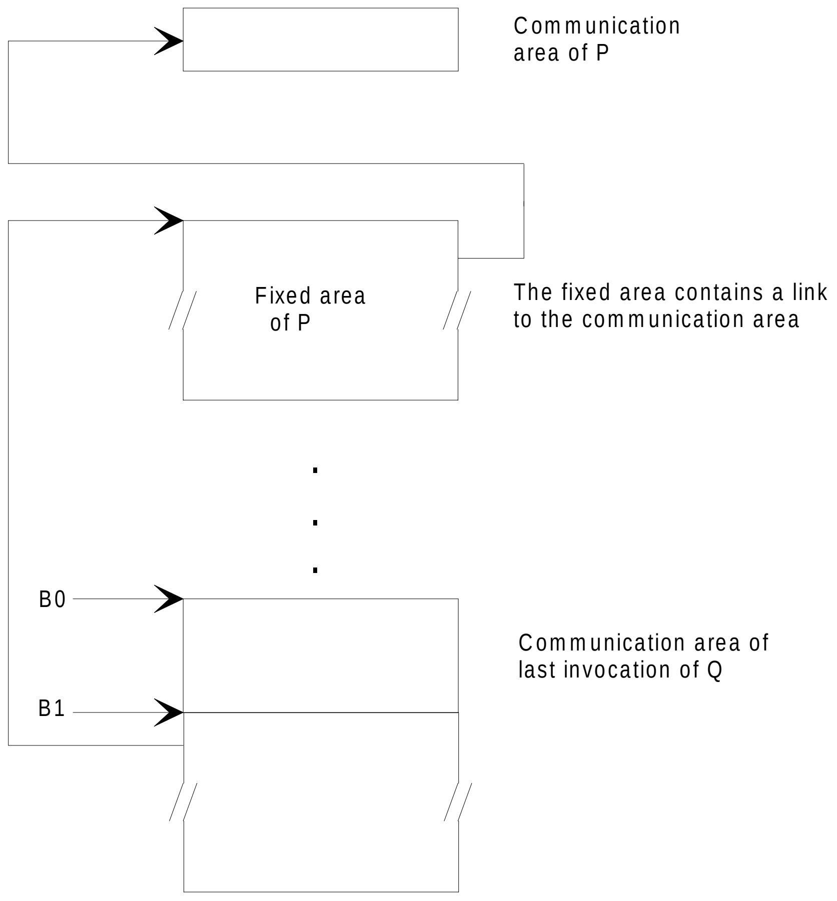

Communication area of P
Fixed area of P
The fixed area contains a link to the communication area
B0
B1
Communication area of last invocation of Q

Figure 8-5. Parameter Data - Stack Frame Layout With Link Array

8.2 SCOPE USAGE

Beware of the difference between scope and allocation: an object can be allocated but not accessible.

Example:

```
P: PROC;
Q: PROC;
DCL X PTR EXT;
END Q;
END P;
```

X is allocated when it is in P, in fact as soon as the program starts, but it is not accessible because its scope is restricted to Q.

Generally speaking, declaring the variables and procedures at the lowest possible level achieves some data encapsulation. Moreover it may increase performance as accessing local automatic data may be cheaper than accessing global automatic data as explained previously.

8.3 STORAGE SHARING

There are two kinds of storage sharing, static (completely known at compile-time) and dynamic.

Example:

```
DCL X FIXED BIN(31) BASED(ADDR(Y));
```

implies dynamic sharing because other locaters can be used for X.

8.3.1 Static Versus Dynamic

Performance is improved by changing dynamic sharing (using the BASED attribute) to static sharing (using the DEFINED or OVERLAY attributes).

If sharing is static the compiler knows which variables share the same storage. It can thus define precisely which variables will be destroyed by an assignment. Therefore code optimization will be more efficient.

Example:

```
   DCL (X,Y) FIXED BIN(31);
   DCL (B1 BASED(P1), B2 BASED(P2)) FIXED BIN(31);
   X = 3;
   B1 = 5;
   Y = X + 1;        /* X will be reloaded from memory */
   B2 = B1-1;        /* B1 will also be reloaded from memory */
```

Use the NOMAPPED attribute if possible and if the BASED attribute cannot be avoided (see below).

Variable information such as lengths and dimensions are recomputed at each reference for BASED objects.

8.3.2 Sharing Table For Based Variables

Refer to the table given in the GPL Reference Manual.

Note especially that program control variables (ENTRY, POINTER), should not share storage with data of other types.

Example:

```
   DCL P PTR;
   DCL B BIT(32) BASED;
P and B should not share storage. If they do, the DEFINED
attribute must be used as follows:
   DCL P PTR;
   DCL B BIT(32) DEFINED (P);
```

9. Declarations

The addressing of data is discussed in Section VIII. More information about declaration attributes is given below.

9.1 ALIGNMENT

Aligning certain types of data on certain boundaries can influence performance. Alignment can be considered in two parts.

9.1.1 The Alignment Of Parameters

BIT STRING and LOGBIN data, and aggregate data consisting entirely of BIT STRINGs and LOGBINs are bit-aligned by default unless the BYTE attribute is specified.

If this is the case, a descriptor (called a DOPE) is added to the CALL statement, and references to the parameter in the called procedure are made through this descriptor. This increases both code length and execution time.

Example:

```
DCL B BIT(32); /* level one, then BYTE*/
CALL P1(B); CALL P2(B);
P1: PROC(X);
DCL X BIT(32);
X=(32)"0"B;
END P1;
P2: PROC(X);
DCL X BIT(32) BYTE;
X=(32)"0"B;
END P2;
```

The results of this example are shown below.

```
                             Ratio not BYTE/BYTE
Code Size (BYTE)                    ~3
Timing (Call)                       ~2
       (Body of Pi)                 ~5
       (Total)                      ~4
```

--> For best results, use the BYTE attribute when calling procedures in other languages. Except for GPL, there is no language implemented on GCOS7 that can send or receive parameters that are not byte aligned.

9.1.2 The Alignment Of Other Data

Aligning data on a natural boundary, for example aligning PTR, FIXED BIN(31) and FLOAT BIN data on a WORD boundary, decreases the time needed for memory access.

The best alignment for AUTOMATIC level-1 data is automatically chosen by the compiler.The programmer has only to worry about:

- the alignment of STATIC data
- the alignment of elements in a structure (for all storage classes).

Example:

```
DCL 1 S WORD,
    2 P PTR,
    2 F FIXED BIN(15),
    2 B BIT(1)
```

is more efficient than:

```
DCL 1 S WORD
    2 B BIT(1),
    2 P PTR,           /*only BYTE aligned*/
    2 F FIXED BIN(15);
```

This is because in the second case P and F have only a BYTE boundary (although S is WORD aligned). Note that the improvement in performance may be up to 25% on a load-store sequence.

9.2 ADJUSTABLE ELEMENTS

Data is called adjustable if its dimension or length is not known at compile time.

Only AUTOMATIC and BASED data may be adjustable.

Constant expressions in dimension or length (such as CHAR(2+2)) lead to an adjustable element.

For automatic data the values are computed when entering the block in which the object is declared. For based variables they are computed at each reference.

For self-defining BASED structures, an implicit pointer is needed and the dimension and length are derived from the object based by this pointer.

Example:

```
DCL 1 S BASED (P),
    2 L FIXED BIN(15),
    2 C CHAR(L);
```

Q->C = P->C; assumes that the length of Q->C is P->L.

Note that both dimension and length may be variable as in:

```
DCL T(I:I+N) CHAR(N) AUTO;
```

9.3 INITIALIZATION OF DATA

Use of expressions may increase readability and maintainability of programs.

Example:

```
DCL   1  MESSAGE_TABLE STATIC,
      2 * FIXED BIN (15)  INIT (DISP(M1)),
      2 * FIXED BIN (15)  INIT (DISP(M2)),
        .
        .
        .
      2 * FIXED BIN (15)  INIT (DISP(M25)),
      2 M1,
      3 * BIT(8)        INIT(FIXED(HASH("ABC",0,97),8)),
      3 * BIT(8)        INIT(FIXED(3,8))
      3 * CHAR(3)       INIT ("ABC")
        .
        .
        .
      2 M25,
      3 * BIT(8)
      3 * BIT(8)
      3 * CHAR(5)       INIT (" TU") ;
```

which looks like a table of names.

The table contains a list of indexes to the names. Each name cell contains a hash code value, the length of the name and the name

itself.
Instead of an index, a pointer may be used and the declaration will be

as follows:

```
   2 * PTR INIT (ADDR(M1)),
```

and so on.
Note that AUTOMATIC data may also be initialized. The code for the initialization is provided by the compiler in the prologue of the block in

which the variable is declared.
For example if you want to initialize all the elements of an array

with an initial value you can write:

```
DCL A (25) FIXED BIN(15)
    INIT ((HBOUND (A,1)) -1) ;
```

The parameterized expression to compute the number of elements will be:

```
HBOUND (A,1) - LBOUND (A,1) + 1
```

Duplication factors may also be expressions.

If you have some performance constraints, you may find the following information helpful.

- For large static objects, static initialization (with the INITIAL attribute) is more efficient than dynamic initialization (with assignment statements).
- For structures, (with any storage class), global dynamic initialization is more efficient than field-by-field dynamic initialization.

Example:

```
DCL 1 S,
    2 D FIXED DEC(12),
    2 R FLOAT BIN(53),
    2 C CHAR(8),
    2 P PTR;
DCL 1 S_INIT CST,         /*use the CST attribute*/
    2 * FIXED DEC(12) INIT(-31.),
    2 * FLOAT BIN(53) INIT(3.0 E0),
    2 * CHAR(8) INIT ("ABCDEFGH"),
    2 * INIT (NULL());
   S = S_INIT;
```

is more efficient than:

```
   S.D = -31.;
```

9.4 USE OF THE NOSUBRG ATTRIBUTE

NOSUBRG cannot be overridden.

Example:

```
   DCL T(10) PTR;
   DCL V(10) NOSUBRG PTR;
   (SUBRG): T(i)=v(i);            /*checking on T only*/
```

This code is very efficient if NOSUBRG is specified when the lower bound is 0 and the length of an element is a power of two. In this case the index value is computed through a shift instruction.

This attribute makes it possible to reference arrays with more than 64K elements, such as an array which maps a large segment.

Example:

```
DCL T(0:1048575) PTR NOSUBRG BASED;
```

9.5 ATTRIBUTES THAT IMPROVE PERFORMANCE

9.5.1 Short

This attribute may be used for all strings whose length is adjustable but less than or equal to 256. When moving 256 bytes for example, the gain will be about 20%.

Example:

```
DCL C1 CHAR(I) BASED(P) SHORT;
DCL C2 CHAR(J) AUTO SHORT;
```

NOTE: The programmer must ensure that the attribute is used consistently. In the example, if I or J is greater than 256 an unpredictable result will occur.

9.5.2 Nomap

This attribute may be used for any data that cannot be modified indirectly.

Example:

```
DCL L FIXED BIN(15) AUTO NOMAP;
DCL BUFFER CHAR(L) BASED(P) SHORT;
DCL P PTR NOMAP;
BUFFER = REPEAT ("00" H, MEASURE (BUFFER)-1));
```

The code generated will be efficient in this case. But if NOMAP is not specified for L and S, a temporary data item of length L will be created in the stack, initialized with "00"H and then moved to BUFFER, because the modification of BUFFER may change either L or P.

NOTE: The programmer must use this attribute consistently.

9.5.3 Input

This attribute indicates that an object will not be modified. It can increase the readability of the program as it restricts the context modified in the procedure.

Note that the compiler checks that the object will actually not be modified.

It may also improve performance in some cases:

As explained below, literal arguments are passed to procedures or functions by making a copy of the object in the stack, and passing this copy to the called procedure (to avoid modification attempts). If the parameter is declared with the INPUT attribute, the compiler knows that no update can be made. It will therefore pass the address of the argument to the constant section instead of making a copy.

```
DCL P1 ENTRY (CHAR(100));
DCL P2 ENTRY (CHAR(100) INPUT);
CALL P1 ((100)" ");
CALL P2 ((100)" ");
```

The first call will need 100 bytes in the stack for the copy and a move instruction on 100 bytes. Both will be avoided in the second case.

NOTE: On the other hand, as an address in the constant section is passed, the parameter must be forced to be the passed value in some contexts:

```
DCL PEXT ENTRY (FIXED BIN(31) INPUT);
CALL PEXT ((20));
```

9.5.4 Reducible

This attribute may apply for any function that does not depend on its context in any way.

Example:

```
NEXT: PROC(P) REDUCIBLE RETURNS(PTR);
DCL P PTR INPUT;
DCL 1 S BASED(P) NOMAP,
    2 DATA
    2 NEXT PTR;
    RETURN (S.NEXT);
END NEXT;
IF NEXT(Q) = NULL()
THEN Q = NEXT(Q);         /*NEXT will be called only once*/
```

NOTE: The programmer is responsible for using this attribute consistently.

In the following cases, REDUCIBLE may not be used:

- A variable is modified that is local to the function and is not AUTOMATIC.
- A variable is referenced that is local to the function and is not AUTOMATIC.
- Data from outside is used (GET, READ, GETOD(),...).

9.5.5 Constant

As explained in section 8.1, access to CONSTANT data is more efficient than access to STATIC data.

9.5.6 Byte

The effects of this attribute on parameters are discussed in the subsection about using the GPL compiler.

10. References

10.1 RESOLVING REFERENCES

Name hiding can occur if the same name is declared in several blocks which overlap.

Example:

```
P: PROC;
DCL A FIXED BIN(31);
Q: PROC;
DCL A PTR;
END Q;
END P;
```

NOTE: The name A, which refers to a FIXED BINARY object, cannot be accessed in Q.

! The same name can also occur in several structures in the same block.

The reference is resolved by searching in the current block firstly for a name which is wholly applicable, and secondly for a name which is partly applicable. If no name is found, this algorithm is repeated for all the containing blocks. This process is described in more detail in Section V of the GPL Reference Manual.

Example:

```
P: PROC;
DCL A FIXED BIN(31);
DCL 1 T,
    2 A CHAR(1);
Q: PROC;
DCL 1 S,
    2 A PTR;
A = NULL();
END Q;
END P;
```

The statement A = NULL(); is legal. Referring to "S.A" as "A" is an incomplete qualification as "A" is applicable to "A FIXED BIN(31)". But as explained above, the name A FIXED BIN(31) is hidden by the name A PTR in the structure S, although the reference is only partly applicable.

Consequently "A FIXED BIN(31)" can no longer be accessed in Q. A can be accessed from T through a complete reference T.A, which is not applicable to any data from Q.

10.2 SHORT-CUT IN THE ADDRESSING PATH

Some objects have long addressing paths either because of their storage class (see Section III) or because of the logic of the program (several levels of BASED data).

If such objects are frequently used, especially when referenced in loops, a good way to improve performance is to create a short-cut in the addressing. This is done in the following way:

Determine the longest sub-path in the program that remains constant. Then declare an automatic pointer and assign it with the address equal to the short-cut.

Example:

```
 1 Q: PROC(P);
 2 DCL P PTR;
 3 DCL 1 S1 BASED (P) NOMAP,
 4     2 P1 PTR,
 5     2 A FIXED BIN(31);
 6 DCL 1 S2 BASED (P1) NOMAP,
 7     2 B FIXED BIN(31),
 8     2 P2 PTR,
 9     2 C CHAR(B) SHORT;
10 DCL V FIXED BIN(31) BASED(P2) NOMAP;
11 DCL I FIXED BIN(31);
12 DO I = 1 TO B;
13 IF SUBSTR (C,I,1) =" "
14 THEN V = V +1;
15 END;
16 END Q;
```

This program can be improved by adding the declarations:

```
DCL R0 PTR AUTO INIT(P1);
DCL R1 PTR AUTO INIT(P2);
```

after line 10 and replacing the loop by:

```
D0 I = 1 TO R0 ->B;
IF SUBSTR (R0->B,I,1) =" "
THEN R1->V = R1->V + 1;
END;
```

The code generated is longer than of the initial program, but the execution speed is increased. Assuming that 50% of characters in C are blanks, the behavior of the program is given below.

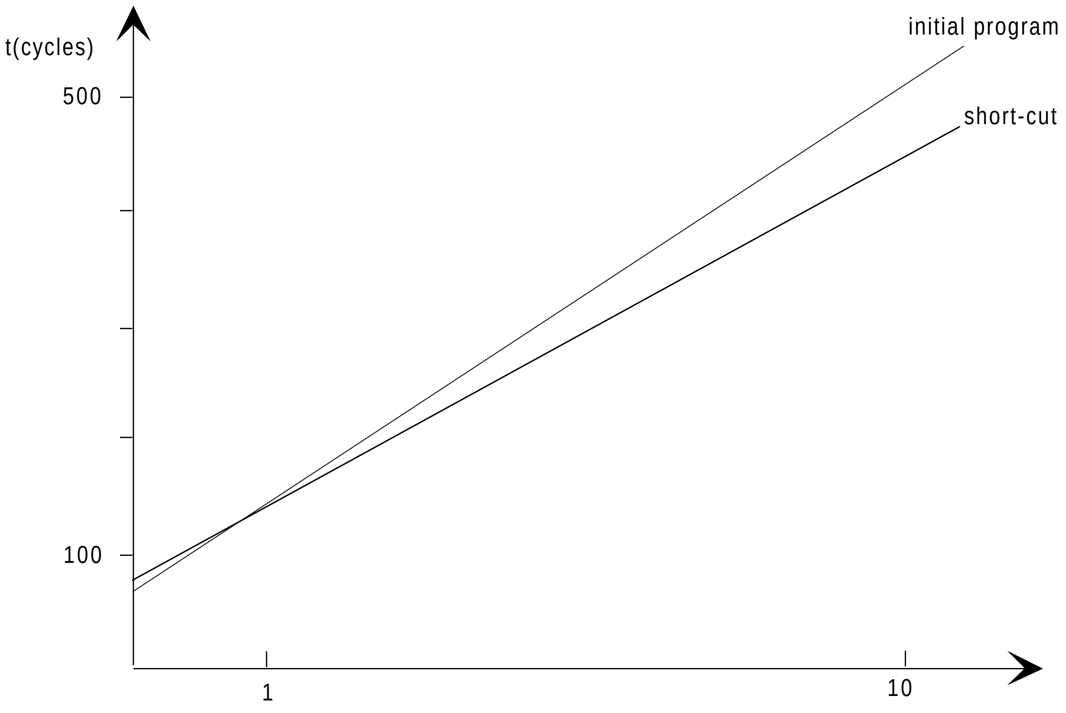


Figure 10-1. Program Behavior With Short-cuT

10.3 PROCEDURES AND FUNCTIONS

The programmer declares when INTERNAL procedures and functions are RECURSIVE. Procedures and functions that have EXTERNAL scope or that are assigned to variable entries are automatically RECURSIVE.

A procedure with a single scalar output parameter may be changed to a function (with the RETURNS attribute) to improve readability and in some cases, performance. This may be done if the procedure can be declared with the REDUCIBLE attribute (see Section VII) and/or if it returns a value with a data type which is not CHARACTER STRING and FIXED DECIMAL. In this latter case the value is returned in a register or a register pair.

It may also avoid the problem of an argument passed by value which cannot contain a returned value (see below).

10.4 PARAMETERS AND ARGUMENTS

Matching between parameter and arguments is done by the CALL statement. According to the argument, it is passed either by reference or by value. In the first case it can be modified as the address of the variable is passed to the procedure. In the second case, a temporary data structure is built which contains the value to be passed. The data passed by the caller cannot therefore be updated.

In addition a descriptor called a DOPE is added to the CALL statement in certain cases. This contains information about the argument, which can be passed by reference or value.

10.4.1 The Argument Is Passed By Value

This is the case if at least one of the following conditions is fulfilled:

- The argument is a literal (see the paragraph 4.5.3., INPUT).
- The argument is an expression.
- The argument is a variable reference enclosed between parentheses.
- An implicit conversion is needed.
- The precision or length of the argument is not equal to that of the parameter.
- The argument is a pseudo-variable reference. In the example below, all the arguments are passed by value.

```
CALL P1 (3, " ");
CALL P2 (A!!B, 3 + I, F(X), ADDR(Z));
CALL P4 ((I));
DCL P5 ENTRY (LOGBIN(32)BYTE), I FIXED BIN(31);
CALL P5(I);
DCL P6 ENTRY (FIXED BIN(15)), J FIXED BIN(31);
CALL P6(J);
CALL P7(UNSPEC(I));
```

Note that an argument that is passed by value cannot contain an output value. The example below illustrates this.

```
DCL CM CHAR(8) INIT(" "),
CALL P(CH);   /* CH contains blanks */
P: PROC(C);
DCL C CHAR(6);
C = (6) "X";         /* The argument is modified
                        if it is declared with the
                        CHAR(6) attribute */
END P;
```

10.4.2 Descriptors

A descriptor is necessary in the following cases:

- If the parameter is a BIT STRING, or a LOGBIN structure or a structure which contains only BIT or LOGBIN data, and which does not have the BYTE attribute. In this case, the descriptor contains the byte address and the bit displacement.
- The parameter is declared with the CHAR(*) attribute. In this case, the descriptor contains the address and the length of the string.
- The entry is declared with the OPTIONS(VARIABLE) attribute (see below).

10.4.3 Variable Number Of Arguments

The only case for which the number of arguments may be not equal to the number of parameters, is when the entry is declared with the OPTIONS (VARIABLE) attribute.

NOTES:

1. This feature is allowed for EXTERNAL procedures only.
2. The ARG_COUNT builtin function supplies the number of arguments. The declared parameters are accessed through their names, the others through the ARG_PTR builtin function. For non-declared parameters a descriptor is built which contains the address and the length of the argument.

Example:

```
DCL P ENTRY (PTR, CHAR(4) INPUT) OPTIONS(VARIABLE);
CALL P (ADDR(X), "ACCD", 21, "ASCDEFGM");
```

In P the following statements can be found:

```
P: PROC (Q,C) OPTIONS(VARIABLE);
DCL Q PTR;    DCL C CHAR(4) INPUT;
DCL V3 FIXED BIN(31) BASED(P3) NOMAP;
DCL V4 CHAR(L) BASED(P4) NOMAP;
DCL L FIXED BIN(31);
DCL (P3,P4) PTR;
SELECT (ARG_COUNT());
  WHEN (0,1,2) CALL ERROR ("NOT ENOUGH ARGS");
  WHEN (3) CALL ARG_PTR(3,P3);
  WHEN (4) DO;
 CALL ARG_PTR(3,P3);
 CALL ARG_PTR(ARG_COUNT(), P4,L);
END;
OTHER CALL ERROR ("TOO MANY ARGS");
END;
            ....
```

Note that if no argument with rank n is passed, the following builtin function:

```
CALL ARG_PTR (n, P, L);
```

returns NULL() in P and -1 in L.

10.4.4 Empty Arguments

If an argument is known to be meaningless in some contexts, it is possible to pass a so-called empty argument as follows:

```
   CALL P (A,*,B);
```

The second parameter is empty.

In the called procedure, it is possible to know if an empty argument has been passed by testing the address of the parameter to see if it is NULL().

Example:

```
   P: PROC (P1,P2,P3);
   DCL (P1,P2,P3) CHAR(10);
   IF ADDR(P2) = NULL()
   THEN CALL MESSAGE ("P2 WAS EMPTY");
            ...
```

Note that this may be a way to allow an internal procedure to have a variable number of arguments.

The CALL statements can be:

```
                           CALL P(A,*,*);
                           CALL P(A,B,*);
                           CALL P(A,B,C);
```

and the number of argument actually passed can be counted in P by testing the address of all parameters to see if they are NULL().

Another way to implement such a feature is the use of secondary entry points.

```
    MAIN: PROC(P1,P2);
           ....
    E   : PROC(P1);
           ....
```

11. Expressions

11.1 GENERAL REMARKS

Expressions are evaluated in the order which corresponds to the precedence of the operators, after any implicit conversions of the operands have been performed if necessary. The order of evaluation is given in the GPL Reference Manual (Section VI). It may be altered by using brackets Constant expressions or sub-expressions are evaluated at compile time. Common sub-expressions may be evaluated only once if the compiler knows that none of the operands is modified.

11.2 USING BRACKETS

! The use of brackets is advised to enhance readability, particularly in expressions involving comparison operators.

Remember that logical operators (except monodic "not") have lower precedence than comparison operators.

For example, the expression

```
IF A=B ! C & D + E
```

may be easier to read if it is written:

```
IF (A = B) ! (C & (D + E)
```

In this second example, to test if A is equal to the "and" between B and C, the statement must be written as follows:

```
IF A = (B & C) because A = B & C means (A = B) & C.
```

11.3 PRECISION AND LENGTH

The precision or length of an expression is managed by the compiler according to the following rules:

- Comparison operators always return a BIT(1) result.
- LOGBIN results have the precision 32.
- FLOAT BINARY results have a precision equal to MAX (p1, p2).
- FIXED results (BIN or DECIMAL) have a precision equal to 1 + MAX (p1, p2). This is limited to 31 for + and - dyadic operators, and p1 + p2 is limited to 31 for + and /.
- -String results have a length equal to MAX (l1,l2).

If the precision of an expression has an important influence on the result (for example, the builtin function CHAR), it is strongly recommended that the precision be managed using the appropriate builtin function.

Example:

```
CHAR(exp)
```

may be written as:

```
CHAR(FIXED (exp,n))
```

Note that from these rules, it follows that if an extended FLOAT BINARY value occurs in an expression, the result is also an extended FLOAT BINARY value. This may decrease performance.

! For string operators the shortest string is extended to the right with "0"B if it is a BIT string, or with " " if it is a CHARACTER string, so that the two operands are the same length. This may lead to unpredictable results in certain cases, as in the following example:

```
IF (X = Y) & B8 THEN CALL P;
```

In this case, B8 as BIT(8) X=Y is evaluated first returning a BIT(1) value (see above), that is extended to the right with 7 zeros. The AND with B8 are then performed. The statement is therefore equivalent to:

```
IF (X=Y) &  SUBSTR (B8,1,1) THEN CALL P;
```

In other words the 7 rightmost bits of B8 will have no influence on the condition.

11.4 CONVERSIONS

The conversions BIT to FIXED or LOGICAL BINARY, LOGICAL to FIXED BINARY and vice versa are implicit in GPL.

When converting from BIT to FIXED BINARY, the BIT STRING is considered as a positive number with a precision equal to the length of the BIT STRING. If the length of the BIT STRING is greater than 31, any bits to the left of the 31 rightmost bits are ignored.

For example, if you have:

```
DCL (I, L) BIT(8), Z CHAR(50), T(10) PTR ;
```

the following expressions are correct:

```
SUBSTR (Z, I, L)
T (I)
```

The code generated is not altered by the conversion. However, an observation message is output by the compiler.

When converting from BIT to LOGICAL BINARY, the BIT STRING of length N is interpreted as a LOGICAL BINARY data item with precision 32. Any bits to the left of the 32 rightmost bits are ignored. If N is less than 32 the 32-N leftmost bits of the LOGICAL BINARY data item are set to zero.

When converting from FIXED BINARY to LOGICAL BINARY, the 32 bits that represent the signed number are considered as a LOGICAL BINARY data item of precision 32.

Example:

```
DCL X LOGICAL BINARY(8);
X = -1;
```

The signed number -1 is considered as 32 bits all set to 1. The eight right-most bits are stored in X. Thus the value of the variable X is 255.

! Be careful when converting from FIXED BINARY to BIT as the absolute value of the number is converted. Writing X = 1; when X is a BIT(8) variable, generates code that may seem strange. The reason for this is that 1 is a decimal representation of a FIXED BINARY variable with precision 1 (one decimal digit). A conversion to the binary base is therefore necessary first. The precision of the result is given by the formula:

```
1 + CEIL (P* 3.32),
```

In this case, the precision is 5. Then a conversion from FIXED BINARY(5) to BIT(8) is performed which leads to the result "00001000"B.

To be sure of the result, write:

```
X = FIXED(1, 8); or X = 00000001B;
```

When converting from LOGICAL BINARY to FIXED BINARY, the unsigned number that is the value of the LOGICAL BINARY data item of precision P is considered as a signed positive number with a precision equal to MIN(P,31).

If the precision of the LOGICAL BINARY is 32 the leftmost bit is ignored.

Example:

```
DCL X FIXED BINARY(31);
DCL Y LOGBIN(32);
 X=-1;
 Y=X;               /* Y = 2**32-1 */
 X=Y;               /* X = 2**31-1 */
```

! For performing conversions of the types described above, you are strongly recommended to use the builtin functions FIXED, LOGBIN and BIT. These functions are described in Section VIII of the GPL Reference Manual.

For the conversion FIXED BINARY to CHAR, GPL provides two builtin functions, BINARY and CHAR.

These are also described in Section VIII of the GPL Reference Manual.

Remember that no implicit conversion from or to FIXED DECIMAL or FLOAT BINARY is provided.

11.5 CONDITIONAL EXPRESSIONS

expressionconditional
! Note that FIXED BINARY and LOGBIN data is not boolean data but arithmetic data, although such data is allowed to be included in condition expressions.

Therefore the following statements are allowed:

```
IF F THEN ...
or
IF L THEN ...
```

where F is FIXED BINARY and L is LOGBIN. However, they produce inefficient code, as a conversion to BIT is needed.

If L is a LOGBIN(1), a good way to test it is to write:

```
IF L = 0 THEN ...
or
IF L = 1 THEN ...
```

But if B is a BIT(1) it is more readable to write:

```
IF  B THEN ...
or
IF ^B THEN ...
```

If the condition is multiple, some parts of the expression may not be evaluated in some cases.

In the following example:

```
IF (P^=NULL()) & (P->X = 0) THEN ...
```

! The use of such a construction may be dangerous because the short-cut in resolving the expression is bound to the context. To avoid such a dependence on the evaluation, expand the statement as follows:

```
IF P^=NULL() THEN IF P->X = 0 THEN ...
```

11.6 REAL COMPARISONS

! Because of rounding or truncation errors that may occur in floating point computations, the programmer is strongly advised against direct comparison. It is better to use a "tolerance range".

Example:

```
   DO UNTIL ( X=X0);
   ...
   END;
```

can lead to an infinite loop because X may never exactly reach the value X0. The following example is much safer:

```
DO UNTIL (ABS(X-XO)<=EPS);        /* EPS is a %REPLACE
                                     e.g.1E-5 */
   ...
   END;
```

12. Statements

The GPL statements are described in Section IX of the GPL Reference Manual. However, further information is given in this section on how the use of certain statements, such as DO, SELECT, LEAVE and IF can aid structured programming.

12.1 ASSIGNMENT

When making an equivalence between a variable reference and an expression, such as in the statement:

```
V = exp;
```

where V is a variable reference and "exp" is an expression, an implicit conversion may be needed according to the data types of V and exp. See Section 11 for further details.

The structure assignment is allowed if the two structures have the same shape, meaning that they have the same hierarchy and their elements are of the same type. If so, the assignment is performed through a single MOVE instruction.

Arrays may also be assigned if they are connected.

All the elements of an array may be set to the same value with a statement of the form:

```
   T = exp;
```

This means that T(i) is assigned with "exp", for i in the range LBOUND (T,1) to HBOUND (T,1). T need not be connected.

12.2 BEGIN

The variables declared in a BEGIN block are not visible outside the block. Hence a BEGIN block may be used to restrict the scope of certain objects, especially in the cases explained below.

As BEGIN blocks may be viewed as procedures generated "on-line", the data declared in the block is that which is necessary to perform the action corresponding to the procedure. Note that objects declared in a DO group can be viewed from the outside.

The use of a BEGIN block can restrict the scope of large or variable automatic data items allocated in the variable area, see Section VIII. This allows you to allocate such resources at the very point you need them.

Example:

```
BEGIN;
DCL WORK CHAR(I);
    ...
END;
```

Like DO and SELECT groups, a BEGIN block may be used for packaging purposes via the WITHIN attribute.

12.3 DO/LEAVE

This is a very powerful statement for repetitive processing. Some brief examples are given here.

- To get an element from a simple linked list, use the following:

```
DCL 1 CELL BASED,
    2 DATA CHAR(12),       /* for instance...*/
    2 NEXT PTR;
DO  P = HEAD_PTR REPEAT (P->CELL.NEXT)
                 WHILE  (P^=NULL());
        ...
END;
```

- The way to get the next element is more elaborate. A function may be used:

```
DO P = HEAD_PTR REPEAT (GET_NEXT(P))
                WHILE  (P^=NULL());
     ...
END;
GET_NEXT: PROC (P) RETURNS (PTR) REDUCIBLE;
DCL P PTR INPUT;
     ...
END GET_NEXT;
```

The WHILE option enables you to exit from the beginning of the loop.

The UNTIL option enables you to exit from the end of the loop.

The DO FOREVER option with the LEAVE statement enable you to exit from the middle of the loop.

UNTIL loops are executed at least once. WHILE and TO loops may never be executed at all.

! Be careful about the exit condition, particularly in the case of TO loops and overflows. As explained in the GPL Reference Manual (from which the notation is taken), the exit condition may be expressed by the following if the BY value is negative:

```
   IF ( v > e2) THEN LEAVE;
if the BY value is positive or null, or
   IF ( v < e2) THEN LEAVE;
```

However, the statement:

```
v = v + e3;
```

may cause an overflow and the following conditions will not be met:

```
v > e2 (or v < e2)
```

Example:

```
   DO LGB8 = 1 TO 255 ... /*LGB8 is a LOGBIN(8)*/
   DO FB15 = 1 TO 32767 ... /*FB15 is a FIXED BIN(15)*/
```

The loops above will never end. To ensure termination, the best way in this case is to use the following:

```
DO LGB8=1 REPEAT (LB8 + 1) UNTIL (LGB8 = 255);
```

!   UNTIL or WHILE loops on real values must be used cautiously because of truncation errors. See the subsection 11.5, "CONDITIONAL EXPRESSIONS".

A LEAVE <label>; statement helps to make the code and the group nesting level less dependent on each other. For example, if the following piece of code:

```
    DO FOREVER;
       ...
    LEAVE;
       ...
    END;
```

is changed to:

```
    DO FOREVER;
       ...
    DO I = 1 to 10;
       ...
    LEAVE;
    END;
       ...
    END;
```

It is not possible to leave the DO FOREVER loop.

12.4 PROCEDURE AND ENTRY

A list of aliases to an entry point can be specified by:

```
ALIAS1: ALIAS2: MAIN: PROC;
 ...
END;
```

ALIAS1 and ALIAS2 are entry names that are aliases of the procedure MAIN. To label an empty statement located just before a PROCEDURE or ENTRY statement, a null statement is necessary.

Example:

```
L:;
P: PROC;
   GOTO L;
END P;
```

If the semicolon after L: is omitted, the statement GOTO L will be illegal since L will denote a procedure.

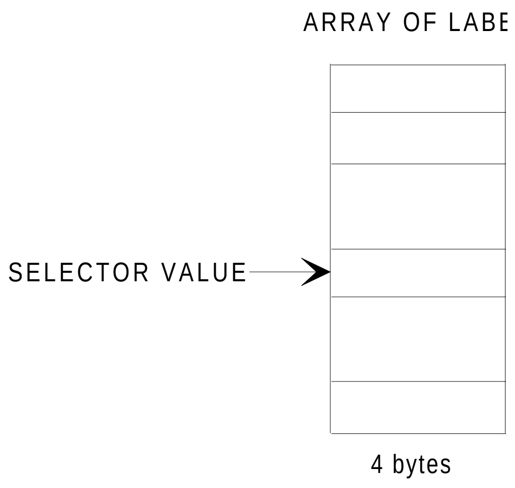
Secondary entry points can be defined with the ENTRY statement. The parameter list must be a sub-list of the parameter list given in the PROCEDURE statement. If a procedure is activated through a secondary entry point and if a parameter which does not belong to this entry point is referenced, the result is unpredictable.

12.5 SELECT

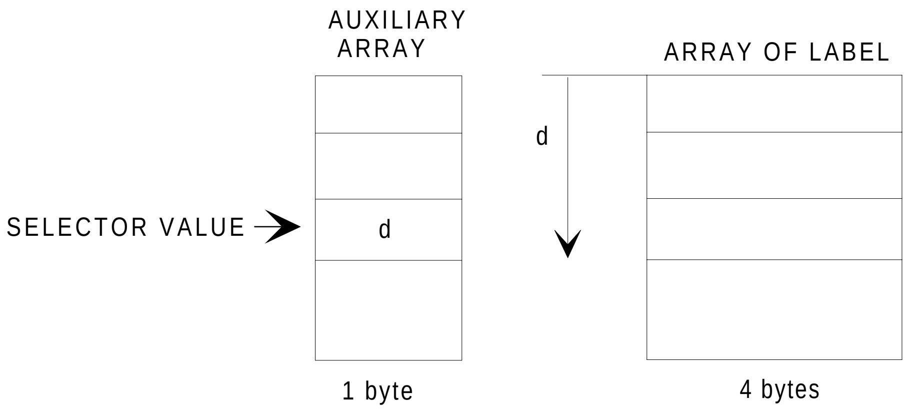
There are 4 cases of generation for a SELECT statement:

1. In the first case, the SELECT statement is expanded to an IF statement. If it exists, the expression in the SELECT clause is evaluated only once.

2. In the second case, the compiler generates an array of labels in order to select the right case.



3. In the third case, an auxiliary array is constructed by the compiler in order to save space.



4. The fourth case is identical to Case 3 except that the auxiliary array item is two bytes in length.

The following definitions help explain the generation algorithm.

- Let S be a boolean the meaning of which is: "There is an expression in the SELECT clause".
- Let E be a boolean the meaning of which is: "There is an OTHERWISE clause in the SELECT statement".
- Let C be a boolean the meaning of which is: "The WHEN clauses contain only literal integers".
- Let W be the number of WHEN clauses.
- Let N be the total number of expressions inside the WHEN clauses.
- Let MAX be the greatest value of the expressions inside the WHEN clauses.
- Let MIN be the smallest value of the expressions inside the WHEN clauses.
- Let R be the range, i.e. R = MAX-MIN +1
- Let X be the number of bytes for an auxiliary array entry.

Note that MAX, MIN and R are significant only if C is true.

The algorithm is as follows:

```
IF (N > 6 & C & S)
THEN DO;
     IF N + 1 < 64
     THEN X = 1;
     ELSE X = 2;
     IF (E|(R * X) + 4 *(W + 1) < 4 * R)
     THEN IF X = 1
          THEN Case 3
          ELSE Case 4
     ELSE Case 2
     END;
     ELSE Case 1
```

The cases require the following storage:

Case 2:	`4 * (R + 1)`
Case 3:	`R + 4 * (W + 1)`
Case 4:	`2 * R + 4 * (W + 1)`

For Case 1, the code may be optimized by first specifying the WHEN cases that will occur most frequently, because all the values are tested in the order given in the source text

For example, to test the return code from a primitive, you can put the case "DONE" first:

```
SELECT;
  WHEN ($H_TESTRC DONE;);
  WHEN ($H_TESTRC SFNUNKN;) DO;
     ...
  END;
  WHEN ($H_TESTRC EFNUNKN;) DO;
     ...
  END;
  OTHER DO;
     ...
  END;
  END;
```

Prefixing the SELECT group with NOSUBRG avoids using a "compute subscript" instruction, which is expensive, to obtain the branch table entry if Cases 2, 3 or 4 are used.

12.6 PACKAGING OF GPL PROGRAMS

This is the way in which pieces of data and code are put together. Note that ring 2 is not allowed with GCOS7-V3B. It will be changed to ring 3 at compilation or linkage time if CODE=OBJCD is specified.

12.6.1 Data

Only data with the STATIC attribute may be pooled together using the WITHIN attribute and given specific attributes.

In order to be independent of the LINKER flow, EXTERNAL data must always be allocated in the same segment. (Note that the default is the blank segment).

12.6.2 Code

If the main procedure has no WITHIN attribute, the code and the linkage section are joined together. To give specific attributes, specify a WITHIN attribute as follows:

```
P : PROC WITHIN (RN(3, 3, 3, 1, 1)) ; ...
```

Now the segment will have the specified attributes.

13. Builtin Functions

13.1 PARAMETERIZATION

The independence of code and data may be achieved using certain builtin functions, namely:

- HBOUND and LBOUND to obtain the bounds of arrays.
- MEASURE and LENGTH to obtain the length of an object. (LENGTH applies only to strings).
- DISP and BITDISP to obtain the displacement of an object in a structure with respect to the level-1 item.

Example:

```
DCL T(-3:5) PTR;
DO I = LBOUND (T,1) TO HBOUND (T,1);
    ...
END;
```

There is no change if the values of the bounds are modified.

13.2 HANDLING VARIABLE LENGTH STRINGS

As explained in Section IX, the use of the NOMAP and SHORT attributes is strongly advised to improve performance. Generally speaking, the use of builtin functions on variable length strings is as efficient as the use of basic equivalent statements.

However, care must be taken to prevent the program from handling zero length strings.

13.3 POINTER HANDLING

Pointers must be handled with their specific builtin functions, namely:

- POINTER (p,x) to obtain the xth byte in the segment to which p points.
- ADDREL (p,x) to obtain the xth byte in the segment after the address to which p points.
- ALLOC (p,s) to obtain the next address in the segment after the one pointed to by p, at which the structure s can be allocated.
- REL (p) to obtain the displacement in the segment corresponding to the address pointed to by p.

13.4 CONVERSIONS

As explained previously, the use of specific builtin functions is recommended to perform conversions, although some conversions are implicit in the language.

Remember that no implicit conversion from or to FIXED DECIMAL or FLOAT BINARY data is allowed.

Note that conversion builtin functions change the data type but not the value, whereas the UNSPEC builtin function changes the value but preserves the internal representation.

13.5 MOVING STRINGS

Because an overlap can occur when moving strings, it is a good idea to use the builtin function MOVERTL, especially when shifting a string to the right.

Example:

```
DCL S CHAR(100);
    SUBSTR (S,1,1+MEASURE(S)-I)=SUBSTR (S,I);
```

The string SUBSTR is shifted I places to the left.

```
CALL MOVERTL (SUBSTR (S,I),SUBSTR (S,1,1+MEASURE(S)-I));
```

The string SUBSTR is shifted I places to the right.

14. Optimizing with GPL

14.1 INTRODUCTION

14.1.1 The Goals of the Optimizer

The GPL language, like other high-level programming languages (for example C, PASCAL, and FORTRAN 77) allows programmers to compose algorithms using concepts that are more abstract than those of the assembly language, thus improving productivity and maintenance. Because of this, the program code generated by a high-level language can be less effective than code written in assembly language. In effect, a high level language does not allow the programmer to improve object code by composing algorithms that are at the level of the machine.

The example below shows how an indexed table address, compiled at the assembler level, develops some expressions that a programmer can not.

```
DCL (A (0:100), B (0:100))      FIXED BIN (31);
DCL (I, J)                  FIXED BIN (15);
   DO I = 0 TO 100;
      DO J = 1 TO 100;
         A (I + J) = B (J + I);
      END;
   END;
```

The compiler evaluates addresses that translate the assignment statement of the innermost loop. Those addresses are as follows:

```
ADDRESS [A (I+J)] = ADDRESS [A] + 4 * (I + J)
ADDRESS [B (J+I)] = ADDRESS [B] + 4 * (I + J)
```

The programmer can not avoid the redundant expressions that the compiler creates, and these redundancies can be extremely taxing on the efficiency of the loop.

The main goal of the optimizer is not to compensate for the eventual weakness of a program. Rather, it is to reduce the inefficiencies of the generated code that are inherent in high-level programming languages.

14.1.2 The Local Optimizer

Before the 80.0 version, the GPL compiler had only two optimization levels, the statement optimization level and the extended linear sequence optimization level. The statement level is automatically activated when using DEBUG option.

In the first level, the scope of the optimization is limited to the algorithm expressions within a source instruction. In the second, the scope of the optimization is extended to a set of instructions, called a linear sequence of a basic block, situated between two label definitions: a label being explicit in the source text, or a label being implicit and generated by the compiler (for example, a conditional instruction, or a loop).

The two optimization levels perform the following principal functions:

- Constant folding.
- Copy propagation (or assign folding).
- Deletion of local redundant expressions.
- Deletion of useless code.

14.1.3 The Global Optimizer

The global optimizer, available in GPL 80.0, extends the optimization reach for a whole procedure. The global optimizer improves local optimization in the following areas:

1. Constant folding and copy propagation.
2. Deletion of redundant global expressions.
3. Deletion of useless or inaccessible code.

A good understanding of the program graph and the flood of data that it handles allows the global optimizer to create an elaborate optimization. This is due to the manipulations that the optimization functions perform on an internal representation of the source code. These manipulations include deletion, insertion, and instruction replacement. These global optimization functions are as follows:

4. Anticipation and temporization.
5. Deleting partially redundant expressions.
6. Removing invariant expressions in loops.
7. Strength reduction of loops and processing loop control variables.

In addition, the global optimizer has two other functions characterized by an expansion effect on the generated code. These are as follows:

8. Loop unrolling.

9. Procedure merging (also called in-line insertion).

NOTES:

1. The optimization functions perform at the procedure level. There are no inter-procedural optimizations.

2. There are two types of local or global optimizing improvements:

   - Increased speed in program execution.

   - Decreased volume of code generation, except in optimization cases of loop unrolling (8) and procedure merging (9).

Restrictions in Optimizing

The optimizer follows these rules:

Efficiency Rule	The optimizing functions work only if the application shows an improvement in storage or time efficiency in all possible execution cases of the program.
Coherence Rule	An optimization function must never affect the semantics of a program. If a program executes correctly and conforms to the definition of a language without optimization, then optimization must not cause the program to abort.
Compromised Time and Storage Rule	The optimizer gives greater importance to the optimization functions that contribute a gain in execution time than to those that contribute to the reduction of generated volume of source code.

14.1.4 Optimization Levels

The GPL compiler has five optimization levels. Each level is guided by one of the OPTIMIZE parameter levels, as follows:

OPTIMIZE=0	No optimization
OPTIMIZE=1	Local optimization, limited to the source statement (the source instruction).
OPTIMIZE=2	Local optimization, limited to an extended linear sequence. This is the default level.
OPTIMIZE=3	Global optimization avoiding code expansion (loop unrolling, procedure merging).
OPTIMIZE=4	Global optimization with possible code expansion.

Only the OPTIMIZE=1 level is compatible with the debugging option. This is the default level when DEBUG is specified.

14.2 GLOBAL OPTIMIZER FUNCTIONS

This section describes the different functions of the global optimizer and gives an example of each in the GPL source language. The functions are presented independent of each other. In the examples, you can concentrate on one optimization function at a time, without considering the possible effects from other functions. When you actually use the global and local optimizer, the functions are linked together and have a cumulative effect.

The global optimizer works on the internal image of the source code that is closest to the machine code. It is possible for the optimizer to have a greater effect than shown here in the following examples. For example, the address expression is not developed when indexing an array.

14.2.1 Constant Folding and Copy Propagation

When the optimizer has the operand values of a sub-expression, it can calculate directly the resulting values. By repeating this process, the propagation reaches all the program expression, as long as those expressions are valid.

```
A = 1;
IF VALID THEN
      X = A + 3;
ELSE
      X = A + 1;
```

This gives the following, after optimization:

```
A = 1;
IF VALID THEN
      X = 4;
ELSE
      X = 2;
```

Constant folding and coy propagation use the basic elementary operations (arithmetic, logical, and comparative) in their scope of applications. However, the compiler does not evaluate a constant expression during compilation if the expression causes an exception. An overflow or an illegal operation are examples of exceptions.

14.2.2 Deleting Globally Redundant Expressions

An expression, at a particular point in a program, is globally redundant if it was previously evaluated with the same values, regardless of how the program is running.

In the example below, the expression "A + B" is globally redundant:

```
X =   (blank);
IF A > B THEN
    X = 10;
ELSE
    X = 20;
Y = A + B + D;
```

This optimization function deletes all the redundant expressions in the program. It does this by grouping together all common sub-expressions. After optimization, the above example gives the following:

```
T = A + B;
X = T + C;
IF A > B THEN
    X = 10;
ELSE
    X = 20;
Y = T + D
```

The optimization function interprets the value of the intermediary variable, T, as the value of the already-memorized "A + B" sub-expression. The compiler keeps the sub-expression value in a machine register.

NOTE: This function is legal only if the value of the expression and variable are the same. Consequently, if one of the variables X, A, or B is BASED, it must have the NOMAP attribute to allow the optimization. Optimization would also not be possible, for example, in the following context:

```
S: PROC (X, A, B);
```

This is because S procedure could be called by a statement, resulting in the following:

```
CALL S (A, A, A);
```

14.2.3 Deleting Useless or Inaccessible Code

When using the optimization functions, some program code can become useless or inaccessible. This often occurs after constant folding and copy propagation. This is shown in the following example.

```
BEGIN;
DCL (A, B)       FIXED BIN (15);
     A = 1;
     B = A - 1;
     IF A < B THEN
        C = B;
     ELSE
        C = A;
     C = C * 2;
END;
```

After constant folding and copy propagation, this gives the following:

```
BEGIN;
DCL (A, B)       FIXED BIN (15);
     A = 1;
     B = 0;
     IF 1 < 0 THEN
        C = 0;
     ELSE
        C = 1;
     C = C * 2;
END;
```

Deleting Useless Code

When the optimization functions evaluate the above example, it creates some useless code, with A and B declared in the current block. Deleting the useless code results in the following:

```
BEGIN;
DCL (A, B)        FIXED BIN (15);
     IF 1 < 0 THEN
        C = 0;
     ELSE
        C = 1;
     C = C * 2;
END;
```

Deleting Inaccessible Code

Constant folding and copy propagation can also reveal some inaccessible code. The previous example, which shows this, is reduced to the following:

```
BEGIN;
DCL (A, B)          FB15;
     C = 1;
     C = C * 2;
END;
```

14.2.4 Anticipation and Temporization

Two of the optimization functions reduce the object code, but do not shorten program execution time. These functions either bring forward or set back expressions that use the IF-THEN-ELSE instruction in the program. They move the expressions that are within the THEN and ELSE outside, towards the top or the bottom. In this way, the expressions are evaluated only once. The optimization function that brings an expression forward is called anticipation. The function that sets an expression back is called temporization.

The following is an example of anticipation:

```
IF  U > V THEN DO;
    X = A + B;
    A = U;
END;
ELSE DO;
    X = A + B;
    B = V;
END;
```

This yields the following after optimization:

```
X = A + B;
IF U > V THEN;
    A = U;
ELSE
    B = V;
```

The following is an example of temporization:

```
IF U > V THEN DO;
    A = U;
    X = A + B;
END;
ELSE DO;
    B = V;
    X = A + B;
END;
```

This yields the following after optimization:

```
IF U > V THEN
    A = U;
ELSE
    B = V;
X = A + B;
```

14.2.5 Deleting Partially Redundant Expressions

An expression, at a particular point in a program, is partially redundant if the expression has been already evaluated with the same value in another point in the program. Partial redundancy is weaker than global redundancy.

This optimization function eliminates partial redundancies in the program, without interfering with the coherence rule. Partial redundancy is shown in the example below:

```
IF X = 1 THEN
    X = A + B;
ELSE
    A = 1;
X = A + B;
```

In the example above, the assignment of X = A + B is partially redundant. This is because there is one path that executes it twice, uselessly. In contrast, this assignment is not globally redundant because there is one path where there is no redundancy.

It is possible to eliminate the partial redundancy X = A + B by moving it from the IF instruction into the ELSE instruction, as follows:

```
IF X = 1 THEN
    X = A + B;
ELSE DO;
    A = 1;
    X = A + B;
END;
```

14.2.6 Removing Loop Invariants

An expression located in the body of a loop is invariant when its evaluation remains constant throughout the execution of the loop. In the following examples, the expressions A + B, and SQRT (Y) are loop invariants.

Example 1:

```
DO I = 1 TO 10;
     X (I) = A + B;
END;
```

Example 2:

```
DO I = 1 TO J;
     X (I) = A + B;
END;
```

Example 3:

```
DO I = 1 TO 10;
     IF Y > 0 THEN
        X (I) = SQRT (Y);
END;
```

To remove a loop invariant, the optimization function evaluates all the invariant expressions outside of the loop. This transformation is possible only if does not involve an expression evaluated outside the loop, when there was a path in which that expression was not evaluated before. When the loop invariants are removed from the examples above, the results are as follows.

Example 1, from above, after optimization:

```
T = A + B;
DO I = 1 TO 10;
     X (I) = T;
END;
```

Moving the loop invariant , "A + B", to the top, as in example 1, is successful because there is at least one whole iteration in this loop (in this case, the number of iterations is 10).

In the second example, the lower bound, 1, is known, but the higher bound, J, is not known. The optimization function can rearrange the code without changing the semantics. This simplification allows the optimization function to remove the loop invariant.

Example 2, from above, after rearrangement:

```
IF J >= 1 THEN
   DO I = 1 TO J;
      X (I) = A + B;
END;
```

Example 2, after optimization:

```
IF J >= 1 THEN DO;
    T = A + B;
    DO I = 1 TO J;
       X (I) = T;
    END;
END;
```

It is not possible to remove the loop invariant, SQRT (Y), from the third example. This is because no rearrangement can be made that does not interfere with the coherence rule.

14.2.7 Strength Reduction and Processing of Loop Control Variables

14.2.7.1 Strength Reduction

The strength reduction optimization function replaces, in loops, an expensive operation with one that is equivalent, but more economic. The result of the operation remains the same, but requires less power to accomplish. This optimization function operates on arithmetic multiplication in the following two steps:

Step 1:

The detection of all the variables in the loop, progressing step by step through each iteration. Let X be a variable and K be a loop invariant, progressing as follows:

```
X = X + K
```

Step 2:

The replacement of multiplications of the following type:

```
X * C
```

Where C is a loop invariant by an intermediary variable, T. Variable T is correctly initialized and modified at the end of the loop by the following assignment:

```
T = T + K * C
```

where the product of K * C is evaluated at compile time.

An example of this optimization function is:

```
DO I = 1 TO 10 BY 2;
   X = X + 4 * I;
END;
```

After optimization:

```
T = -4
DO I = 1 TO 10 BY 2;
     T = T + 8;
     X = X + T;
END;
```

14.2.7.2 Processing of Loop Control Variables

When in a loop, the compiler can know the number of iterations, and the loop control test is substituted by an equivalent one. The equivalent test uses one of the intermediary variables that the strength reduction function created.

The example from above (after the strength reduction) can be reformulated to make the loop exit test more specific. This is as follows:

```
    T = -4;
    I = 1;
LAB:
    T = T + 8;
    X = X + T;
    I = I + 2;
    IF I <= 10 THEN GOTO LAB;
```

In this way, the substitute control test, which is possible in this example, leads to the following:

```
    T = -4;
    I = 1;
LAB:
    T = T + 8;
    X = X + T;
    I = I + 2;
    IF T ^= 36 THEN GOTO LAB;
```

This manipulation deletes the induction variable, I (when it is no longer working in the loop) only by adding the assignment of the last value of I at the end of the loop. The example above shows this optimization function as follows:

```
    T = -4;
    I = 11;
LAB:
    T = T + 8;
    X = X + T;
    IF T ^= 36 THEN GOTO LAB;
```

14.2.8 Loop Unrolling

Loop unrolling consists of artificially reducing the number of iterations in a loop and duplicating the body of the loop a certain number of times. The number of duplications depends on the size of the loop and the number of its iterations. This optimization applies only if the number of iterations is known at compile time.

For small size loops, the expansion is total. A small loop is one in which the number of iterations does not exceed 20. In other loops, the expansion is partial, provided that the ratio of expansion is not great. The loop unrolling optimization function limits itself to only the lowest level loops, as shown in the following example.

```
DO I=1 TO 25;
   K = 25 * (I - 1);
   DO J = 1 TO 25;
      X (K + J) = J;
   END;
END;
```

As the number of iterations of this loop is greater than 20, this is not a small loop. After partial expansion, this gives the following:

```
DO I = 1 TO 25;
   K = 25 * (I - 1);
   DO J = 1 TO 5;
          X (K + J) = J;
          J = J + 1;
          X (K + J) = J;
          J = J + 1;
          X (K + J) = J;
          J = J + 1;
          X (K + J) = J;
          J = J + 1;
          X (K + J) = J;
          J = J + 1;
   END;
END;
```

This program can be optimized using the algorithms described above.

14.2.9 Procedure Merging

The optimization function of procedure merging (in-line insertion) works by substituting all the references to procedures and functions with their corresponding code. This speeds the program execution time.

Generally speaking, either all the calls of a procedure are merged, or none of them are. The decision of when to merge procedures is based upon the following criteria.

Intrinsic Criteria

- The procedure is a non-recursive procedure (directly or indirectly)
- The procedure does not contain the OUTLINE attribute.
- All the internal procedures can be merged.
- If the procedure contains a label that is branched to from outside (a non-local GOTO), it must be called only one time.
- Each formal parameter corresponds to an effective parameter (no empty argument in CALL statement).

Criteria Resulting From Implementation

- The procedure size is reasonable after the procedure merging.
- The procedure does not have an aggregate-type parameter, such as an array or structure.
- The procedure does not have a fixed-decimal parameter.
- The procedure does not have a parameter redefined by a variable declared with the DEFINED attribute.
- If containing SELECT instructions or some aggregate variables, the procedure can be called one time at most.
- The procedure is not a function argument of another procedure.

NOTE: The criteria requiring implementation are susceptible to change from one version to another. Therefore, the programmer can not use them to control procedure merging.

Even when the INLINE attribute is used, if at least one of the above criteria is not satisfied, the procedure merging is invalidated, and an observe message is issued.

14.3 USING THE GLOBAL OPTIMIZER

The quality of the code generated with the global optimization functions permits the compiled programs to execute more rapidly. However, because the global optimizer slows the program compilation, it is best to use it only in the final phase of program development.

For the initial testing, it is recommended to use the default optimization level (OPTIMIZE=2). This works well for local optimization running on a linear extended sequence. If the debugging option is running, then only the first optimization level (OPTIMIZE=1) can be used. This level is that running on a source statement.

The global optimizer functions work independently with only one procedure at a time. There are no inter-procedural optimization functions. The procedure calls that are not inserted on line limit the effects.

The use of a non-local goto has an equally limiting effect on the optimizations. It is not advisable to write procedures that are too large, which misuse registered variable declarations.

NOMAP and REDUCIBLE attributes should be used as often as possible because, by default, the variables are MAPPED and the procedures are IRREDUCIBLE. This limits considerably the effect of global optimization.

A. Compiler Limits

<=6768	Number of different indentifiers (median length=16).
128	Maximum number of ISNs.
20	Maximum number of nested blocks.
20	Maximum number of nested levels in a structure.
10	Maximum number of nested iterative do_groups.
20	Minimum number of nested select_groups.
15	Maximum number of declared parameters or arguments.
4096 Kbytes	The size which aggregate can not exceed.
<=65535 bytes	Size of an array element.
<=32767 bytes	Size of a parameter.
20	Maximum number of arrays of labels per block.
32767	Number of lines of a program.
<= 255	Length of a source line.

Composite operators as >, <, ... cannot be split on two source lines.

An arithmetic constant can not be split on two or more source lines.

A CHARACTER STRING constant can not be greater than 256 bytes in the code but can be up to 64K byte long in the INIT attribute.

The length of an identifier can not exceed 31 characters.

A statement cannot contain references to more than 100 different variables or labels.

B. Compiler Messages

These are messages that are displayed in the JOR by the GPL compiler.

```
CCG00. COMMON CODE GENERATOR VERSION vv.nn <update-id>
```

Meaning:	Information message indicating the version of the code generator used by the compiler. <update-id> gives its modification level.

```
GPL00.(vv.nn) SUMMARY FOR program_name
      <error-summary> [no] CU produced
```

Meaning: Information message displayed for each external procedure compiled by the activation of the GPL compiler. It indicates:

- The version of the compiler: vv.nn.
- The name of the compiled program.
- The summary of errors detected, i.e. the number of errors for each severity level if relevant or the phrase 'NO ERROR'.
- Whether or not a CU was produced.

```
GPL.K1  ERROR WHEN OPENING THE PRTLIB
         RC = edited return code
```

Meaning:	The GPL compiler failed to OPEN the permanent report file for the reason indicated by the return code. The most common user error is: RC = EFNUNKN; the report file specified has not been found on the indicated volume
Result:	The compiler proceeds but the listing is produced in the standard SYSOUT file, so it will be deleted after being printed.
Action:	Correct the JCL if relevant or contact the Service Center.

```
GPL.K2. ERROR WHEN OPENING THE PRTFILE
         RC = edited return code.
```

Refer to message GPL.K1.

```
GPL.K3. ERROR WHEN OPENING THE INLIB
```

Meaning:	The GPL compiler failed to OPEN the file containing source programs to be compiled. The file can be either a library or an input enclosure. The most common return code is EFNUNKN which indicates that the library specified in the INLIB parameter of the GPL JCL statement has not been found on the specified volume.
Result:	No compilation is performed.
Action:	Correct the JCL if relevant or contact the Service Center.

```
GPL.K4. ERROR WHEN OPENING THE CULIB, OBJECT CODE WILL NOT BE
        PRODUCED.
        RC = edited return code
```

Meaning:	The GPL compiler failed to OPEN the library where the produced objects should be stored for the reason indicated by the return code. Such a message is very unusual when a temporary CU library is used. When a permanent CU library is used, the most common error is: `RC = EFNUNKN` which indicates that the library specified in the CULIB parameter of the GPL JCL statement has not been found on the specified volume.
Result:	The compiler continues its processing. It will not generate any CUs.
Action:	Correct the JCL if relevant or contact the Service Center.

```
GPL.K5. ERROR WHEN PROCESSING SOURCE LIST (BUILD).
         RC = edited return code
```

Meaning:	A problem occurred when the compiler attempted to retrieve source member names from the input library. The reason is indicated by the return code.
Result:	No compilation is performed.
Action:	Report the problem to the Service Center.

```
GPL.K6. ERROR WHEN OPENING INLIB SUBFILE member-name (OPENS).
         RC = edited return code.
```

Meaning:	An incident occurred when the compiler attempted to access a source member from the input library. The reason is indicated by the return code. The incident may be due to a system error, but the most common error is that the member does not exist (return code: EFNUNKN).
Result:	The compilation is aborted.
Action:	Report the problem to the Service Center.

```
GPL.K7. ERROR WHEN OPENING PRTLIB SUBFILE proc_name_L (OPENS)
```

Meaning:	An incident occurred when the compiler attempted to create the member that will receive the listing created by the compilation. Note that the name of the member is derived from the procedure name by adding the "_L" suffix. The reason is indicated by the return code. The incident may be due to a system error.
Result:	The listing will be stored in the standard SYSOUT file, so it will be deleted after being printed.
Action:	Report the problem to the Service Center.

```
GPL.K8. ERROR WHEN CLOSING INLIB SUBFILE member_name (CLOSES)
         RC = edited return code
```

Meaning:	An incident occurred at the end of reading the source program. The reason is indicated by the return code. Such an incident is very unusual and may be due to a system error.
Result:	The compilation of the source program continues.
Action:	Check that the compilation was correctly performed and report the problem to the Service Center.

```
GPL.K9. ERROR WHEN CLOSING PRTLIB SUBFILE proc_name_L (CLOSES)
         RC = edited return code
```

Meaning:	An incident occurred at the end of the creation of the listing in the print library. The reason is indicated by the return code. Such an incident is very unusual and may be due to a system error. Note that the name of the member is derived from the procedure name by adding the "_L" suffix.
Result:	If no serious error has been detected in the source program, the CU will already have been generated at the time of the incident. The listing may however be accessible from the print library.
Action:	Report the problem to the Service Center.

```
GPL.K10 ERROR WHEN CLOSING CULIB
         RC = edited return code
```

Meaning:	An incident occurred when the compiler attempted to CLOSE the CU library. The return code gives the reason of the incident. The incident may be due to a system error.
Result:	The CUs have already been produced and may be accessible in the CU library.
Action:	If the incident was an I/O error, check the disk drive and the disk pack. Contact the Service Center if necessary.

```
GPL.K12. ERROR WHEN CLOSING INLIB
          RC = edited return code
```

Meaning:	An incident occurred when the compiler attempted to CLOSE the library containing the source programs. The return code gives the reason for the incident. The incident may be due to a system error.
Result:	The compiler processing continues.
Action:	If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Cen ter if necessary.

```
GPL.K12. ERROR WHEN CLOSING SYSOUT
          RC = edited return code
```

Meaning:	An incident occurred when the compiler attempted to CLOSE the standard SYSOUT file containing the report. The return code gives the reason for the incident. The incident may be due to a system error.
Result:	This incident occurs at the end of compiler processing when the CUs have already been produced. The listings may be accessible and may have successfully printed.
Action:	If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL.K12. ERROR WHEN CLOSING PRTLIB
          RC = edited return code
```

Meaning:	An incident occurred when the compiler attempted to CLOSE the report file, either a library specified in the PRTLIB parameter of JCL or a sequential file specified in the PRTFILE parameter of JCL. The return code gives the reason for the incident. The incident may be due to a system error.
Result:	Refer to the preceding message.
Action:	Refer to the preceding message.

```
GPL.K13. ERROR WHEN WRITING ON SYSOUT (PUT)
          RC = edited return code
```

Meaning:	The compiler was unable to write a record to the standard SYSOUT file containing the report. The reason is indicated by the return code. Such an incident is very unusual and may indicate a system error.
Result:	The compiler stops. However, the generation phase of the current program has already been performed. Message GPL00 in the JOR indicates which CUs have already been produced. A partial listing of the program being processed may be accessible.
Action:	If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL.K13. ERROR WHEN WRITING ON PRTLIB (PUT)
          RC = edited return code
```

Meaning:

The compiler was unable to write a record in the report file, either a library specified by the PRTLIB parameter or a sequential file specified in the PRTFILE parameter. The return code indicates the reason for the incident. The return code DATALIM means the file or library is full and can no longer be extended. Note that:

- The PRTFILE is processed in append mode.
- In the PRTLIB, the listing of the procedure "procname" is stored in the member "procname_L" and replaces those created by a previous compilation of "procname".

Result:

Refer to the preceding message.

Action:

Refer to the preceding message.

```
GPL.K14. ERROR WHEN READING member_name FROM INLIB (GET)
         RC = edited return code
GPL.A61. ERROR WHEN READING member_name FROM INLIB (GET)
```

Meaning:	The compiler was unable to read a source record either from a user library or from the standard SYS.IN library. The reason of the incident is given by the return code. Such an incident is very unusual and may indicate a system error.
Result:	The member is not compiled, control passes to the next member.
Action:	If the incident was an I/0 error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL.K15. THE SOURCE MEMBER member_name IS EMPTY
```

Meaning:	The specified member given as input to the compiler does not contain any records.
Result:	If several compilations were requested, the compiler goes to the next compilation.

```
GPL.K16. ERROR WHEN OPENING THE SYSOUT
          RC = edited return code
```

Meaning:	The compiler was unable to OPEN the standard SYSOUT file in order to create the compiler report. The return code gives the reason for the incident. Such an error is very unusual and may indicate a system error.
Result:	The compiler stops. The CU has already been produced in the CU library.
Action:	If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL.K17. THE SOURCE MEMBER member-name DOES NOT EXIST IN THE
INLIB
```

Meaning:	The user asked for compilation of a source program present in the member "member_name" but the member does not exist in the specified source library.
Result:	If several compilations were requested, the compiler goes on the next member.
Action:	Correct the JCL.

```
GPL.K18. ERROR WHEN READING THE CR101 FOR member-name
          RC = edited return code
```

Meaning:	The compiler was unable to read the control record 101 either from a user library or from the standard SYSIN Library. The reason for this incident is very unusual and may indicate a system error.
Result:	The control record is ignored.
Action:	As for GPLK.16.

`GPL.K19. THE SOURCE MEMBER member_name IS NOT IN SSF FORMAT`

Meaning:	The source given in input to the compiler is not in SSF format and so the compiler cannot process it. This error may occur when an input enclosure is used and no "TYPE" parameter is specified in the $INPUT JCL statement. In this case TYPE = DATA is assumed.
Result:	The compiler stops.
Action:	If the source was in an input enclosure use TYPE = DATASSF or TYPE = GPL in the $INPUT JCL statement. If the source was in a permanent library, use Library Maintenance to create a member in SSF format. More detailed explanations on data format and data types can be found in the Library Maintenance Reference Manual.

`GPL.K20. THE SOURCE MEMBER member_name IS NOT IN GPL LANGUAGE`

Meaning:	The source member given as input to the compiler has neither TYPE=DATASSF nor TYPE = GPL.
Result:	The compiler proceeds, but the results may be unpredictable if the input text is JCL commands or a COBOL source program.
Action:	Change the TYPE of the source member using the Library Maintenance processor. More detailed explanations on source member types can be found in the Library Maintenance Reference Manual.

`GPL.K21. THE TYPE OF THE INLIB LIBRARY SHOULD BE SL`

Meaning:	The library given as input to the compiler is not a Source Language library.
Result:	The compiler stops.
Action:	Check the JCL and correct it if relevant.

`GPL.K21. THE TYPE OF THE CULIB LIBRARY SHOULD BE CU`

Meaning:	The library specified in the CULIB parameter is not a Compile Unit library.
Result:	The compiler stops.
Action:	Check the JCL and correct it if relevant.

`GPL.K21. THE TYPE OF THE PRTLIB LIBRARY SHOULD BE SL`

Meaning:	The library specified in the PRTLIB parameter is not a Source Language library.
Result:	The compiler stops.
Action:	Check the JCL and correct it if relevant.

```
GPL.K22. THE COMPILER GIVES UP IN phase-name PHASE
          WHEN PROCESSING proc-name.
```

Meaning:	Information message displayed when a compilation is halted. It indicates: - The name of the program which was currently processed: proc_name. - In which phase of compilation the processing was given up: phase_name. Another message in the JOR or in the report usually says why the processing has been suspended.

```
GPL.K23  THE COMPILER ABORTS IN phase-name PHASE WHEN COMPILING
          member_name
```

Meaning:	This message is displayed when the compiler aborts due to an internal error. It indicates: - The name of the program which was currently processed: proc_name. - In which phase of compilation the compiler aborted: phase_name.
Action:	Report the problem to the Service Center.

```
GPL.K30  UNKNOWN TARGET COMPUTER
```

Meaning:	The value requested for target code is unknown, (CODE parameter in JCL statement).
Result:	The compiler stops
Action:	Correct the CODE parameter (see paragraph 4.2.1).

```
GPL17. OPENS CULIB WORK MEMBER: member_name
        RC = edited return code
```

Meaning:	The compiler was unable to create the member in the CU library to receive the CU being generated. The return_code indicates the reason for the incident.
Result:	The compiler stops. The old version of the program being compiled is still available in the CU library because the compiler creates the new version of the CU in a temporary member and replaces the old version by the new one only when the CU generation phase is completed.
Action:	Refer to the preceding message.

```
GPL18. OPENS CULIB OLD_MEMBER: member_name
        RC = edited return code
```

Meaning: The compiler was unable to access the member that contains the old CU version of the program being compiled in order to replace it by the new version. The return code indicates the reason for the incident; it is probably due to a system error.

Result: The compiler stops.

Action: If the incident was an I/O error, check the disk drive and the disk pack supporting the file, contact the Service Center if necessary.

```
GPL19. procname IS ALREADY AN ALIAS IN CULIB. DUPLICATE NAME
```

Meaning: An attempt was made to create a CU while there already exists in the library another CU which contains either a secondary entry point whose name is the same as the name of the procedure being compiled.

Example:

```
 Procedure P1: PROC     P2:  PROC:
                 .
                 .
           P2 :ENTRY ;
                 .
                 .
                 .
            END P1 ;   END P2 ;
```

The procedure P1 is already compiled; in the CU library directory there exist two entries P1 and P2. Both entries lead to the same member; P2 is said to be an alias of P1 therefore a new member P2 cannot be added to the library.

Result: The new CU cannot be created in the library.

Action: Use the LIST command of Library Maintenance CU to get the name of the procedure that contains the secondary entry point then rename the new procedure or use a new CU library.

```
GPL20. GET CULIB OLD MEMBER: member_name
        RC = edited return code
```

Meaning: In order to replace an old version of the CU by the new one, the compiler reads the old CU. An incident occurs while reading a record. The return code gives the reason for the incident. It is probably due to a system error.

Result: The compiler stops.

Action: Refer to message GPL18.

```
GPL21. PUT CULIB WORK MEMBER: member_name
```

Meaning:	The compiler was unable to write a CU record in the CU library. The reason for the incident is given by the return code. The return code DATALIM indicates that the CU library is full. The compiler generates the CU in a work member before replacing the old version of the CU by the new one. In this way, enough room must be provided in the CU library to create the work member even when an old version of the CU already exists in the CU library.
Result:	The compiler stops. The old version of the CU, if it exists, is still available in the CU library
Action:	Compile the program again using another CU library. If the incident was an I/O error, check the disk drive and the disk pack supporting the library or delete the old CU using the Library Maintenance CU.

```
GPL22. STOW(ADD) CULIB ALIAS alias_name TO member_name.
        RC =  edited return code
```

Meaning:	The compiler is compiling the GPL program named "member_name". This program contains a secondary entry point named "alias_name". The compiler is trying to store in the directory of the CU library, the name of the secondary entry point as an alias of the main entry point i.e. both names will lead to the same CU member. An incident occurred during the operation. The reason for the incident is given by the return code. The most common return code is "DUPNAME". This means that the name of the secondary entry point already exists in the directory of the library either as a main entry point or as the secondary entry point of another procedure.
Result:	The new CU is created in the library but the implied name is not catalogued in the directory as an alias of this CU.
Action:	Use the LIST command to check the contents of the CU library.

```
GPL23. CLOSES(DELETE) CULIB MEMBER: member_name
        RC = edited return code
```

Meaning:	The compiler was unable to delete the old version of the CU in the CU library. The reason for the incident is given by the return code. Such an incident is very unusual and may indicate a system error.
Result:	The compilation continues. Use the LIST command to check the contents of the library.
Action:	If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL24. CLOSES CULIB WORK MEMBER = member_name.
        RC = edited return code
```

Meaning: The compiler was unable to CLOSE the CU work member. The reason for the incident is given by the return code. Such an incident is very unusual and may indicate a system error.

Result: The compiler stops. The old version of the CU is normally available in the CU library.

Action: Refer to the message GPL23.

```
GPL25. STOW(DELETE) CULIB ALIAS: alias name OF member_name.
        RC = edited return code
```

Meaning: An old version of the CU being created already exists in the library. The old version of the program had some secondary entry point whose name was catalogued in the CU library as an alias name of the main entry point. An incident occurred while deleting this alias, the name of which is given in the message. The reason for the incident is given by the return code.

Result: The compilation continues. The alias involved is not deleted from the directory of the library. Further consequences can be:

- Error GPL22 may appear in the same compilation when the compiler tries to add this name as an alias of the new CU.

- The return code ADDROUT may be output at linkage time when this name is referenced.

Action: Refer to message GPL23.

```
GPL28. CHNAME CULIB FROM WORK member_name_1 TO member_name_2.
        RC = edited return code
```

Meaning: An old version of the CU being created already exists in the library. The compiler has created the new version of the CU in a member with a work name (member_name_1). After having deleted the old member, the compiler is renaming the work member with its actual name (member_name_2). An incident occurred during this operation. The reason is given by the return code.

Result: The compiler stops.

Action: If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact the Service Center if necessary.

```
GPL30. VMMACC WORK. RC = edited return code
```

Meaning:	A problem has arisen in the management of the virtual memory files used by the compiler. The return code gives the reason for the incident. Such a message indicates a system error.
Result:	The compiler stops.
Action:	Contact the Service Center.

```
GPL31. VMFOP WORK.
        RC = edited return code
```

Refer to message GPL30

```
GPL32. VMFCL VORK.        RC = edited return code
```

Refer to message GPL30.

```
GPL35. SEGSIZE FCB_POOL
        RC = edited return code
```

Refer to message GPL30.

```
GPL47. VMM TABLE OVERFLOW
```

Meaning:	An internal problem has arisen in the management of the virtual memory work files used by the compiler.
Result:	The compiler stops.
Action:	Report the problem to the Service Center.

```
GPL48. UPDATE: ERRONEOUS LENGTH
```

Refer to the message GPL47.

C. Example GPL Program

This short program, called SAMPLE_GPL, reads a text and computes the number of occurrences of each word encountered in the text. These statistics are then output. The program calls external entry points to deal with environment management (I/O and errors).

All the listings produced during the development of the program are given below.

```
*********************************************************************************************
*
*********************************************************************************************
*
**** GCOS7
****
****                                              G P L
****
****                                              VERSION: 80.00  DATED:  SEP 29,1989
****
*****SAMPLE_GPL_L*********************************
20****************************************
*********************************************************************************************
*


Active options are :
OBJ, NDEBUG, WARN, OBSERV, MAP, NDCLXREF, XREF, LIST, NDEBUGMD, CASEQ, ILN, OBJCD, LEVEL=GPL,
OPTIMIZE=STATEMENT
10:54:36  MAY 31, 1990 X3463.1   Compilation of LSFY.DOC.SLLIB: SAMPLE_GPL

Sample_gpl : PROC;
/*
  This little program is intended to read a text and to compute
  the number of occurrences of each word encountered in the text
  and then to output these statistics.
  It calls external entry points dealing with environment
  management (i_o and errors).
*/
%REPLACE Buffer_max_length BY 100;
%REPLACE Max_number_of_words BY 200;
%REPLACE Max_word_length BY 20;
DCL 1 Input_interface EXTERNAL STATIC,
2 Buffer_length FIXED BIN (15),
2 Buffer CHAR (Buffer_max_length),
2 End_of_file BIT (1) INIT ("0"b) ;
DCL (Read_in_buffer, Write_word_array) ENTRY EXTERNAL;
DCL  Error  ENTRY (CHAR (*) INPUT) EXTERNAL;
DCL 1 Word (Max_number_of_words) EXTERNAL STATIC,
    2 Name CHAR (Max_word_length),
    2 Counter FIXED BIN (15);
%REPLACE Separators BY ",.;'()? ";
%REPLACE Letters BY "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
DCL  Number_of_words FIXED BIN (15) EXTERNAL STATIC INIT (0);
DCL  Buffer_index FIXED BIN (15),
  Length_of_word FIXED BIN (15),
  I FIXED BIN (15);
    DO FOREVER;                              /* Get a new buffer */
      CALL Read_in_buffer;
      IF End_of_file
      THEN LEAVE;
      Buffer_index = 1;
      DO FOREVER;
        I = VERIFY (SUBSTR (Buffer, Buffer_index, Buffer_length-Buffer_index), Separators);
        IF I = 0 THEN LEAVE;
        Buffer_index = Buffer_index + I - 1;
           I = VERIFY (SUBSTR (Buffer, Buffer_index, Buffer_length-Buffer_index), Letters);
           SELECT (I);
                 WHEN (1) DO;
                      CALL Error ("Illegal character " !! SUBSTR (Buffer, Buffer_index,1));
                      Buffer_index = Buffer_index + 1;
                 END;
                 WHEN (0) DO;
                      Length_of_word = Buffer_length - Buffer_index + 1;
                      CALL Process_word (SUBSTR (Buffer, Buffer_index, Length_of_word));
                      Buffer_index = Buffer_index + Length_of_word;
                 END;
                 OTHER DO;
                      Length_of_word = I - 1;
                      CALL Process_word (SUBSTR (Buffer, Buffer_index, Length_of_word));
                      Buffer_index = Buffer_index + Length_of_word;
                 END;
           END;                                    /* Select */
           IF Buffer_index > Buffer_length THEN LEAVE;
        END;
      END;
      CALL Write_word_array;

Process_word : PROC (Cur_name);
DCL  Cur_name CHAR (*) INPUT;
DCL  i FIXED BIN (15);
         DO i = 1 TO Number_of_words;
                    IF Cur_name = Word (i).name
                    THEN DO;
                         Word (i).Counter = Word (i).Counter + 1;
                         RETURN;
                    END;
         END;
         IF Number_of_words = Max_number_of_words
         THEN CALL Error ("Sorry, your vocabulary is too wide for me");
         IF MEASURE (Cur_name) > Max_word_length
         THEN CALL Error ("Sorry, this word is too long");
         Number_of_words = Number_of_words + 1;
         Word (Number_of_words).Name = Cur_name;
         Word (Number_of_words).Counter = 1;
      END Process_word;
  END Sample_gpl;
```

Figure C-1. Compiler Source Listing

```
SAMPLE_GPL                        /  0     08/                  PROCEDURE
INPUT_INTERFACE                   /  1     00/                  DATA
NUMBER_OF_WORDS                   /  2     00/                  DATA
   3 SYMDEFS GENERATED:    2 REFERENCE DATA.       1 REFERENCE PROCEDURES.
THE ADDRESSES ABOVE REFER TO INTERNAL SEGMENT NUMBERS (ISN'S) WHICH ARE MAPPED INTO
SEGMENT TABLE NUMBERS (STN'S) AND SEGMENT TABLE ENTRIES (STE'S) BY THE STATIC LINKER.
```

Figure C-2. SYMDEF Data Map

```
                                  /  0     0C/=  /  0     10/   SEGMENT NUMBER
SAMPLE_GPL                        /  0     10/                  PROCEDURE
INPUT_INTERFACE                   /  0     14/                  DATA
READ_IN_BUFFER                    /  0     18/                  PROCEDURE
WRITE_WORD_ARRAY                  /  0     1C/                  PROCEDURE
ERROR                             /  0     20/                  PROCEDURE
WORD                              /  0     24/                  DATA
NUMBER_OF_WORDS                   /  0     28/                  DATA
                                  /  0     08/= E/  0    290/   SEGMENT NUMBER
   9 SYMREFS GENERATED:    3 REFERENCE DATA.       4 REFERENCE PROCEDURES.    2 SEGMENT
NUMBERS.
THE ADDRESSES ABOVE REFER TO INTERNAL SEGMENT NUMBERS (ISN'S) WHICH ARE MAPPED INTO
SEGMENT TABLE NUMBERS (STN'S) AND SEGMENT TABLE ENTRIES (STE'S) BY THE STATIC LINKER.
```

Figure C-3. SYMREF Data Map

```
LINKAGE SECTION          /  0     10/      17C     (     380)
CODE SEGMENT             /  0    18C/      2D6     (     726)
DATA SEGMENT             /  1     00/       67     (     103)
DATA SEGMENT             /  2     00/       02     (       2)
```

Figure C-4. Segment Map

```
LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC  LINE:LOC LINE:LOC LINE:LOC LINE:LOC
LINE:LOC

ISN:   0
 1:290    28:298    29:2A4    30:2B2    31:2B6    33:2BC    34:2E6    34:2EE   35:2F2
36:300
37:32A    39:346    40:382    41:388    43:38C    44:39A    45:3E0    46:3E8   48:3EC
49:3F6
50:43C    53:444    53:450    54:454    55:454    56:454    57:18C    60:194   61:1AE
63:1CC
64:1DE    66:1E2    67:1EC    68:1F8    69:224    70:230    71:25C    72:262   73:27C
74:28C
75:460


LOC:LINE  LOC:LINE  LOC:LINE  LOC:LINE  LOC:LINE  LOC:LINE LOC:LINE LOC:LINE LOC:LINE
LOC:LINE

ISN:   0
18C:57    194:60    1AE:61    1CC:63    1DE:64    1E2:66    1EC:67    1F8:68   224:69
230:70
25C:71    262:72    27C:73    28C:74    290:1     298:28    2A4:29    2B2:30   2B6:31
2BC:33
2E6:34    2EE:34    2F2:35    300:36    32A:37    346:39    382:40    388:41   38C:43
39A:44
3E0:45    3E8:46    3EC:48    3F6:49    43C:50    444:53    450:53    454:56   460:75
```

Figure C-5. Line Location Data Map

```
BUFFER_MAX_LENGTH         INTEGER      100                                    9      14
LETTERS                   CHAR_STRING "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef"     22      36
                                      "ghijklmnopqrstuvwxyz"
MAX_NUMBER_OF_WORDS       INTEGER      200                                   10      18      67
MAX_WORD_LENGTH           INTEGER      20                                    11      19      69
SEPARATORS                CHAR_STRING ",.;'()? "                             21      33

2 BUFFER(INPUT_INTERFACE)        ?BR7.4->2   CHAR(100)         EXT STATIC            DCL: 14
                                                              33   36   39   44   49
  BUFFER_INDEX                   ?BR1.18     FIXED BIN(15)       AUTO                  DCL:
24
                                                              31m  33   33   35   35m  36
36   39   40   40m
                                                              43   44   45   45m  49   50
50m  53
2 BUFFER_LENGTH(INPUT_INTERFACE) ?BR7.4->0   FIXED BIN(15)      EXT STATIC    DCL: 13
                                                              33   36   43   53
2 COUNTER(WORD)                  ?BR7.14->14 ARRAY FIXED BIN(15)EXT STATIC    DCL: 20
                                                              63   63m  73m
  CUR_NAME                       ?BR1.C->0   CHAR(*)            PARAM         DCL: 58
                                                              61   72
2 END_OF_FILE(INPUT_INTERFACE)   ?BR7.4->66  BIT(1)             EXT STATIC    DCL: 15i
                                                              29
  ERROR                          ?BR7.10     ENTRY RECURSIVE    EXT           DCL: 17
                                                              39   68   70
  I                              ?BR1.14     FIXED BIN(15)      AUTO          DCL: 26
                                                              33m  34   35   36m  37   48
  I                              ?BR1.1C     FIXED BIN(15)      AUTO          DCL: 59
                                                              60m  61   63   63
1 INPUT_INTERFACE                ?BR7.4->0   STRUCTURE          EXT STATIC    DCL: 12
NO REF
  LENGTH_OF_WORD                 ?BR1.16     FIXED BIN(15)      AUTO          DCL: 25
                                                              43m  44   45   48m  49   50
2 NAME(WORD)                     ?BR7.14->0  ARRAY CHAR(20)     EXT STATIC    DCL: 19
                                                              61   72m
  NUMBER_OF_WORDS                ?BR7.18->0  FIXED BIN(15)      EXT STATIC    DCL: 23i
                                                              60   67   71   71m  72   73
  PROCESS_WORD                   ?BR1.4      PROCEDURE          INT           DCL: 57
                                                              44   49
  READ_IN_BUFFER                 ?BR7.8      ENTRY RECURSIVE    EXT           DCL: 16
                                                              28
  SAMPLE_GPL                     ?BR7.0      PROCEDURE RECURSIVE EXT          DCL: 1    NO
REF
1 WORD                           ?BR7.14->0  ARRAY STRUCTURE    EXT STATIC    DCL: 18   NO
REF
  WRITE_WORD_ARRAY               ?BR7.C      ENTRY RECURSIVE    EXT           DCL: 16
                                                                                   56
```

Figure C-6. Cross Reference List

```
+ + + NO ERROR MESSAGES + + +
     OBJECT CODE PRODUCED
```

Figure C-7. Summary Page

```
***********************************************************************************************
*
***********************************************************************************************
*
**** GCOS7
****
****                                    L I N K E R
****
****                                                VERSION: 90.00  DATED: JUN 30,1986
****
*****SAMPLE_GPL_K********************************************* 18  -
2**************************
***********************************************************************************************
*

ADDITIONAL  INFO:     4   5

 1              CODE (DEFAULT):      OBJC     OBJD
*********************************** LINKER CONTROL STATEMENTS
********************************
 2     LIST=S ,
***************************************** TASK=MAIN
*****************************************
 3                    PROCESS OCCURRENCES : P0
 4                    FATHER PROCESSES :    NONE
 5                                          BASE      1ST PAGE   NB.PAGES SH INITSIZE MAXSIZ
 6                    STACK RING 0           8.14     NONE         0      3        0    4096
 7                    STACK RING 1           8.15      8.16        5      3     2048   16384
 8                    STACK RING 2           8.1C     NONE         0      3     2048   16384
 9                    STACK RING 3           8.1D     NONE         0      3     2048   32768
10  ENTRY POINT = SAMPLE_GPL       LOCATION:  8.10.000008     IN CU: SAMPLE_GPL
11 ======================================GROUP
INFORMATION=========================================
12  MINIMUM CONTROL MEMORY REQUIRED :    8416     MINIMUM USER   MEMORY REQUIRED :
12176
13  FIXED SIZE SEGTS. CUMULATED SIZE:   16560     VAR SIZE SEGS CUMUL INITIAL SIZE:
6240
14  VAR SIZE SEGS CUMUL MAXIMAL SIZE:  294912     LOAD MODULE SIZE   :
23861
15                                  CONTROL SEGMENTS
16                SEG NUM                     SEG NUM
17  PGCR          9. 0      PCS               8. 0
18  NPCS          8. 1      ITS LIST          9. 2
19  TASK.DIR.     9. 3      DEBUGGING         9. 5
20  PG PCP S      9. 6      OPTION            9. 7
21  PGFECB        9. 8      DECB              9. 9
22  SEMPH. POOL   9. D      SYMBMAP           9. C
23  TERMINATION   9. 4      ASL2              9. 1
24  ASL3          8. 3
25                                  GLOBAL SEGMENTS
26  SEGNAME      SEG NUM   CONTAINS
27  __BLANK       8.11                      LOCATION                            LOCATION
28                          INPUT_INTERFACE      000000   NUMBER_OF_WORDS           000067
29                          WORD                 000069   INPUT                     001199
30                          OUTPUT               0011A1
31  __SLFICB      9. A                      LOCATION                            LOCATION
32                          H_S_INPUT            000000   H_S_OUTPUT                00004D
33  __REFTAB      9. B                      LOCATION                            LOCATION
34                          H_STND2_TDF1         000004   H_DFPRE_UOPF              000011
35                          H_TASKM_UABT         00001E   H_DFPRE_UCFM              00002B
36                                  SEGMENT  LIST
37  SEG.   IN CU.ISN                TYPE SH RF RD WR EX WP EP  G  S    SIZE MAXSIZE  CONT.P.
38  8. 0  PCS                      .D.  3  3  3  0  0  W                320          *
39  8. 1  NPCS                     .D.  3  3  3  1  0  W                 32          *
40  8. 3  ASL3                     .D.  3  3  1  0  0  W                 16   32768  *
41  8.10  SAMPLE_GPL.0             C.L  3  3  3  3  3     E            1136           0
42  8.11  __BLANK                  .D.  3  3  3  3  3  W               4528           0
43  8.12  ERROR.0                  C.L  3  3  3  3  3     E            1504           0
44  8.13  ERROR.6                  .D.  3  3  3  3  0  W                 16           0
45  ....
46  9. 0  PGCR                     CD.  2  3  3  0  3  W  E            4576
47  9. 1  ASL2                     .D.  2  3  1  0  0  W                 80   32768
48  9. 2  ITS LIST                 .D.  2  0  3  1  0  W                208
49  9. 3  TASK.DIR.                .D.  2  3  3  0  0  W                 48
50  9. 4  TERMINATION              .D.  2  3  3  0  0  W         S       96
51  9. 5  DEBUGGING                .D.  2  3  3  1  0  W                  0   32768
52  9. 6  PG PCP S                 .D.  2  0  1  0  0  W  E               0   32768
53  9. 7  OPTION                   .D.  2  2  3  3  0  W                  0   32768
54  9. 8  PGFECB                   .D.  2  3  1  0  0  W                  0   32768
55  9. 9  DECB                     .D.  2  3  3  1  0  W                  0   32768
```

```
56  9. A  __SLFICB                 .D.  2  3  3  1  3  W                160
57  9. B  __REFTAB                 .D.  2  3  3  0  0  W                 64
58  9. C  SYMBMAP                  .D.  2  3  3  1  0                   240
59  9. D  SEMPH. POOL              .D.  2  3  3  1  1  W         S    3632
60  .....
=======================================LIST OF CU  (S) ======================================
          ERROR                INLIB      CREATED 10:16:12  MAY 28, 1990   BY:     GPL   80.0
                   CU OPTION :  EOD
          SAMPLE_GPL           INLIB      CREATED 10:16:12  MAY 28, 1990   BY:     GPL   80.0
                   CU OPTION :  EOD
***************************************LINKAGE  REPORT**************************************
                 NO ERRORS DETECTED
                 -------------------------
.  OUTPUT MODULE PRODUCED ON LIBRARY   ;009315.TEMP.LMLIB
MODULE IS OF CLASS  (CODE):          0

              NUMBER OF ITEMS PROCESSED
              -------------------------
                - COMPILE UNITS        2
                - SYMDEFS             11 ( PROC     4, DATA     7)
                - SYMREFS             25 ( PROC     7, DATA    18)
                -   CALLED SYSDEFS     4
                -   NB OF CALL '''     6
                - EXT. DATA NAMES     11
                - SEG.ENTRIES USED   510 (TYPE 2  255,TYPE 3  255)
                   TYPE 2 VACANT     241
                   TYPE 3 VACANT     225 IN        MAIN
******************************************L*I*N*K*E*R***************************************
*************************************** END OF SESSION **********************************LAST
PERCENTAGE OF SPACE USED           2
```

Figure C-8. LINKER Listing

Index

Technical publication remarks form

Title :	DPS7000/XTA NOVASCALE 7000 GPL User's Guide Languages: General

Reference N° :	47 A2 36UL 03	**Date:**	**July 1990**

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please include your complete mailing address below.

NAME : ______________________________ Date : __________

COMPANY : ______________________________

ADDRESS : ______________________________

____

Please give this technical publication remarks form to your BULL representative or mail to:

Bull - Documentation D$^{ept.}$
1 Rue de Provence
BP 208
38432 ECHIROLLES CEDEX
FRANCE
info@frec.bull.fr

Technical publications ordering form

To order additional publications, please fill in a copy of this form and send it via mail to:

BULL CEDOC		
357 AVENUE PATTON	**Phone:**	+33 (0) 2 41 73 72 66
B.P.20845	**FAX:**	+33 (0) 2 41 73 70 66
49008 ANGERS CEDEX 01	**E-Mail:**	srv.Duplicopy@bull.net
FRANCE		

CEDOC Reference #	**Designation**	**Qty**
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		

[_ _] : The latest revision will be provided if no revision number is given.

NAME: ______________________________ Date: __________

COMPANY: ______________________________

ADDRESS: ______________________________

____

PHONE: ____________________ FAX: ____________________

E-MAIL: ______________________________

For Bull Subsidiaries:

Identification: ______________________________

For Bull Affiliated Customers:

Customer Code: ______________________________

For Bull Internal Customers:

Budgetary Section: ______________________________

For Others: Please ask your Bull representative.

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
47 A2 36UL 03