C Language

User's Guide

DPS7000/XTA
NOVASCALE 7000

Languages: C

REFERENCE
47 A2 60UL 06

DPS7000/XTA NOVASCALE 7000

C Language

User's Guide

Languages: C

February 2005

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
47 A2 60UL 06

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

Preface

Scope and Objectives

This manual provides information about the C language under the GCOS 7 operating system.

It describes how to compile, link, execute, debug and maintain C programs with a maximum of efficiency, and how to use the special macros available to C programmers under GCOS 7 (the Run-Time Package).

Intended Readers

This document is primarily intended for C programmers who wish to use the C language under GCOS 7, though it should also be suitable for inexperienced C programmers if they have available the necessary additional documentation mentioned below.

Structure

The manual is divided into two parts. The first part is through section 9. It describes how to use the relevant GCOS 7 tools to produce your program, and may be read completely by those unfamiliar with this operating system. The second part is from section 9 through section 25. It deals with the Run-Time Package, and is meant primarily for reference.

Section 1	gives an overview of the GCOS environment of C programs.
Section 2	gives an example of how to produce, compile, link, execute and check a simple C program in interactive mode.
Section 3	describes the compilation of C programs.
Section 4	describes the linking of C programs.
Section 5	describes execution and debugging.
Section 6	describes various programming techniques for reducing the size and increasing the execution speed of C programs.
Section 7	describes certain particularities in the behavior of C under GCOS 7.
Section 8	describes the C language building packages.

Section 9	describes the optimizing C programs.
Section 10	deals with the run-time environment.
Section 11	deals with general I/O considerations.
Section 12	deals with file processing.
Section 13	deals with formatting I/O.
Section 14	deals with memory allocation, conversions, environment functions and the random number generator.
Section 15	deals with character handling.
Section 16	deals with string handling, buffer management and memory management.
Section 17	deals with non-local jump.
Section 18	deals with the mathematical package.
Section 19	deals with time functions.
Section 20	deals with STDARG functions.
Section 21	deals with Diagnostics.
Section 22	describes the signal function.
Section 23	describes error condition reports.
Section 24	describes the localization package.
Section 25	describes the standard definition file.
Appendix A	gives the file and volume syntax.

Associated Documents

The following manuals are referred to in conjunction with the present manual:

- For GCOS 7 JCL functions

JCL Reference Manual *47 A2 11UJ*
JCL User's Guide..... *47 A2 12UJ*

- For GCOS 7 interactive (GCL) functions

IOF Terminal User's Reference Manual:
Part I Introduction to IOF *47 A2 38UJ*
Part II GCL Commands (VBO) *47 A2 39UJ*
Part III Directives and General Processor commands..... *47 A2 40UJ*

IOF Programmer's Manual *47 A2 05UJ*

System Overview..... *47 A2 04UG*

- For creating/modifying C source code

Text Editor User's Guide *47 A2 05UP*
Full Screen Editor User's Guide..... *47 A2 06UP*

- For file access

UFAS-Extended User's.Guide...... *47 A2 04UF*

- 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 debugging C Programs

PCF User's Guide..... *47 A2 15UP*

- For C language definition and C Language Primitives definition:

C Language Reference Manual *47 A2 23TJ*
C Language System Primitives *47 A2 64UL*
TDS C Language Programmer's Guide..... *47 A2 07UT*

Syntax Notation

The JCL/GCL commands described in this document 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
{item} {item} {item}	A list of items enclosed in braces indicates a choice of values. 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.
<u>ITEM</u>	An underlined item is a default value. It is the value assumed if none is specified.
= , $ * / \ .	Enter these special non-alphabetic characters as shown.

EXAMPLES:

(1)

```
[                                  ]
[ WHEN={IMMED                    } ]
[       {[dd.mm.yy.]hh.mm        } ]
[       {+nnnn{W|D|H|M}item      } ]
[                                  ]
```

This means you can specify:

- Nothing at all, in which case WHEN=IMMED applies.
- WHEN=IMMED (the same as nothing at all).
- WHEN=22.30 to specify a time (and today's date).
- WHEN=10.11.87.22.30 to specify a date and time.
- WHEN=+0002W to specify 2 weeks from now.
- WHEN=+0021D to specify 21 days from now.
- WHEN=+005H to specify 5 hours from now.
- 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)
```

❑

Table of Contents

3. Compilation

4. Linking

5. Execution and Debugging

6. Programming Considerations

7. GCOS 7 Specific Considerations

8. Building Packages

9. Optimizing with C

10. Run-Time Environment

11. General I/O Considerations

12. File Processing

13. Formatting I/O

14. The Use of STDLIB_H

15. Character Handling

16. The Use of STRING_H

17. Non-Local Jump

18. Mathematical Package

24. Localization

25. Standard Definition Header File

A. File and Volume Syntax

Table of Graphics

Figure

1. Introduction

1.1 The C Language

The C language is a programming language that is versatile and suitable for a very wide range of applications. C is a structured language in which flow-control is ensured by a system of nested loops. It provides a wide variety of data structures and a powerful set of operators.

1.2 The DPS7 Environment

This manual is about the C language in the DPS7 000 environment. It explains how to compile, link, and run your programs in this environment, and any special restrictions that may apply. You can use the C language in batch or interactive mode.

1.2.1 Batch JCL

In batch mode you submit your work as a "job" which is run "off-line". The job is controlled by a language called JCL (Job Control Language). If you wish to work in batch mode, consult the *JCL Reference Manual* and the *JCL User's Guide*.

1.2.2 Interactive GCL

In interactive mode you submit your work from an "on-line" terminal, from which you can communicate with the system using a language called GCL (GCOS 7 Command Language). Those readers wishing to work in interactive mode should consult the *IOF Terminal User's Reference Manual*.

2. Getting Started

2.1 A C Compiler Session

This section gives you an example of how to build a C program during an interactive session under IOF in GCL mode. The steps in this example are as follows:

1. Enter the user library through LIBMAINT.
2. Enter the editor using the command ED.
3. Write the source.
4. Store the editor work buffer and specify the type of language used: w (cl) essai_c.
5. Leave LIBMAINT (return to S level).
6. Compile the source via the GCL procedure: C. In this case, the compile unit (cu) is produced in TEMP.CULIB$TEMPRY. The listing output option: list is activated.
7. Examine the output listing (in TEMP.SLLIB$TEMPRY).
8. Call the linker using the GCL procedure lk. Here the Load Module (LM) is produced in the temporary library TEMP.LMLIB$TEMPRY.
9. Execute using exec_pg with the STDOUT assigned by default to the user terminal.
10. Re-execute, this time with STDOUT assigned to a sub-file of the user library.
11. Check results in the assigned subfile.

2.1.1 Enter LIBMAINT

Note that the library into which to put the C source code is called lsfy.cc.use.sllib.

```
S: lmn sl lsfy.cc.use.sllib
>>>13:43 LMN 40.02  110 -5
C:
```

2.1.2 Enter the Text Editor

In the editor, type the 'a' (append) command to enter input mode.

```
C: ed
R: a
I:
```

2.1.3 Write the Source Text

At the end, type '/' to get out of input mode.

```
I: /*example of C program */
I: #
I: #
I: #include <stdio_h>
I: #
I: #
I: main()
I:{
I: printf ("Hello! Welcome to the C/GCOS compiler session ! \n");
I:}
I: /
R:
```

2.1.4 Store the Work Buffer

Store the source text in a member called essai_c. Note that the type of data specified is 'cl'.

```
R: w (cl) essai_c
R: /
C:
```

2.1.5 Return to System Level

Type '/' to leave LIBMAINT.

```
C: /
<<<13:48
S:
```

2.1.6 Compile

Use the C (compile C) command, followed by the name of the member, the name of the library and the 'list' option.

```
S: c essai_c lsfy.cc.use.sllib list
>>>14:24 C 0.00
     14:24:36  JUN 03, 1986 X5163.11  compilation of LSFY.CC.USE.SLLIB: ESSAI_C
   CL.10(0.00 ) summary for ESSAI_C: no error,cu produced.
<<<14:25
```

2.1.7 Examine the Listing

Use the editor command 'r' to read the member containing the listing, then the command 'p' (print) to display the listing.

```
S: ed
>>>14:26 ED  10.02   110  -5
R:
R: ressai_c_1
R: ,$p
********************************************************************************
********************************************************************************
****                              GCOS7                                     ****
****                                                                        ****
****                                C                                       ****
****                                                                        ****
****                VERSION:  0.00  DATED: FEB 15, 1985                     ****
****                                                                        ****
********************************************************************************
*****************************ESSAI_C_L******************************************
********************************************************************************

active options are :

OBJ, WARN, OBSERV, NCHECK, NMAP, NXREF, LIST, NEXPLIST,
ILN, LEVEL=STD, LFATAL,
14:24:36
JUN 03, 1986
X5163.11
compilation of LSFY.CC.USE.SLLIB: ESSAI_C
CD=06/03/86 CT=14:00 MF=06/03/86 MT=14:15 SL=CL MN=1
  1
  2 /*example of C program */
  3 #
  4 #
  5 #include <stdio_h>
157 #
158 #
159 main()
160{
161 printf ("Hello! Welcome to the C/GCOS compiler session !  \n") ;
162}
           + + + NO ERROR MESSAGES + + +
                OBJECT CODE PRODUCED
*****************************ESSAI_C_L******************************************
*****************************C*L************************************************
+++
R: /
<<<14:27
```

2.1.8 Scan the Error Messages

The compiler identifies erroneous lines and corresponding error message with a dollar character ($). In this way, you can easily locate them with the scanner or the text editor.

2.1.8.1 Using the Scanner

At the R: level, the /$/$ request prints all erroneous lines and messages.

2.1.8.2 Using Edit

If your listing is on a library, first enter the R request to read the listing. Then, to see the erroneous lines, enter the GL/[C$/ request.

2.1.8.3 Using FSE

If your listing is on a library, first enter the RA request to read the listing. Then, for the erroneous lines, enter the UC*=/[C$/ request.

2.1.9 Link

To link, give the name of the compile unit. When you do not specify an input library, the linker searches for the compiler unit in TEMP.CULIB$TEMPRY. The load module is placed in TEMP.LMLIB$TEMPRY.

```
S: lk essai_c

>>>14:28 LK  82.00    11  -1
WORKING ON: ESSAI_C
LKOO.(82.00)   SUMMARY FOR  ESSAI_C  NO ERROR DETECTED .  OUTPUT MODULE PRODUCED

<<<14:28
```

2.1.10 Execute with Output to User Terminal

Use the command exec_pg (execute program), giving the name of the load module to be executed.

```
S: exec_pg essai_c

   Hello! Welcome to the C/GCOS compiler session !
```

2.1.11 Re-execute with Output to a Subfile

This is another way of testing the program. Here, specify that the output of the C program is to be sent to the member essai_out in the library lsfy.cc.use.sllib.

```
S: exec_pg essai_c file1=stdout asg1=lsfy.cc.use.sllib..essai_out

S: lmn sl lsfy.cc.use.sllib

>>>14:33 LMN 40.02  110 -5
```

2.1.12 Check Results

Use the editor command 'r' (read) to bring the member essai_out into the editor's workspace. Then use the 'p' print command to examine the contents of this member.

```
C: ed

R: ressai_out

R:^,$p

   Hello Welcome to the C/GCOS compiler session !
R: /
C: /
<<<14:34
S:
```

Your test is completed and you are now back at system level.

3. Compilation

3.1 Batch Compilation

3.1.1 Syntax of C Statement in JCL

```
C {   INFILE = (sequential-input-file-description)                          }
  {          {  *input-enclosure-name                                       }}
  {          {{ member-name                                                }}}
  {          {{                                 { INLIB= (input -library }  }}}
             {{                                 {          -description)  }  }}}
  {          {{                                 { INLIB1                  }  }}}
  {SOURCE =  {{  (member-name[member-name]...)   { INLIB2                  }  }}}
  {          {{                                 { INLIB3                  }  }}}
  {          {{                                 {                         }  }}}
  {          {{  (star-name[star-name]...)

   [                 {  TEMP                          }  ]
   [  CULIB      =   {                                  }  ]
   [                 {  (output-library-description)  }  ]

   [                 { SYS.C.INCLUDE                  }  ]
   [  INLIBZ     =   {                                  }  ]
   [                 {  (input-library-description)   }  ]

 [{       }]  [{          }]  [{      }]  [{       }]  [{       }]
 [{ NLIST }]  [{ NEXPLIST }]  [{ NMAP }]  [{ NXREF }]  [{ WARN  }]
 [{ LIST  }]  [{ EXPLIST  }]  [{ MAP  }]  [{ REF   }]  [{ NWARN }]

 [{ OBSERV  }]  [{ NROUND }]  CODE =  { OBJA  }]  [{ LFATAL  }]
 [{ NOBSERV }]  [{ ROUND  }]          { OBJD  }]  [{ LOBSERV }]
 [{         }]  [{        }]          { OBJCD }]  [{         }]

   [          { ANSI     }]  [{      }]  [{     }]  [{         }]  [{         }]
   [ LEVEL = { STANDARD }]  [{      }]  [{     }]  [{         }]  [{         }]
   [          { GCOS 7   }]  [{ OBJ  }]  [{ ILN }]  [{ NCHECK }]  [{ NDEBUG }]
   [          { GANSI    }]  [{ NOBJ }]  [{ XLN }]  [{ CHECK  }]  [{ DEBUG  }]

   [             {               }]  [          ]  [          ]  [     {     } ]
   [ OPTIMIZE = { 0, 1, 2, 3, 4 }]  [ PACKAGE ]  [ PSEGMAX ]  [ K11={ Y/N } ]
   [             {               }]  [          ]  [          ]  [     {     } ]
```

```
[{           {     SYS.OUT                   }  ]
[{ PRTFILE=  { (print-library-description)   }  ]
[{           {                               }  ]
[{ PRILIB =  { (print-library-description)   }  ]

   [ EXPLIB = source-library-description ]

   [ EXPONLY] [INLINE] [MODSTRNG]                          ;
```

3.1.2 Description of Parameters

The following subsections describe the parameters that can be used in the C statement. Note that the following symbolic names refer to standard parameter groups and are described fully in the *JCL Reference Manual*.

Sequential-input-file-description
input-library-description
output-library-description
print-file-description
print-library-description

The sequential-input-file-description defines a sequential file (on a magnetic tape or disk) which has been created by the LIBMAINT utility using the OUTFILE parameter.

The input-library-description defines a library of type SL which has been created by:

```
BLIB TYPE=SL, LIB=input-library-description
     SIZE=n,COMPACT,MEMBERS=m;
```

and whose members have been updated by:

```
MNLIB TYPE=SL, LIB=input-library-description ...;
```

or:

```
EDIT, LIB=input-library-description;
```

Members containing C language source code may be in SARF or SSF. In the latter case they must be of type 'CL'(for C language).

The output-library-description defines a library of type CU which has been created by:

```
BLIB TYPE=CU, LIB=output-library-description SIZE=n,
MEMBERS=m;
```

For each source code member compiled without error a compile unit (CU) is created, containing the object code. Each CU has the same name as the corresponding source member.

The print-file-description defines a sequential file (on a magnetic tape or disk) into which the compilation listing is put.

The print-library-description defines a library of type SL which has been created by:

```
BLIB TYPE=SL, LIB=input-library-description
     SIZE=n,COMPACT,MEMBERS=m;
```

For each C source compiled, the compiler creates a subfile containing the listing. This member has the same name as the source member, suffixed by '_L'.

To summarize, if we compile a C source member called PP, we create:

- If there is no compilation errors, a CU called PP.
- If PRTLIB is specified, an SL member called PP_L.

3.1.2.1 The SOURCE, INFILE, INLIB and INLIBn Parameters

These parameters are used to specify the name and location of the program or programs to be compiled. A series of programs may be compiled during a single execution of the compiler.

One of either the SOURCE or INFILE parameters must be specified in a C statement. All of the other parameters are optional. SOURCE and INFILE may not appear in the same statement.

In using these parameters, the simplest case is when the source program is held in an input enclosure. In this instance the following statement will suffice:

```
C SOURCE = *input-enclosure-name;
```

where the input-enclosure-name is the name of an input enclosure contained in the same job.

EXAMPLE:

```
C SOURCE = *IN;
$INPUT IN, TYPE =DATASSF;
main ()
.
.
.
$ENDINPUT IN;
```

❑

If the source program is held in a library, the name of the library and the member are both specified in the statement as follows:

```
C SOURCE = member-name
  INLIB  = (input-library-description);
```

EXAMPLE:

```
C SOURCE=mb INLIB=product.sllib;
```

This means that the program to be compiled is in the member 'mb' of the catalogued library 'product.sllib'.

However, one or more libraries can also be specified in a single JCL statement as follows:

```
LIB SL INLIB1 = (input-library-description)
       [INLIB2 = (input-library-description)
       [INLIB3 = (input-library-description]];
       C SOURCE = member-name;
```

The LIB JCL statement defines a "search path" for the compiler. The compiler will search for the source program specified by member-name, first in the INLIB1 library, then in the INLIB2 library and finally in the INLIB3 library. The first member found will be compiled and any others of the same name will be ignored.

Note that the LIB JCL statements shown in this section do not contain all possible parameters. See the *Library Maintenance Reference Manual* for further details.

If source programs of the same name occur in the search path, the program to be compiled may be chosen by specifying its library with the INLIBn parameter of the JCL statement in the C language. In this case, the normal search path is overridden by the INLIBn parameter. The statement format is as follows:

```
LIB SL INLIB1 = (input-library-description)
    [INLIB2 = (input-library-description)
    [INLIB3 = (input-library-description)]];

                                {INLIB1 }
    C SOURCE = member-name      {INLIB2 }
                                {INLIB3 }
```

The three methods of specifying a member name and library, described above, may also be used when a series of source programs is to be compiled in a single execution of the compiler (Serial Compilation). In this case the SOURCE parameter must specify a series of member names. For example:

```
C SOURCE = (member-name[,member-name]...)
  INLIB = (input-library-description);
```

Source programs may also be read from a sequential file on disk or magnetic tape (this may be, for example, a tape file written by the LIBMAINT utility using the OUTFILE option). The INFILE parameter is used for this purpose, as follows:

```
C INFILE = (sequential-input-file-description);
```

The file specified in the INFILE parameter can contain one or several source programs.

❑

The Star Convention

As an alternative to specifying a list of member names in the SOURCE parameter, a range of member names can be specified using the "star-convention" (same as the star convention used by the LIBMAINT utility). The following statement is used:

```
LIB SL INLIB1 = (input-library-description)
    [INLIB2 = (input-library-description)
    [INLIB3 = (input-library-description)]];

    C SOURCE = (star-name [star name] ...)

    { INLIB = (input-library-description)  }
    { INLIB1                               }
    { INLIB2                               }
    { INLIB3                               }
```

Note that if the library to be used is specified in the C statement (i.e. no library search is carried out), the INLIB1 is assumed to be specified.

Using the star convention, all the library member names in the specified library having certain common characteristics can be selected for compilation. Conversely, all names having certain characteristics can be excluded from compilation, the rest being compiled. For a description of the star convention, see the *Library Maintenance Reference Manual*.

The parentheses in the SOURCE parameter are mandatory only when there is more than one star-name or when the star-name begins with an asterisk.

When the FROM = and the TO = specifiers are used, the star-name including these specifiers must be enclosed between apostrophes.

3.1.2.2 The INLIBZ Parameter

The INLIBZ parameter specifies the library in which the system include files are kept (such as STDIO_H). The default is the standard system library (SYS.C.INCLUDE). The INLIBZ parameter is equivalent to the INCLUDE parameter in GCL.

To change the name of this library, specify another name in input_library_description. parameter, The library contains all the include files that compile the C programs. For further information on include search rules, see paragraph 3.3.

3.1.2.3 The CULIB Parameter

The CULIB parameter specifies the library in which the resulting compile unit is to be stored. An output-library-description or the word TEMP may be used in the CULIB parameter.

If a library is specified, it must have been previously allocated by, for example, the BLIB command, unless the output-library-description specified in the CULIB parameter contains the SIZE parameter (see the *Library Maintenance Reference Manual*).

If TEMP is specified, the compile unit will be written as a temporary member of a system library. If the CULIB parameter is omitted, this is equivalent to CULIB = TEMP. The member name given to the compile unit will be the same as the name of the member containing the source code.

When linking compile units produced with no CULIB parameter, or with CULIB = TEMP, the compile unit library TEMP should be present in the library search path that precedes the LINKER statement.

FOR EXAMPLE:

```
LIB CU INLIB1 = TEMP
       INLIB2 = ...;
```

❑

However, if TEMP is the only input compile unit library, no JCL statement LIB CU is required to define the search path.

3.1.2.4 The LIST, NLIST, EXPLIST, and NEXPLIST Parameters

These parameters determine the form of the source listing. NLIST specifies that the source program listing is not to be produced. However, lines associated with the production of error messages will be produced. LIST specifies that the complete program listing is to be produced and is the default option.

NEXPLIST is used to suppress the listing of lines of source program which were inserted by the INCLUDE command. Note that lines inserted by LIBMAINT or EDIT and not renumbered will not be listed either in this case.

The combination of NLIST and EXPLIST is meaningless. In fact, there are only three valid combinations:

```
NLIST [NEXPLIST]
LIST NEXPLIST
EXPLIST [LIST]
```

In all these cases, the lines containing errors are printed.

NLIST	Only the lines with errors are printed, in their non-expanded form. If an error is found in source code which is part of a macro expansion, a marker will indicate the macro as the error location.
LIST NEXPLIST	All the lines of the source code are printed as they would have been by another processor such as LIBMAINT. Those lines not in the source code (INCLUDE level=0) but which contain errors are printed as above.
LIST EXPLIST	All lines processed by the compiler are printed just as they have been received from the preprocessor. The preprocessor commands (for example, define, include, and ifdef) are not printed, nor the source code contained in non-active branches of the preprocessor commands (if, ifdef, ifndef). On the other hand, the macros are expanded, and all lines of the 'included' files are also printed, whatever the INCLUDE level may be.

3.1.2.5 The MAP and NMAP Parameters

The MAP parameter produces a data map and line location map. The MAP parameter produces a line location map only if the OBJ parameter is specified (explicitly or by default), and if there are no errors of severity greater than 2 in the program unit. A data map will also be produced in this case, provided that the XREF parameter is specified. The data address information of the data map is inserted in the cross reference listing.

The NMAP parameter specifies that no such listings are required and is the default option.

3.1.2.6 The XREF and NXREF Parameters

The XREF parameter produces a cross-reference listing of data and then included library members in alphabetic order. The format of these listings is described in the CROSS REFERENCE LISTINGS below. NXREF means that no such cross-reference listings are required (default parameter).

3.1.2.7 The NWARN and NOBSERV

These parameters inhibit the sending of warnings (errors of severity 2) and observations (errors of severity 1).

3.1.2.8 The ROUND and NROUND Parameters

The parameter ROUND indicates that operations performed on real numbers should be executed with rounding. The result is increased convergence of scientific calculations at the cost of a reduction in speed, since instructions with rounding use more time.

3.1.2.9 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: DPS7/X5,X07 and 64/DPS`
- `Class C: DPS7/X0,X17,X27,DPS7000`
- `Class D: DPS7/1XX7.`

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

If CODE=OBJCD is used, the program can be run on a class C or D computer. CODE=OBJCD is the default.

A program compiled with CODE=OBJA and executed on a class D computer lead to a loss of precision on floating point results.

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

The LIST command of the LIBMAINT CU processor may be used to get information on the compatibility class of a compiled unit.

It must be interpreted as follows:

CU Class	A-C Compatible	C-D Compatible
0 or none	yes	yes
1	yes	no
2	no	yes
3	no	no
4	unknown	unknown

NOTE:

For your reference, when the C compiler runs under GCOS 7-V3A, the default is OBJA. When the C compiler runs under CGOS 7-V3B, GCOS 7-V5, or GCOS 7-V6, the default value is OBJCD. V6 does not support Class A.

3.1.2.10 The LEVEL and LFATAL and LOBSERV Parameters

These parameters verify that a program complies to the desired level of language. There are two main language levels: the STANDARD level and the ANSI level.

The STANDARD level is fully compatible with the previous release, and it supports programs written in pre-ANSI C. This level complies with the definition in the *C Language Reference Manual*. This is the default level, for reasons of compatibility.

The ANSI level verifies that the program complies to the X3J11 ANSI standard.

Both language levels allow some specific extensions. Mostly, this is the & feature for "by reference" parameter passing. The extensions they allow are as follows:

- GCOS 7 is an extension of STANDARD level.
- GANSI is an extension of ANSI level.

A fatal diagnostic is issued if a feature does not belong to the requested level. This diagnostic is fatal by default (this is confirmed by the LFATAL keyword), and it produces no object code. If the LOBSERV keyword is specified, it produces only an observation (sev 1).

3.1.2.11 The OBJ and NOBJ Parameters

If OBJ is specified the compiler generates a compile unit in the library specified in the CULIB parameter or, by default, in the temporary library. If there has been an error of severity greater than 2 (more severe than observations and warnings), as indicated in the JOR (Job Occurrence Report), the compiler does not generate a compile unit.

If NOBJ is specified the compiler does not generate a compile unit and so compilation time is greatly shortened. For a program with errors of severity greater than 2, no compile unit is produced and the OBJ and NOBJ parameters have no effect.

The NOBJ parameter is especially useful when you want a listing of a compilation of your program but you do not want an execution. If NOBJ is specified the MAP parameter has no effect (NMAP is assumed).

3.1.2.12 The ILN and XLN Parameters

By default the compiler numbers the program source lines, starting from 1 and increasing by 1. This constitutes the Internal Line Number (ILN) which is then used in the table of cross-references and in the sending of error diagnostics at compile time.

The parameter XLN indicates that the table of cross-references and the code/line correspondence table quote the external line number. That is the one that used with the LIBMAINT or EDIT utilities.

3.1.2.13 The CHECK and NCHECK Parameters

The CHECK parameter enables you to generate extra the code. This code verifies any attempt to access array objects outside the bounds of the array.

3.1.2.14 The DEBUG Parameter

This parameter enables you to debug the C program using PCF in symbolic address mode.

3.1.2.15 The PRTFILE and PRTLIB Parameters

These two parameters are optional. The default depends on the execution mode of the compiler:

- In batch mode, the default is PRTFILE = SYS.OUT (and the listing is printed out at the end of the compilation job).
- In interactive mode, the default is PRTLIB = TEMP.
  - Print_file designates a sequential file on disk or tape into which the compiler writes the output listing
  - Print_library designates an SL type library created by the BLIB command.

For each program compiled, the compiler creates a unit containing the output listing. The name of the unit created is that of the source file suffixed by _L.

Thus, compilation of the program whose source member name is PP generates:

- If no error, a CU named PP
- If PRTLIB is specified, a unit named PP_L.

3.1.2.16 The OPTIMIZE Parameter

The compiler requests the optimization level that is specified in the OPTIMIZE parameter. There are five levels of optimization. They are as follows:

OPTIMIZE=0	No optimization. This level produces very inefficient code.
OPTIMIZE=1	The optimization is limited to the source statement. When DEBUG is on, this is the default value because there is no program transformation.
OPTIMIZE=2	Local optimization. The optimization is limited to a basic block, which is a portion of program between two label definitions or branch instructions. This is the default level.
OPTIMIZE=3	Global optimization. The optimization induces some program transformation, which produces efficiently generated code. For more information, see the optimization description in section 9.1.3.
OPTIMIZE=4	Global optimization. To reduce execution time, some procedures or functions can be inserted into the code. For more information, see the optimization description in section 9.2.9.

NOTES:

1. Levels 3 and 4 increase the amount of compilation time. It is best to use them only when a program is fully debugged.
2. Level 4 is useful only in packaging mode, either automatic or manual. Automatic packaging mode is the PACKAGE option, and manual mode is the #pragma PACKAGE. For more information, see section 8.
3. Level 4 can increase the amount of generated code.

3.1.2.17 The PACKAGE Parameter

This parameter specifies that the source file requires automatic packaging. The default is no automatic packaging.

For more information on automatic packaging, see section 8.4.

3.1.2.18 The PSEGMAX Parameter

This parameter specifies the size used to "segcode" this file automatically. If the size of the current code segment is equal to this size, the compiler uses another segment in which to place the object code for the function. Refer to chapter 3.2.2.25 **PSEGMAX Parameter**.

Because there are at most only 12 segment entries, this allows several short segments instead of one large one. This option is especially useful in packaging mode.

The default is no segcoding, in which case the object code for a source file (or a package) is put in a single segment.

3.1.2.19 The K11 Keyword

The parameter K11=Y specifies that every function of the C library is considered as built in. You cannot redefine them to have your own implementation. This allows the compiler to insert some functions in the user code. The K11 keyword is equivalent to the BUILTIN keyword in CLANG command.

3.1.2.20 The EXPLIB Parameter

The EXPLIB parameter specifies an output SL library that stores the result of the preprocessing phase. The C program that results can be then compiled.

The name of the member that is the combination of that of the source member and the suffix "_I".

For example, given the following source member:

```
C SOURCE=MYCFILE INLIB=MYLIB EXPLIB=MYLIB;
```

After execution, this gives the following member:

```
MYCFILE_I in MYLIB.
```

3.1.2.21 The EXPONLY Parameter

The EXPONLY parameter specifies that only the preprocessing phase of the compiler is executed. Any syntactical or semantical errors are not detected. The EXPLIST or EXPLIB option uses this.

3.1.2.22 The INLINE Parameter

The INLINE parameter activates an optimizing function that automatically merges procedures. For more information, see the subsection on procedure merging.

3.1.2.23 The MODSTRNG Parameter

The MODSTRNG parameter specifies if character strings can be modified. If this parameter is not present, the specified level determines the default value. The STANDARD level has modifiable strings as the default, but ANSI does not. In this way, this parameter increases portability of files between levels.

3.2 Interactive Compilation

3.2.1 Syntax of "CLANG" Command in GCL

```
{  CLANG  }
{  CL     }
{  C      }
            [ SOURCE = ( star31 [ star31 ] ... ) ]
            [ INLIB = { INLIB1 | INLIB2 | INLIB3 | lib78 } ]
            [ CULIB = lib78  ]
            [INCLUDE = { lib78 |SYS.C.INCLUDE  } ]
            [OPTIMIZE = { 0 | 1 | 2 | 3 | 4 } ]
            [ PACKAGE = { bool | 0 } ]
            [ LIST = { bool | 1 } ]
            [ MAP = { bool | 0 } ]
            [ DEBUG = { bool | 0 } ]
            [ XREF = { bool | 0 } ]
            [ WARN = { bool | 1 } ]
            [ OBSERV = { bool | 1 } ]
            [ ROUND = { bool | 0 } ]
            [ EXPLIST = { bool | 0 } ]
            [ LEVEL = { GCOS7 | STANDARD | GANSI | ANSI } ]
            [ SILENT = { bool | 0 } ]
            [ PRTLIB = lib78 ]
            [ BUILTIN = { bool | 0 } ]
            [ PSEGMAX = { bool | 0 } ]
            [ EXPLIB = lib78 ]
            [ EXPONLY = { bool | 0 } ]
            [ MODSTRNG = { bool | 0 } ]
            [ INLINE = { bool | 0 } ]
            [ INFILE = file78 ]
            [ PRTFILE = file78 ]
            [ OBJ = { bool | 1 } ]
            [ CODE = name8 ]
            [ LNUMBER = { I | X } ]
            [ LFATAL = { bool | 1 } ]
            [ CHECK = { bool | 0 } ]
            [ BATCH = { 0 | 1 | 2 } ]
```

Parameters:

SOURCE	up to 31 star-names denoting the names of the source programs to be compiled.
INLIB	the library containing the source programs to be compiled. This may be expressed as INLIB1, INLIB2, INLIB3 referring to the source library search path as defined by the MWINLIB SL command, or as a library name. When omitted, the default source output library #SLIB, as defined by the MWLIB SL command, is first assumed; if #SLIB is undefined, a temporary library is first assumed. If the source program is not found there, INLIB1, 2, and 3 are searched next.
CULIB	the library in which the resulting Compile Unit(s) are to be stored. When omitted, the default output CU library #CLIB, as defined by the MWLIB CU command, is assumed. If #CLIB is undefined, a temporary library is assumed.
INCLUDE	the system library in which the included files are stored. The default value is the standard system library SYS.C.INCLUDE.
OPTIMIZE	the optimization level of the generated code. There are 5 levels of optimization:

OPTIMIZE=0	No optimization.
OPTIMIZE=1	Local optimization, at source statement level. (When DEBUG is specified, this becomes the default value.)
OPTIMIZE=2	Local optimization, limited to an extended linear sequence, such as a portion of a program between two label definitions or branch instructions. This is the normal default value.
OPTIMIZE=3	Global optimization, without code expansion.
OPTIMIZE=4	Global optimization, with code expansion (for example, loop unrolling).

PACKAGE	if 1, the source file requires automatic packaging. Packaging produces one compile unit for the entire file. Programs are enhanced by encapsulating data and shortening the time needed to call functions. Performance is improved. The default value is no automatic packaging.
LIST	if 1 (default), a listing of the source program is produced; if 0, no source listing is produced.
MAP	if 1, an allocation map for the data is produced; if 0 (default), no such map is produced.
DEBUG	if 1, the resulting Compile Units are eligible for being debugged by the Program Checkout Facility in symbolic mode; if 0 (default), only "effective address" mode debugging will be available.
XREF	if 1, a cross-reference table of the data is produced; if 0 (default), no such table is produced.
WARN	if 1(default), warning (severity 2) diagnostics are issued; if 0, warnings are not issued.
OBSERV	if 1 (default), observation (severity 1) diagnostics are issued; if 0, observations are not issued.
ROUND	if 1, floating point results are rounded, if 0 (default), floating point results are truncated.
EXPLIST	if 1, included program lines are listed; if 0 (default), included lines are not listed.
LEVEL	the level of the standard which is to be applied checking the language syntax. STANDARD (default) means standard C; GCOS 7 means standard C plus GCOS 7 extensions. ANSI checks that the program complies with the X3J11 ANSI standard, while GANSI is an extension of the ANSI level. Errors are reported together with the erroneous lines at the user's terminal.
SILENT	if 1, errors are not reported at the user's terminal; if 0 (default), errors are reported together with the erroneous lines at the user's terminal.

PRTLIB	the library in which the listing is to be produced. The name of the listing in the library is built up from the program name with suffix _L (for example, MYPG_L for source program MYPG). When omitted, the default printout library #PRTLIB, as defined by the MPRTLIB command, is assumed. If #PRTLIB is undefined, a temporary library is assumed.
BUILTIN	if 1, on-line insertion of C primitives is enabled.
PSEGMAX	if 1, segmentation of the generated code is automatic.
EXPLIB	an output SL library into which the result of the preprocessing phase is written. This result is a C program that can then be compiled. The name of the member created is the name of the source member suffixed by "_I".
EXPONLY	if 1, only the compiler preprocessing phase is executed; syntactical and semantic errors are not detected. The default value is 0.
MODSTRNG	if 1, literal character strings are declared modifiable, thus reducing the differences between the STANDARD and ANSI modes, and increasing portability. The default value is 0.
INLINE	if 1, an optimizing function that performs automatic procedure merging is activated. The default value is 0.
INFILE	an alternative to SOURCE and INLIB designating a sequential file containing the programs to be compiled.
PRTFILE	an alternative to PRTLIB designating a sequential file where the listing is to be stored.
OBJ	if 1 (default), object Compile Units are generated for valid source programs; if 0, no Compile Units are generated, even if the source programs are valid.
CODE	the identification of the machine for which the code is to be generated. Refer to the compiler documentation for the acceptable values.
LNUMBER	if I (default), line numbers appearing in cross-reference table are internal (that is, compiler-generated) line numbers; if X, line numbers are the external (that is, source program) numbers.

LFATAL — if 1 (default), constructs that do not fall into the designated level (LEVEL parameter) are reported as fatal errors; if 0, these are reported as observations.

CHECK — if 1, the CHECK parameter enables you to generate the code corresponding to the verification of the tables during their reference; if 0 (default), no code is generated.

BATCH — if 0 (default), the compilation is executed in interactive mode; if 1, the compilation is executed in batch mode as an absentee job with default batch execution parameters NHOLDOUT, JOR=ABORT, and SEV=3; if 2, the compilation is executed in batch mode, and the user may supply the following job parameters through entry on a separate screen:

```
CLASS = [ A - ZZ ]
PRIORITY = [ 0 - 7 ]
HOLDOUT = [ 0 | 1 ]
DEST = [ name8 ][ .name8 ]
HOLD = [ 0 | 1 ]
BANNER = [ 0 | 1 ]
BANINF = [ char12 [ char12 [ char12 [ char12 ]]]]
JOR = [ NORMAL | ABORT | NO ]
JOBNAME = [ name8 ]
SEV = [ 1 | 2 | 3 | 4 | 5 ]
```

For more information concerning these absentee job execution parameters, refer to the description of the ENTER_JOB_REQ (EJR) directive in Section 2, Part 3.

If BATCH=1, the compilation will be executed as an absentee job with jobid CLANG. If BATCH=2, the absentee jobid will be the value specified by the user; the default value is CLANG.

In batch mode, the BATCH parameter is ignored.

Constraints:

- Either SOURCE or INFILE must be specified.
- INLIB may only be used in conjunction with SOURCE.
- PRTLIB and PRTFILE are mutually exclusive.
- If BATCH>0, no more than three source names can be specified.
- When compilation is requested as an absentee job using the EJR directive with the PROC parameter, the value passed for BATCH must be 0; the following syntax is therefore incorrect:

  ```
  EJR PROC=CLANG VL=(SRC1 BATCH=1)
  ```

- References to statically attached catalogs are not supported in absentee mode (that is, when BATCH>0).

3.2.2 Description of Keywords

3.2.2.1 SOURCE

Up to 31 star-names denoting the names of the source programs to be compiled.

3.2.2.2 INLIB

The library containing the source programs to be compiled. This may be expressed as INLIB1, INLIB2, INLIB3 referring to the source library search path as defined by the MWINLIB SL command, or as a library name. When omitted, the default source output library #SLIB, as defined by the MWLIB SL command is first assumed; if #SLIB is undefined, a temporary library is first assumed. If the source program is not found there, INLIB1, 2, and 3 are searched next.

3.2.2.3 CULIB

The library in which the resulting compile unit(s) are to be stored. When omitted, the default output CU library #CLIB, as defined by the MWLIB CU command is assumed. If #CLIB is undefined, a temporary library is assumed.

3.2.2.4 INCLUDE

The library in which the system include files are kept. When omitted, the standard system library (SYS.C.INCLUDE) is used.

3.2.2.5 OPTIMIZE

The compiler requests the optimization level that is specified in the OPTIMIZE parameter. There are five levels of optimization, 0 through 4, and level 2 is the default.

3.2.2.6 PACKAGE

This parameter specifies that the source file requires automatic packaging. The default is no automatic packaging.

For more information on automatic packaging, see section 8.4.

3.2.2.7 LIST

If 1 (default), a listing of the source program is produced; if 0, no source listing is produced.

3.2.2.8 MAP

If 1, an allocation map for the data is produced; if 0 (default), no such map is produced.

3.2.2.9 DEBUG

If 1, the resulting Compile Units are eligible for being debugged by the Program Checkout Facility in symbolic mode; if 0 (default), only "effective address" mode debugging will be available.

3.2.2.10 XREF

If 1, cross-reference tables of the data and then, included library members are produced; if 0 (default), no such tables are produced.

3.2.2.11 WARN

If 1 (default), warning (severity 2) diagnostics are issued; if 0, warnings are not issued.

3.2.2.12 OBSERV

If 1 (default), observation (severity 1) diagnostics are issued; if 0, observations are not issued.

3.2.2.13 ROUND

If 1, floating point results are rounded; if 0 (default), floating point results are truncated.

3.2.2.14 EXPLIST

If 1, included program lines are listed; if 0 (default), included lines are not listed.

3.2.2.15 LEVEL

The level of the standard which is to be applied when checking the language syntax.

3.2.2.16 SILENT

If 1, errors are not reported at the user's terminal, if 0 (default), errors are reported together with the erroneous lines at the user's terminal.

3.2.2.17 PRTLIB

The library in which the listing is to be produced. The name of the listing in the library is built up from the program name with suffix _L (e.g. MYPG_L for source program MYPG). When omitted, the default printout library #PRTLIB, as defined by the MPRTLIB command, is assumed. If #PRTLIB is undefined, a temporary library is assumed.

3.2.2.18 BUILTIN

The BUILTIN parameter specifies that every function of the C library is considered as built in. You cannot redefine them to have your own implementation. This allows the compiler insert some functions in the user code.

3.2.2.19 PSEGMAX

This parameter set to 1 specifies that this file requires automatic segcoding. The size of the current code segment is the implementation-defined value, the compiler uses another segment in which to place the object code for the function. The current implementation-defined value is 56 Kbytes.

Because there are at most only 12 segment entries, this allows several short segments instead of one large one. This option is especially useful in packaging mode.

The default is no segcoding, in which case the object code for a source file (or a package) is put in a single segment.

3.2.2.20 EXPLIB

The EXPLIB parameter specifies an output SL library that stores the result of the preprocessing phase. The C program that results can be then compiled.

The name of the member that is the combination of that of the source member and the suffix "_I".

For example, given the following source member:

```
C SOURCE=MYCFILE INLIB=MYLIB EXPLIB=MYLIB;
```

After execution, this gives the following member:

```
MYCFILE_I in MYLIB.
```

3.2.2.21 EXPONLY

The EXPONLY parameter specifies that only preprocessing phase of the compiler is executed. The eventual syntactical or semantical errors are not detected. The EXPLIST or EXPLIB option uses this.

3.2.2.22 MODSTRNG

The MODSTRNG parameter specifies if characters can be modified. If this parameter is not present, the specified level determines the default value. The STANDARD level has modifiable strings as the default, but ANSI does not. In this way, this parameter increases portability of files between levels.

3.2.2.23 INLINE

The INLIN parameter activates an optimizing function that automatically merges procedures.

3.2.2.24 INFILE

An alternative to SOURCE and INLIB designating a sequential file containing the programs to be compiled.

3.2.2.25 PRTFILE

An alternative to PRTLIB designating a sequential file where the listing is to be stored.

3.2.2.26 OBJ

If 1 (default), object compile units are generated for valid source programs; if 0, no Compile Units are generated, even if the source programs are valid.

3.2.2.27 CODE

The identification of the machine for which the code is to be generated. Refer to the compiler documentation for the acceptable values. (See 3.1.2.9.)

3.2.2.28 LNUMBER

If I (default), line numbers appearing in cross-reference table are internal (i.e. compiler-generated) line numbers; if X, line numbers are the external (i.e. source program) numbers.

3.2.2.29 LFATAL

If 1 (default), constructs that do not fall into the designated level (LEVEL parameter) are reported as fatal errors; if 0, these are reported as observations.

3.2.2.30 CHECK

The CHECK parameter enables you to generate the code corresponding to the verification of the tables during their reference.

3.2.2.31 BATCH

If 0 (default), the compilation is executed in interactive mode; if 1, the compilation is executed in batch mode as an absentee job with default batch execution parameters NHOLDOUT, JOR=ABORT, and SEV=3; if 2, the compilation is executed in batch mode, and the user may supply the following job parameters through entry on a separate screen:

```
CLASS = [ A - ZZ ]
PRIORITY = [ 0 - 7 ]
HOLDOUT = [ 0 | 1 ]
DEST = [ name8 ][ .name8 ]
HOLD = [ 0 | 1 ]
BANNER = [ 0 | 1 ]
BANINF = [ char12 [ char12 [ char12 [ char12 ]]]]
JOR = [ NORMAL | ABORT | NO ]
JOBNAME = [ name8 ]
SEV = [ 1 | 2 | 3 | 4 | 5 ]
```

For more information concerning these absentee job execution parameters, refer to the description of the ENTER_JOB_REQ (EJR) directive in Section 2, Part 3.

- If BATCH=1, the compilation will be executed as an absentee job with jobid CLANG. If BATCH=2, the absentee jobid will be the value specified by the user; the default value is CLANG.
- In batch mode, the BATCH parameter is ignored.

3.2.3 Constraints

The following constraints apply to the parameters:

- Either SOURCE or INFILE must be specified.
- INLIB can be used only in conjunction with SOURCE.
- PRTLIB and PRTFILE are mutually exclusive.
- TURM version for BATCH related contraints.

3.2.4 Examples

`CL  APG`	(compile program APG from the default SL library into the default library)

`CL  B`
`MYPROJ.MYLIB`
`CULIB=MYPROJ.MYCULIB`
	(compile program B from the specified SL library into the specified library)

`CL  (A* B*)`	(compile programs with names starting with A or B from the default SL lib. into the default library)
`CL  X* XREF MAP` `PRTLIB=MYPROJ.LSTG`	(compile a set of programs with the specified options and printout library)

3.2.5 Interactive Compilation by Menu

The menu below corresponds to the GCL command CLANG.

These parameters are explained in section 3.2.2.

```
*********************************************************************
*                            Emulator xdku                          *
*                                                                   *
*                               CLANG                          -->:*
*                                                                   *
*                          compile C program(s)                     *
*                                                                   *
* SOURCE                   source program names                     *
*                                                                   *
* INLIB                    input library (default is #SLIB)         *
*                                                                   *
* CULIB                    resulting CU library (default is #CLIB)  *
*                                                                   *
* INCLUDE                  system library to include                *
* SYS.C.INCLUDE                                                     *
* OPTIMIZE                 optimization level (0-4)               2 *
* PACKAGE                  compile in packaged mode ?             0 *
* LIST                     produce a listing ?                    1 *
* MAP                      produce a data map ?                   0 *
* DEBUG                    produce a PCF data base ?              0 *
* XREF                     produce a cross reference table ?      0 *
* WARN                     report warnings ?                      1 *
* OBSERV                   report observations ?                  1 *
* ROUND                    rounded arithmetic ?                   0 *
*                                                                   *
* EXPLIST                  list expanded source program ?         0 *
* LEVEL                    GCOS 7,STANDARD,ANSI,GANSI    STANDARD  *
* SILENT                   silent mode ?                          0 *
* PRTLIB                   listing library (default is #PRTLIB)     *
*                                                                   *
* BUILTIN                                                         0 *
* PSEGMAX                  automatic segmentation ?               0 *
* EXPLIB                   output library for expanded source       *
*                                                                   *
* EXPONLY                  preprocessing only ?                   0 *
* MODSTRNG                 constant strings not writtable ?       0 *
* INLINE                   try to inline functions ?              0 *
*                                                                   *
*                                                                   *
*                                                                   *
*                                                                   *
*                                                                   *
* ----------------------------------------------------------------- *
*                                                                   *
*                                                                   *
* EXPL LIGNE                                              L05:C79   *
*                                                                   *
*********************************************************************
```

3.3 Searching for Include Files

There are two forms of syntax for the #include preprocessor command. They are as follows:

#include "member_name"	The standard search path applies to this syntax because the file is assumed to be the user's include file. If INLIB exists, the compiler searches first in the INLIB library, and then it searches in INLIB1, INLIB2, and INLIB3. The compiler searches last in INLIBZ (or INCLUDE) library. For more information about INLIBi, see section 3.1.2.1.
#include <member_name>	This syntax applies for a system include file. The compiler searches in the INLIBZ (or INCLUDE) only.

NOTES:

1. A member_name is the name of the subfile contained in the specified library.
2. For compatibility reasons, a period is converted to an underscore in the member name variable. A slash (/) is converted to a dash (-) and lower case to upper case. For example, dir/name.h becomes DIR-NAME_H.

3.4 Compiler Messages in the JOR

Here is a description of the messages displayed in the JOR by the C compiler.

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

Meaning: This is an information message giving the version of the code generator used by the compiler. The field <update-id> gives the modification level of the compiler.

CCG UNKNOWN GENERATOR CODE:<name>

Meaning: The generation code <name> in option CODE is not accessible.

Result: The compiler aborts.

Action: Correct the CODE parameter (see code option).

CL01 ERROR WHEN OPENING THE PRTLIB.
RC = edited-return-code

Meaning: The C compiler failed to open the permanent report file; the reason is indicated by the return code. The most common return code is RC=EFNUNKN, which indicates that the report file specified has not been found on the specified volume.

Result: The compiler proceeds but the listing is produced in the standard SYSOUT and will be deleted after being printed.

Action: Correct the JCL if it is wrong, otherwise contact your Service Center.

CL02 ERROR WHEN OPENING THE PRTFILE.
RC = edited-return-code

Same as for CL01.

CL03 ERROR WHEN OPENING THE INLIB.
RC = edited-return-code

Meaning: The compiler failed to open the file containing the source program(s) to be compiled. The file can be either a library or an input enclosure. The most common return code is RC=EFNUNKN, which indicates that the library specified in the INLIB parameter of the CL statement has not been found on the volume specified.

Result: No compilation is performed.

Action: Correct the JCL if it is wrong, otherwise contact your Service Center.

CL04 ERROR WHEN OPENING CULIB, OBJECT CODE WILL NOT BE PRODUCED.
RC = edited-return-code

Meaning: The C compiler failed to open the library where the compile unit was to be stored; the reason is indicated by the return code. This message is very unusual when a temporary CU library is being used. When a permanent CU library is being used, the most common return code is RC=EFNUNKN; this indicates that the library specified in the CULIB parameter of the CL JCL statement has not been found on the volume specified.

Result: The compiler continues its processing but does not generate a CU.

Action: Correct the JCL if it is wrong, otherwise contact your Service Center.

CL05 `ERROR WHEN PROCESSING SOURCE LIST (BUILD).`
RC = edited-return-code

Meaning: A problem occurred when the compiler attempted to retrieve source members from the input library. The reason is indicated by the return code.

Result: No compilation is performed.

Action: Report the problem to your Service Center.

CL06 `ERROR WHEN OPENING INLIB SUBFILE member-name (OPENS)`
RC = 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 cause is that the subfile does not exit. (return code: SFNUNKN).

Result: Compilation is aborted.

Action: Report the problem to your Service Center.

CL07 `ERROR WHEN OPENING PRTLIB SUBFILE file-name_L. (OPENS).`
RC = edited-return-code

Meaning: An incident occurred when the compiler attempted to create the member to receive the listing created by the compiler. Note that the name of the member is derived from the file name by adding the "_L" suffix. The reason is indicated by the return code. The incident may have been caused by a system error.

Result: The listing will be stored in the standard SYSOUT; it will, therefore be deleted after being printed.

Action: Report the problem to your Service Center.

CL7 (H_RDN_ELINE): THE SPECIFIED SUBFILE DOES NOT EXIST IN THE LIBRARY.

Meaning: The member specified does not exist in the source library specified.

Result: If several compilations were requested, the compiler advances to the next member.

Action: Correct the JCL.

CL08 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 unusual and may be caused by a system error.

Result: Compilation continues.

Action: Check that the compilation was correctly performed and report the problem to your Service Center.

CL8 (H_RDN_ELINE): MEMBER NOT IN CL LANGUAGE (ILLEGAL SSF DATA TYPE).

Meaning: The source member input to the compiler has not TYPE=CL.

Results: The compiler aborts.

Action: Possibly change the type of the source member using the LIBMAINT utility. (Detailed explanations of source member types can be found in the *Library Maintenance Reference Manual*.) Be sure that the member contains CL source code.

CL09 ERROR WHEN CLOSING PRTLIB SUBFILE file-name_L. (CLOSES).
RC = edited-return-code

Meaning: An incident occurred at the end of creation of the listing in the print library; the reason is indicated by the return code. Such an incident is unusual and may be due to a system error. Note that the name of the member is derived from the file name by adding the "_L" suffix.

Result: The listing may be accessible from the print library. At the time of the incident the CU had already been produced (assuming that no serious errors were detected in the source program).

Action: Report the problem to your Service Center.

CL11 ERROR WHEN CLOSING CULIB
RC= edited-return-code

Meaning: An incident occurred when the compiler attempted to close the CU library; the reason is indicated by the return code. The incident may be due to a system problem.

Result: The CUs had already been produced and may be accessible from the CU library.

Action: If the incident was an I/O error, check the disk drive and disk pack. Contact Field Engineering if necessary.

CL12 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 reason is indicated by the return code. The incident may be due to a system problem.

Result: Compilation continues.

Action: If the incident was an I/O error, check the disk drive and disk pack. Contact Field Engineering if necessary.

CL13 ERROR WHEN CLOSING SYSOUT.
RC = edited-return-code

Meaning: An incident occurred when the compiler attempted to close the standard SYSOUT containing the listing; the reason is indicated by the return code. The incident may be due to a system problem.

Result: The incident occurred at the end of compilation; so the CUs had already been produced. The listing produced may be accessible and may be successfully printed.

Action: If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact Field Engineering if necessary.

CL13 ERROR WHEN CLOSING PRTLIB.
RC = edited-return-code

Meaning: An incident occurred when the compiler attempted to close the report file; the report file may be either a library specified in the PRTLIB parameter or a sequential file specified in the PRTFILE parameter. The return code gives the reason for the incident; it may be due to a system problem.

Result: Same as previous message

Action: Same as previous message

CL15 ERROR WHEN WRITING ON SYSOUT (PUT)
RC = edited-return-code

Meaning: The compiler was unable to write a record in the standard SYSOUT containing the report; the reason is indicated by the return code. Such an incident is unusual and may indicate a system problem.

Result: The compiler aborts. CL00 messages in the JOR indicate which CUs (if any) have already been produced. A partial listing of the program being processed at the time of the incident may be accessible.

Action: If the incident was an I/O error, check the disk pack and the disk drive supporting the file. Contact Field Engineering if necessary.

CL15 `ERROR WHEN WRITING ON PRTLIB (PUT)`
RC = edited-return-code

Meaning: The compiler was unable to write a record in the report file; the report file may be either a library specified in 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 that the file or library is full and cannot be further 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: Same as previous message

Action: Same as previous message

CL16 `ERROR WHEN READING member_name from INLIB (GET)`
RC = edited-return-code

Meaning: The compiler was unable to read a source record either from a user library or from the standard SYSIN library. The reason for the incident is given by the return code. Such an incident is unusual and may indicate a system problem.

Result: The member is not compiled; control passes to the next member.

Action: Same as for CL15

CL17 `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 reason for the incident is indicated by the return code.

Result: The compiler aborts. 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 temporary member and replaces the old version by the new one only when the CU generation phase has been completed.

Action: Same as for CL15.

CL18 `OPENS CULIB OLD_MEMBER: member-name`
RC = edited-return-code

Meaning: The compiler was unable to access the member that contains the old version of the program being compiled; it cannot replace the old version with the new one.

The return code indicates the reason for the incident - it is probably due to a system problem.

Result: The compiler aborts.

Action: If the incident was an I/O error, check the disk drive and the disk pack supporting the file. If, necessary, contact Field Engineering.

CL19 `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 with a secondary entry point (or data with the SYMDEF attribute) whose name is the same as that of the procedure being compiled.

Result: The new CU cannot be created in the library.

Action: Use the LIST command of LIBMAINT 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.

CL20 `GET CULIB OLD MEMBER: member-name`
RC = edited-return-code

Meaning: In order to replace the old version of the CU with the new version, the compiler first reads the old CU. An incident has occurred while reading a record.
The return code indicates the reason for the incident; it is probably due to a system problem.

Result: The compiler aborts

Action: Same as for CL18.

CL21 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. Return code DATALIM indicates that the CU library is full.
Remember that the compiler generates the CU in a work member before replacing the old version of the CU by the new one. Enough room must, therefore, 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 aborts. The old version of the CU (if one exists) is still available from the CU library.

Action: Possibly 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 LIBMAINT CU.

CL22 STOW (ADD) CULIB ALIAS alias-name TO member-name
RC = edited-return-code

Meaning: The compiler is compiling the program named "member-name". This program contains a secondary entry point, or data with the SYMDEF attribute, named "alias-name". The compiler is trying to store, in the directory of the CU library, the name of the secondary entry point (or data with the SYMDEF attribute) as an alias of the main entry point. That is, both names lead to the same CU member.
An incident has occurred during the operation; the reason is given by the return code. The most common return code is "DUPNAME", which means that the name of the secondary entry point (or data with the SYMDEF attribute) already exists in the directory of the library either as a main entry point or as a 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: Possibly use the LIST command of LIBMAINT CU to check the contents of the CU library.

CL23 `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 is indicated by the return code. Such an incident is unusual and may indicate a system problem.

Result: Compilation continues. Use the LIST command of LIBMAINT CU 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 your Service Center if necessary.

CL24 `CLOSES CULIB WORK MEMBER = member-name`
RC = edited-return-code

Meaning: The compiler was unable to close the CU work member; the reason is indicated by the return code. Such an incident is unusual and may indicate a system problem.

Result: The compiler aborts. The old version of the CU is normally available in the CU library.

CL25 `STOW (DELETE) CULIB ALIAS: alias-name OF member-name`
RC = edited-return-code

Meaning: An old version of the CU being produced already exists in the library. The old version of the program had some secondary entry points (or data SYMDEFs) whose names were catalogued in the CU library as alias names of the main entry point. The compiler is deleting these aliases. An incident occurred when the compiler was deleting one of the aliases; the alias name is given in the message. The reason for the incident is indicated by the return code.

Result: Compilation continues. The alias concerned is not deleted from the directory of the library. Further consequences can be:

- Error CL22 in the same compilation when the compiler tries to add this name as an alias of the new CU.
- Return code ADDROUT at linkage time when this name is referenced.

Action: Same as for CL23.

CL26 THE SOURCE MEMBER member-name IS EMPTY

Meaning: The specified input member is empty; it does not contain any records.

Result: If several compilations were requested, the compiler advances to the next compilation.

CL28 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 existed in the library. The compiler has created the new version of the CU in a member with work name "member-name-1". After deleting the old member, the compiler renames the work member using its original name - "member-name-2". An incident has occurred during this operation; the reason is indicated by the return code.

Result: The compiler aborts.

Action: If the incident was an I/O error, check the disk drive and the disk pack supporting the file. Contact your Service Center if necessary.

CL30 VMMACC WORK. RC = edited-return-code

Meaning: A problem has arisen in the management of backing store used by the compiler; the return code gives the reason for the incident. This message indicates a system problem.

Result: The compiler aborts.

Action: Contact your Service Center.

CL31 VMFOP WORK. RC = edited-return-code

Same as for CL30

CL32 VMFCL WORK. RC = edited-return-code

Same as for CL30

CL33 ERROR WHEN OPENING THE SYSOUT.
RC = edited-return-code

Meaning: The compiler was unable to open the standard SYSOUT in order to create the compiler report; the return code gives the reason for the incident. This error is unusual and may indicate a system problem.

Result: The compiler aborts. 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 drive supporting the file. Contact your Service Center if necessary.

CL35 SEGSIZE FCB_POOL. RC = edited-return-code

Same as for CL30

CL37 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 library SYSIN. This incident should be very unusual and may indicate a system problem.

Result: The control record is ignored.

Action: Same as for CL33.

CL38 THE SOURCE MEMBER member-name is NOT IN SSF FORMAT

Meaning: The source program input to the compiler is not in SSF format; the compiler cannot process it. This error may occur when an input enclosure was used and no "TYPE" parameter was specified in the $INPUT JCL statement, in which case the compiler would assume TYPE = DATA.

Result: The compiler aborts.

Action: If the source program is in an input enclosure, specify TYPE=DATASSF OR TYPE=CL in the JCL statement $INPUT. If the source program is in a permanent library, use LIBMAINT to create a member in SSF format (for example, the MOVE command). See the *Library Maintenance Reference Manual* for more details on data format and data type.

CL39 THE SOURCE MEMBER member-name IS NOT IN CL LANGUAGE

Meaning: The source member input to the compiler has neither TYPE=DATASSF nor TYPE=CL.

Results: The compiler proceeds. This may produce strange results if the member does not contain CL source code (for example, if it contains JCL commands or a COBOL source program).

Action: Possibly change the type of the source member using the LIBMAINT utility. (Detailed explanations of source member types can be found in the *Library Maintenance Reference Manual*.)

CL42 THE TYPE OF THE INLIB LIBRARY SHOULD BE SL

Meaning: The library input to the compiler is not a source language library.

Result: The compiler aborts.

Action: Check the JCL, correct it if it is wrong.

CL43 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 aborts.

Action: Check the JCL, correct it if it is wrong.

CL44 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 aborts.

Action: Check the JCL, correct it if it is wrong.

CL45 THE COMPILER GIVES UP IN phase-name PHASE WHEN COMPILING member_name

Meaning: Information message displayed when the compiler has given up (aborted). It indicates:

- the name of the program being processed: proc-name
- the phase of compilation processing: phase-name

Action: Report the problem to your Service Center.

CL46 THE COMPILER ABORTS IN phase-name PHASE WHEN COMPILING member_name

Meaning: This message is displayed when the compiler has aborted due to an internal error. It indicates:

- the name of the program being processed: proc-name
- the phase of compilation the compiler aborted: phase-name

Action: Report the problem to your Service Center.

CL47 VMM TABLE OVERFLOW

Meaning: An internal problem has arisen in the management of backing store working files used by the compiler.

Result: The compiler aborts.

Action: Report the problem to your Service Center.

CL48 UPDATE: ERRONEOUS LENGTH

Same as for CL47

3.5 Compiler Limitations and Restrictions

The maximum number of parameters allowed when calling a function is 128. The maximum number of significant characters in an identifier is 31. For an expression involving nested parentheses, the maximum number of nestings depends on:

- The stack used for syntactic analysis.
- The complexity of the expression.

However, for a simplified expression of the following type, the maximum number of nestings is 75:

```
<list of open parentheses><expressions without parentheses>
<list of closed parentheses>
```

The maximum number of nestings allowed for other structures is as follows:

block	20
structure	20
iterate	20
switch	25
conditional expression	10
conditional compilation	20
constructor type	19
functions call	30
include command	15

3.5.1 Some Compiler Restrictions

The following restrictions apply when compiling.

- The first line of the file to compile must be a blank line in the following situations:
  a. To pass a command string at compilation time.
  b. To define a macro outside a file for a specific purpose.
  c. To include a local file, if necessary.
- The following values cannot initialize an automatic object in the main function:
  a. The value returned by any file-handling functions of the C library.
  b. The argc or argv values.
- In a ternary expression such as E1?E2:E3, E2 must be enclosed in brackets if it is a common expression.

3.6 Compiler Listing

The compiler generates several different listing sections. The options given to the compiler determine which listing is generated. The available listings sections are as follows:

- A source listing with error messages
- Any data segment and line location maps (related to data and code allocation)
- A cross reference listing
- An error summary report

3.6.1 Source and Error Listing

The compiler produces a listing that consists of the following:

- A banner page containing the compiler version. In this example, the compiler version is 40.01.
- A list of the active options. These are either the default values or the values that the user requests.
- The location of the compiled object.

The listing is as follows:

```
*****************************************************************************
*****************************************************************************
**** GCOS7                                                              ****
****                                    C                               ****
****                                VERSION: 40.01  DATED:  JAN 08,1991 ****
*****C_SAMPLE_L**************************************************************
*****************************************************************************

ADDITIONAL  INFO:     9

   active options are :
OBJ, WARN, OBSERV, NCHECK, NMAP, NXREF, LIST, NEXPLIST, NDEBUG, NROUND, ILN,
LEVEL=STD, OBJCD, LFATAL, OPTIMIZE=3
17:08:24  SEP 10, 1991 X1377.5   compilation of ;001377.TEMP.SLLIB: C_SAMPLE
```

This listing also contains the source listing with internal line numbers (ILN), beginning with 1, allocated to the compiler. It can contain any error messages. The dollar ($) character prefixes any erroneous lines.

The following is an example of an error message.

```
**  $ 1 B2 Illegal character, ignored.
```

The form of the error message is as follows:

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

Where:

`aaaa` Either one, two, three, or four asterisks. They indicate the message severity as follows:

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

An observation message indicates the action taken by the compiler when this is not clear from the source code. The NOBSERV or NWARN parameters, when specified in the JCL statement C or the C GPL procedure, suppress the observation messages.

A warning message indicates a possible error. The statement is compiled, but the results can be unexpected. The NWARN parameters, when specified in the JCL statement C or the C GPL procedure, suppress the warning messages.

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 that prevents the compiler from continuing analysis or generating object code. These errors include system, compiler, compiler limit exceeded, user, and use of a feature not included in the level of compilation being used. No compile unit is produced and a message stating this is printed in the summary page and in the JOR. Some of the errors that generate fatal messages include the following:

`code` An identifier of the error that occurred. This identifier consists of a letter and a number. The letter indicates the compiler phase that detected the error. Each letter is described below.

`order-no` When there is more than one error in a line, this number indicates the order in which the errors occurred.

`message-text` A short explanation of the error.

The following table lists each code letter, along with its associated phase and action.

Letter	Phase	Action
A	Option analysis	Correct the option in JCL or GCL command
B	Preprocessor and lexical	Correct the program
C	Syntax analysis	Correct the program to comply with the C syntax
D	semantic analysis of declarations	Correct the program to comply with the C semantic
F	semantic analysis of statements	Correct the program to comply with the C semantic
H	code generation	Internal error: contact your Service Center
I	final allocation	Limit exceeded: restrict your program
N	adapter (allocation)	Generally, limit exceeded: restrict your program
O	local optimizer	Information messages on optimization processing
Q	global optimizer	Information messages on optimization processing
S	optimization services	Limit exceeded: restrict your program or decrease optimization level
Y	debugging	Limit exceeded: restrict your program or do not use symbolic debugging (DEBUG option)

3.6.2 Data Maps

Data maps are produced only if the MAP parameter is specified; NMAP is the default. Three maps are produced:

- SYMDEF map
- SYMREF map
- line location map

A SYMDEF (symbolic definition) is an entry point or data entity within the compile unit, which can be referred to from another compiler 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.

The following shows the SYMDEFS and SYMREFS generated in a compile time example.

```
ACKER_HG            / 0    10/               PROCEDURE
1 SYMDEFS GENERATED: 0 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.

                / 0  0C/=  /  0     18/   SEGMENT NUMBER
                / 0  18/=  /  0     08/   SEGMENT NUMBER
                / 0  14/=  /  0     18/   SEGMENT NUMBER
ACKER_HG        / 0  1C/                  PROCEDURE
H_CLR_EPILOG    / 0  20/                  PROCEDURE
H_CLR_EPROLOG   / 0  24/                  PROCEDURE
H_CLR_EPRINTF   / 0  28/                  PROCEDURE
                / 0  2C/=  /   1     00/  SEGMENT NUMBER
                / 0  08/=  E/  0     30/  SEGMENT NUMBER
                / 0  10/=  E/  0     CA/  SEGMENT NUMBER
10 SYMREFS GENERATED: 0 REFERENCE DATA. 4 REFERENCE PROCEDURES.
6 SEGMENT NUMBERS.
THE ADDRESSES ABOVE REFER TO INTERNAL SEGMENT NUMBERS (ISN'S) THAT
ARE MAPPED INTO SEGMENT TABLE NUMBERS (STN'S) AND SEGMENT TABLE
ENTRIES (STE'S) BY THE STATIC LINKER.
```

3.6.3 Segment Map

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

usage /isn displacement/ size name

Where:

usage	can be LINKAGE_SECTION, CODE SEGMENT, DATA SEGMENT, or P.C.F. DATA BASE.
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 deplacement value of register BR7 at the beginning of the procedure, which is the offset of the linkage section.

```
LINKAGE SECTION      / 0  18/     18   (   24)
CODE SEGMENT         / 0  30/    15E   (   350)
DATA SEGMENT         / 1  00/     33   (   51)
```

3.6.4 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 that contains an address has been output. You can trace the line of source code where the exception occurred on the line location map.

Note that the address given is the byte at which the instructions begin. Also, the instructions generated from one line of source code can occupy a large number of bytes. One line of source code can have more than one entry in the line location map.

The following is an example listing.

```
ISN:    0
132:30   133:38   133:40   135:4A  135:56   137:7E  143:DE 144:10A
145:136   146:180
```

3.6.5 Cross Reference Listings

The data cross reference listing can be used for each object used in the program. It indicates the following about the object:

- Name
- Allocation - for variables
- Class - for other objects
- Data type, storage class, and size for variables and functions
- Line of definition, optionally followed by the line of declaration (enclosed with brackets)
- List of the lines where the object is referenced. The plus (+) symbol denotes multiple referenced, the star (*) symbol denotes a point where the object is modified.

The includes cross reference listing can be used for each valid member included in the program. It indicates the following about the included member:

- Name, translated if necessary to GCOS format (refer to # include command in chapter 7), but with the " " or <> format.
- Location
- Last modification date and time
- List of the lines where the member is included. The plus (+) symbol denotes multiple included, a list of include member names denotes indirect include of the library member where " " denotes an intermediate path.

The following is an example of a data cross reference listing.

```
   _adrec         structure tag                                        65     NO REF
   _dcom          structure tag                                         89
                                REF:  132
   _iob           structure tag                                         71
                                REF:  90   306   311   312
   _its_list      structure tag                                        129
                                REF:  148
   _pcs           structure tag                                        145     NO REF
   div_t          typedef       struct div_t_        typedef           398
                                REF:  490
   div_t_         structure tag                                        394     NO REF
1  error          ?BR1.8        unsigned             auto      4      686
                                REF:  689*  692*  696
   fpos_t         typedef       struct               typedef            70
                                REF:  82    337   339
1  j              ?BR1.C        unsigned             auto      4      686
                                REF:  690*+ 691+  693
   ldiv_t         typedef       struct ldiv_t_       typedef           404
                                REF:  492
   ldiv_t_        structure tag                                        400     NO REF
1  p              ?BR7.44->0    char *[100]          static    400     683
                                REF:  683   691*
   size_t         typedef       long unsigned        typedef            27
                                REF: 310   314   335+  336+  465+  469     471   484+  487+
                                REF: 498   499   501+  502+  617   618     620   622   623
                                REF: 625   626   629   632   634   635     636   640+
   va_list        typedef       char *               typedef            48
                                REF:  321   322   323
   wchar_t        typedef       int                  typedef           386
                                REF:  499   500   501   502
   H_CLR_EMALLOC ?BR7.3C        void *()             extern            0 (  469)
                                REF:  691
   H_CLR_EPRINTF ?BR7.40        int ()               extern            0 (  317)
                                REF:  693   697   699
   TEST_ALLOC1   ?BR7.0         int ()               extern            685
                                REF:  685
```

The following is an example of an includes cross reference listing:

```
DG_C
              INCLUDES CROSS-REFERENCES

  NAME                                          DATE_TIME             ORIGIN

"CMALIB_CRTLX_H"                               06/18/93 14:52     FROM FORT.Z2.SLLIB

             5"GCOSCONF_H""SYSCONF_H""CMA_H"+  5"GCOSCONF_H""SYSCONF_H"

"CTYPE_H"                                      04/04/91 15:10   FROM SYS.C.INCLUDE
             5"GCOSCONF_H""SYSCONF_H"

"DG_H"                                         06/18/93 14:52   FROM FORT.Z2.SLLIB
             37

"DGCCALLT_H"                                   06/18/93 14:52   FROM FORT.Z2.SLLIB
             42"DGCCALLT_H"   43

"DGGLOB_H"                                     06/18/93 14:52   FROM FORT.Z2.SLLIB
             37"DG_H"

"DGGSOC_H"                                     06/18/93 14:52   FROM FORT.Z2.SLLIB
             37"DG_H"

"DGUTIL_H"                                     06/18/93 14:52   FROM FORT.Z2.SLLIB
             37"DG_H"

"EXC_HANDLING_H"                               07/27/93 15:29   FROM FORT.Z2.SLLIB
             5"GCOSCONF_H""SYSCONF_H""CMA_H"     37"DG_H""COMMONP_H"   "RPCEXC_H"

<LOCALE_H>                                     04/04/91 15:12   FROM SYS.C.INCLUDE
             5"GCOSCONF_H""SYSCONF_H""CTYPE_H"
```

The user include CMALIB_CRTLX_H is included several times in CMA_H, which is included in SYSCONF_H, which is included in GCOSCONF_H, which is included on line 5 in DG_C program (XLN). CMALIB_CRTLX_H is also included once in SYSCONF_H and GCOSCONF_H only on the same line.

The system include CTYPE_H is included with the " " format in SYSCONF_H while the system include LOCALE_H is included with the < > format in the CTYPE_H (the system include CTYPE_H is included despite its " " format).

DG_H is included directly on the line 37 of the DG_C program and includes DGGLOB_H, DGSOC_H, and DGUTIL_H.

DGCCALLT_H is included in DGCCALL_H which is included on line 42 of DG_C program and, directly on line 43 of DG_C program.

EXC_HANDLING_H is included via CMA_H, SYSCONF_H, and GCOSCONF_H. EXC_HANDLING_H is included on line 5 of DG_C program and on line 37 of the DG_C program via RPCEXC_H. RPCEXC_H is included by intermediate members whose "parent-" is included via COMMONP_H and DG_H.

3.6.6 Summary Page

The final page of the compilation listing is the summary page. This states whether the compilation produces object code, if any error messages are generated, and gives a summary of the message with line numbers.

When a compilation is successful, generates object code, and produces no error messages, it has a summary page that says only the following:

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

In the example below, one warning message was produced at line 131, and object code was produced.

```
            +++NUMBER OF ERROR MESSAGES+++
            +                           +
            +                           +
            +            *         0    +
            +                           +
            +          * *         1    +
            +                           +
            +        * * *         0    +
            +                           +
            +      * * * *         0    +
            +                           +
            +                           +
            +++++++++++++++++++++++++++++
             ERRONEOUS LINES
  131 **
            OBJECT CODE PRODUCED
```

4. Linking

4.1 General

The LINKER is a utility which builds an executable load module from a set of compile units. These compile units may result from the compilation of programs written in different source languages. The LINKER resolves all references between compile units and sets up links to run-time package procedures and system procedures which are resolved at run-time. This description of LINKER covers the following topics:

- LINKER JCL;
- Serial linkage;
- Interactive use of LINKER;
- LINKER commands of interest to the C programmer;
- Listings of interest to the C programmer.

For a more detailed description of the LINKER listings and commands see the *LINKER User's Guide*.

4.1.1 Segment Numbers

The system recognizes two forms of segment number during compilation, linking, program loading, and execution: the Internal Segment Number and the LINKER Segment Number.

The Internal Segment Number is generated by the compiler to identify the segments within a compiler unit. It appears to the left of the colon in the data map, cross-reference and procedure map produced by the compiler. Internal segment numbers are also included in the segment lists produced by the compiler and the LINKER.

The LINKER Segment Number is generated by LINKER to uniquely identify each segment in the load module. It is formed from a concatenation of segment table number and segment table entry (stn.ste). LINKER segment numbers are included in the segment list produced by LINKER and in the memory dump listing.

4.2 LINKER JCL Statement

The LINKER utility is called by the extended JCL statement LINKER. Figure 4-1 shows the format of the LINKER statement.

```
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) ] ;
```

Figure 4-1. LINKER JCL Statement Format

As the LINKER statement is extended JCL, it must not appear inside a step enclosure. The following example illustrates the use of this statement:

```
$JOB...
    LIB CU    INLIB1=CU.LIB;
    LINKER    PROG_LM
              ENTRY=PROG
              OUTLIB=LM.LIB;
$ENDJOB;
```

The JCL statement LIB CU is used to set up a "search path" for LINKER to enable it to find the referenced compile units. LINKER will look in CU.LIB for a compile unit with a member-name PROG (specified in ENTRY=PROG). This is used as the starting point for building the load module. The resulting load module will be stored in library LM.LIB with the name PROG_LM. Note that, if either the LIB CU statement or the INLIB parameter is used, TEMP will not be included in the search path unless it is specified in one of these statements.

The LINKER utility produces a load module and a listing. The load module may, optionally, be stored in a temporary or a permanent library (OUTLIB parameter). The listing may, optionally, be stored in the standard SYSOUT file or in a permanent library or file (PRTLIB and PRTFILE parameters).

The following paragraphs describe the parameters which may be used in the LINKER statement. Note that the following symbolic names used in Figure 4-1 refer to standard parameter groups which are described in the *JCL Reference Manual*:

```
input-library-description

output-library-description

sequential-input-file-description

print-file-description

print-library-description
```

These parameter groups are not described below. See the *JCL Reference Manual*.

4.2.1 Load-Module-Name Parameter

This parameter is used to specify the name of the load module to be produced by LINKER. The load-module-name must be alphanumeric and must start with a letter. It can be up to 31 characters long.

If there is no ENTRY parameter or command in the LINKER statement, the main compile unit (at which linking starts) is assumed to have the same name as the load module. During the development of a program it is advisable to use the same name for the source program, the compile unit and the load module. It should therefore be normal practice to omit the ENTRY parameter and command from the LINKER statement.

4.2.2 INLIB Parameter

This parameter is used to modify the search path used by LINKER. The input library specified in this parameter will be used as the first library in the search path. Note that, if either INLIB or the LIB CU JCL statement is used, TEMP is not included in the search path unless it is specified in the LIB CU statement.

If no LIB CU JCL statement precedes the LINKER statement and no INLIB parameter is used the search path will be:

1. TEMP compile unit library.
2. SYS.HCULIB system compile unit library.

If the INLIB parameter is used but no LIB CU statement is used, the search path will be:

1. Library specified in INLIB parameter.
2. SYS.HCULIB.

If a LIB CU statement is used but no INLIB parameter is used, the search path will be:

1. Libraries specified in LIB CU statement.
2. SYS.HCULIB.

If a LIB CU statement and the INLIB parameter are both used, the search path will be:

1. Library specified in INLIB parameter.
2. Library specified in LIB CU statement.
3. SYS.HCULIB.

If both LIB CU and INLIB are used, only three libraries can be specified in the LIB CU statement. This is because the search path can contain only four user specified libraries in addition to the SYS.HCULIB, which is included at the end of every search path automatically. If a fourth library (INLIB4) is specified in the LIB CU statement, it will be ignored if the INLIB parameter is also used.

4.2.3 OUTLIB Parameter

The OUTLIB parameter specifies the library in which the load module is to be stored. An output-library-description or the keyword TEMP may be used in the OUTLIB parameter.

If a library is specified, it must have been allocated previously by the LIBALLOC LM utility (see the *Library Maintenance Reference Manual*) unless the SIZE parameter is used in the output-library-description of OUTLIB. If TEMP is specified, the load module will be written as a member of a temporary system library.

If the OUTLIB parameter is omitted, this is equivalent to OUTLIB=TEMP.

The load module is stored in a library according to the following rules:

- If a load module of the same name is not already present in the library, and there is no fatal LINKER error, the load module is stored in the library with the load-module-name given in the LINKER statement.
- If a load module with the same name (normally a former version of the load module) is in the library and there is no fatal linking error, the old load module is deleted and the new one replaces it. If there is a fatal error during the linkage no load module is stored; the old load module is still usable.

When an old version exists in the load module library, it is good practice to use a new load-module-name for storing the new load module to assure retaining the old and new versions together until the new one is proven executable. Once the new load module is debugged, the old version can be deleted and the new one renamed with the old name. Deletion and renaming are done using the LIBMAINT LM utility.

Alternatively, the user can maintain a "stable" and "development" library. The stable library should contain a working version of each program. The development library should contain the latest version of each program currently being developed and tested. Once successfully tested, programs can be moved from the development library to the stable library.

4.2.4 COMMAND and COMFILE Parameters

The COMMAND and COMFILE parameters allow the user to specify a set of commands to be obeyed by LINKER during the linkage process. The commands can be stored in a command file (COMFILE parameter) or can be specified directly (COMMAND parameter). The maximum length of a command string specified in the COMMAND parameter is 2500 characters.

The available commands are ENTRY, LIST, and VACSEG. The commands ENTRY, STACK3, SEGTAB1, and FILE are described briefly below as they are of special interest to the C programmer. For a full description of all commands, see the *LINKER User's Guide*. Commands must be separated by one or more spaces or by a comma and zero or more spaces. The final command may be followed by a semi-colon (;).

The COMMAND and COMFILE parameters can also be used to specify a series of load modules to be linked during a single execution of LINKER.

4.2.5 ENTRY Parameter

This parameter specifies the entry-name to be used as the start point for program execution. The compile unit containing this entry-name will be the first one used by LINKER in building the load module. It can be omitted if the entry-name is the same as the load-module-name.

As it is specified in the reference manual, all C programs must start with the function "main". If a source member contains this function "main", the entry point "main" is transformed into the entry point <source-member-name>. If the function main is in an input_enclosure, the entry point main is transformed into the entry point <input_enclosure_name>.

In the case of a source member, the entry parameter is:

```
ENTRY = <source-member-name>.
```

4.2.6 PRTFILE Parameter

This parameter requests that the LINKER listing be appended to a permanent SYSOUT file for printing or processing at a later stage by, for example, WRITER or any text handling program or utility. Otherwise, the listing is printed at the end of the job and no permanent copy is kept.

If the PRTFILE parameter is used, LINKER adds the listing to the SYSOUT file in append mode. The PRTLIB parameter, on the other hand, replaces any previous listing of the same name (see below). In either case, the LINKER listing will not be printed automatically. Printing can be requested later by using a WRITER JCL statement. Only the Job Occurrence Report will be printed at the end of job execution.

When serial linkage is requested and the PRTFILE parameter is used, all listings are stored in a single file.

4.2.7 PRTLIB Parameter

This parameter is similar to PRTFILE except that the listing will be stored in a member of the library specified in the PRTLIB parameter. If several programs are linked in series when the PRTLIB parameter is used, the listing for each program will be stored in a separate library member. Each library member will be given a name comprising the load-module-name suffixed by "_K". It replaces any member of the same name.

4.2.8 Linker Commands

4.2.8.1 ENTRY Command

The format of the ENTRY command is:

```
ENTRY = member-name
```

The entry command specifies the entry-name to be used as the start point for program execution. This command is used in the same way as the ENTRY parameter. When the COMMAND or COMFILE parameter is used in the LINKER statement, the ENTRY parameter cannot be used. The ENTRY command should be used instead.

4.2.8.2 STACK3 Command

The format of the STACK3 command, as used for an executable C program, is as follows:

```
STACK3= (  INITSIZE=m[K]
        [MAXSIZE=n[K]]

                  { YES }
        PAGING=   {     }    )
                  { NO  }
```

INITSIZE specifies the size of the initial page of the stack. The units are taken as bytes unless the suffix K is present, in which case the units will be in Kilobytes.

MAXSIZE specifies the maximum size of the stack for an executable program. An insufficient value of MAXSIZE leads to an R3STACKOV error message and an abort of the executable program.

PAGING specifies if the stack may be composed of several segments (YES) or must be restricted to only one segment (NO).

The default values are:

```
INITSIZE=2K,
MAXSIZE=16K,
PAGING=NO.
```

4.2.8.3 SEGTABi Command

The format or the SEGTAB command, as used for an executable C program, is as follows:

```
SEGTABi = (VSEG=n)
```

Where i can be 1, 2, or 3, indicating the segment table number, and n can be 0, 1, 2, or 3, indicating the number of table entries.

This command is required only when an executable program uses dynamic allocation that exceeds 64 Kbytes. This occurs if you receive an abnormal return code from the malloc function.

4.2.8.4 FILE Command

The format of the FILE command, as used for an executable C program, is as follows:

```
                   {  SEQ    }     [           { 1 }   ]
FILE = (FILEORG= { }         }     [ NBBUF =   { - }   ]
                   { DIRECT }     [           { 2 }   ]

[                    { 1   }   ]
[  NUMBER =          {     }   ]          )
[                    { nn }    ]
```

This command is used to reserve resources for one or several files when the number of files used simultaneously by the executable program exceeds 10. The value of nn is limited to 50, but you can have more than one FILE command.

4.2.9 Linker Output

The following paragraphs briefly describe the printer output produced by LINKER. For a more detailed description of the printer output see the *LINKER User's Guide*.

The LINKER listing is composed of the following sections.

- Banner page and LINKER commands listing. All commands included in the COMMAND parameter or command file of the JCL statement LINKER are listed in the LINKER commands listing.
- Included compile units (if any). Details are printed for each compile unit included in the load module as a result of using the INCLUDE command.
- Group information. This listing contains general information about the entire process group. The listing is in two parts: global segment list and segment list. The segment list is the most useful part of the LINKER listing for the programmer and is described in more detail below.
- Cross-reference listing (if any). The cross-reference listing is only produced if the LIST=XREF LINKER command is used. In this listing, for each external name, the location of each reference to the name is shown.
- Linkage report and end page. The linkage report gives a summary of the error messages generated by LINKER. This report is described below. The end page simply contains the percentage of the total library space used by all load modules currently in the library.

4.2.9.1 Segment List

The segment list, contains an entry for each segment in the load module (including global segments but excluding segments with H_ prefixed names). The segment list is the most useful part of the LINKER listing for the following reasons:

- The LINKER segment number and internal segment number are shown for each segment generated directly from user source code. The relationship between these segment numbers has to be known when tracing the origin of abnormal step terminations and in analyzing memory dump listings.
- The size of each segment in bytes is shown. This may be useful when estimating working set requirements for program execution.

The headings and information in the segment list which are of use to the C programmer are as follows.

SEG.#	LINKER segment number in the form stn.ste.
IN CU:	The name of the segment as it appears in the segment list of the C line location map.
TYPE	This indicates that the segment contains code (C..), data (.D.) or linkage information (..L). Combinations of these types are also possible (that is, /C.L/).
SIZE	This indicates the size of the segment, in bytes.
MAXSIZE	This indicates, in the case of a variable length segment, the maximum size of the segment, in bytes. Note that SIZE and MAXSIZE values are needed for working set calculations.

4.2.9.2 Linkage Report

The first line of the linkage report contains either "ERRORS DETECTED" or "NO ERRORS DETECTED". If no errors have been detected, the linkage report ends immediately after printing the line "OUTPUT MODULE PRODUCED ON LIBRARY library-name". However, if errors have been detected, a summary of errors is now printed.

The summary of errors comprises one or more of the following lines:

- `WARNINGS (SEV.1) : n`
- `ERRORS SEVERITY 2: n`
- `ERRORS SEVERITY 3: n`
- `ERRORS SEVERITY 4: n`

where "n" is the number of errors in each category. If there are any errors of severity 4 (fatal), an output load module will not be produced and the linkage report will end with the line "NO OUTPUT MODULE PRODUCED". If there are no errors of severity 4 the linkage report will end with the line "OUTPUT MODULE PRODUCED ON LIBRARY library-name".

The end page simply contains the percentage of the total library space used by all load modules currently present in the library.

4.2.9.3 Error Messages

Each error detected at linkage time saves at least one test execution of the user program. In order to detect as many errors and inconsistencies as possible, LINKER carries out checks on the interface between linked procedures. For example, the arguments of a calling and called procedure must be compatible in number and attributes; external data declared in different procedures must have consistent attributes.

When an error is detected, LINKER outputs a message at the point in the listing at which the error occurred. Error messages have one of the following formats:

```
**** WARNING nnnn           message-text

**** ERROR  nnnn SEVERITY s message-text
```

where "nnnn" is the message number, "s" is the severity and "message-text" is an explanation of the situation. Severity "s" may have a value of 2, 3, or 4. (Severity 1 corresponds to a WARNING). Severity 4 is fatal and no load module will be output. The total number of error messages of each severity is given in the linkage report.

4.2.9.4 An Example

```
********************************************************************************
********************************************************************************
****                            GCOS7                                       ****
****                                                                        ****
****                             L K                                        ****
****                                                                        ****
****              VERSION: 90.00 DATED: JUN 30,1986                         ****
****                                                                        ****
********************************************************************************
********************************17-2************************************************
********************************************************************************

ADDITIONAL INFO:      4    5

1  CODE (DEFAULT):      OBJC     OBJD
*****************************LINKER CONTROL STATEMENTS**************************
2  LIST=S,
3  STACK3=(INITSIZE=128K,MAXSIZE=256K) ,
*****************************************TASK=MAIN*******************************
4  PROCESS OCCURRENCES : PO      OBJC     OBJD
5  FATHER PROCESSES :    NONE
6                        BASE      1ST PAGE   NB.PAGES SH INITSIZE MAXSIZ
7  STACK RING 0           8.12     NONE         0      3        0    4096
8  STACK RING 1           8.13      8.14        5      3     2048   16384
9  STACK RING 2           8.1A     NONE         0      3     2048   16384
10 STACK RING 3           1. 0     NONE         0      3   131072  262144
11 ENTRY POINT = ACKER_HG         LOCATION:  8.10.000010     IN CU: ACKER_HG
12 ==================================GROUP INFORMATION============================
13 MINIMUM CONTROL MEMORY REQUIRED :   8176  MINIMUM USER   MEMORY REQED :  135632
14 FIXED SIZE SEGTS. CUMULATED SIZE:   9264  VAR SIZE SEGS CUM INIT SIZE :  135264
15 VAR SIZE SEGS CUMUL MAXIMAL SIZE: 524288  LOAD MODULE SIZE            :   20913
16                 CONTROL SEGMENTS
17                      SEG NUM                      SEG NUM
18 PGCR               9. 0      PCS                8. 0
19 NPCS               8. 1      ITS LIST           9. 2
20 TASK.DIR.          9. 3      DEBUGGING          9. 5
21 PG P0  3  1  0  W              208
22 PGFECB             9. 8      DECB               9. 9
23 SEMPH. POOL        9. C      SYMBMAP            9. B
24 TERMINATION        9. 4      ASL2               9. 1
25 ASL3               8. 3
26               GLOBAL SEGMENTS
27   SEGNAME     SEG NUM   CONTAINS
28   __REFTAB     9. A                    LOCATION                LOCATION
29               H_CLR_EPILOG    000004   H_CLR_EPROLOG              000011
30               H_CLR_EPRINTF   00001F
31                 SEGMENT  LIST
```

```
32 SEG.   IN CU.ISN       TYPE SH RF RD WR EX WP EP  G  S    SIZE MAXSIZE  CONT.P.
33     .....
34   8. 0  PCS             .D.  3  3  3  0  0  W              352          *
35   8. 1  NPCS            .D.  3  3  3  1  0  W               32          *
36   8. 3  ASL3            .D.  3  3  1  0  0  W               16   32768  *
37   8.10  ACKER_HG.0      C.L  3  3  3  3  3     E           400           0
38   8.11  ACKER_HG.1      .D.  3  3  3  3  0  W               64           0
39  .....
40   9. 0  PGCR            CD.  2  3  3  0  3  W  E          4640
41   9. 1  ASL2            .D.  2  3  1  0  0  W               80   32768
42   9. 2  ITS LIST        .D.  2  0  3  1  0  W
43   9. 3  TASK.DIR.       .D.  2  3  3  0  0  W               48
44   9. 4  TERMINATION     .D.  2  3  3  0  0  W         S     96
45   9. 5  DEBUGGING       .D.  2  3  3  1  0  W                0   32768
46   9. 6  PG PCP S        .D.  2  0  1  0  0  W  E             0   32768
47   9. 7  OPTION          .D.  2  2  3  3  0  W                0   32768
48   9. 8  PGFECB          .D.  2  3  1  0  0  W                0   32768
49   9. 9  DECB            .D.  2  3  3  1  0  W                0   32768
50   9. A  __REFTAB        .D.  2  3  3  0  0  W               48
51   9. B  SYMBMAP         .D.  2  3  3  1  0                  48
52   9. C  SEMPH. POOL     .D.  2  3  3  1  1  W         S    3328
53   .....
=======================================LIST OF CU (S)==============================
                  ACKER_HG INLIB CREATED 18:20:24 DEC 19, 1989
                    BY: C-LANG 30..22 CU OPTION:SCIENT EOD
*****************************************LINKAGE REPORT*****************************
NO ERRORS DETECTED
-------------------------
.  OUTPUT MODULE PRODUCED ON LIBRARY   ;000325.TEMP.LMLIB
MODULE IS OF CLASS  (CODE):          0

NUMBER OF ITEMS PROCESSED
-------------------------
  - COMPILE UNITS        1
  - SYMDEFS              1 ( PROC     1, DATA     0)
  - SYMREFS              4 ( PROC     1, DATA     3)
  -   CALLED SYSDEFS     0
  -   NB OF CALL '''     0
  - EXT. DATA NAMES      3
  - SEG.ENTRIES USED   514 (TYPE 2  255,TYPE 3  259)
     TYPE 2 VACANT     242
     TYPE 3 VACANT     228 IN        MAIN
                      LARGE  3 STN 1
*****************************************L*I*N*K*E*R*****************************
*****************************************END OF SESSION**************************
*********************************LAST PERCENTAGE OF SPACE USED      9
```

4.3 Interactive Operation in GCL Mode

LINKER is activated with the LINK_PG command. All parameters are the same as those of the LINKER command in BATCH mode. For more information, see the *IOF Terminal User's Reference Manual* and the *IOF Programmers Manual.*

EXAMPLE:

```
S: LINK_PG?

 1/2          LINK_PG                                  -->:

         link an executable module

LM            name of the executable module prog

SM            is executable module a TPR?        0

INLIB         input library (default is #CLIB)
c.culib

Commands may be read from a file (COMFILE)
or directly supplied (COMMAND), or read from the terminal
(default).
COMFILE       command file

COMMAND       immediate commands
list=e
```

❑

4.4 Separate Compilation

4.4.1 General Information

Separate compiling enables splitting a logical program into several compile units and putting these different compile units together to make up a unique executable program.

Declaratives and definitions of functions and data items enable linking these units with what are known as SYMREFS (symbolic references) and SYMDEFS (symbolic definitions) which are generated at compile time.

The links are resolved by the static LINKER which attempts to match the object definition (SYMDEF) with the object reference(s) (SYMREF).

4.4.2 Implementation

EXAMPLE:

Let P1 be a file containing.

```
1    #include  <STDIO_H>
2    int a=3;
3    main ()
4      { extern int SQUARE  ();
5       printf ("%d\n", SQUARE  (a));
6      }
```

and P2 be a second file containing

```
1    extern int a;
2    int SQUARE  (x) int x;
3    {   a = x + 1;
4       return (a * a);
5    }
```

❑

Compiling P1 produces a compile unit named P1, which contains 2 SYMDEFS: P1 and a. P1 (substituting main): is a procedure type, which is a PROC SYMDEF. The other one, a, is a data type, which is a DATA SYMDEF.

Compiling also produces two SYMREFs:

- SQUARE: it is a procedure type
- printf: it is the same type.

Compiling P2 produces another compile unit with the name P2 containing:

```
a PROC SYMDEF   : SQUARE

a DATA SYMREF   : a
```

These SYMDEFs can be observed with the command "list<cu_name>, alias;" from the LIBMAINT CU. Moreover, the SYMDEFs and SYMREFs previously mentioned in the DATA MAP part of the compile listing can be found again in that listing if the MAP option is active.

If the compile units from different programs are stored in several libraries, it may be necessary to use the JCL command LIB CU. Search rules are the following:

INLIB library
INLIB1 library
INLIB2 library
INLIB3 library

EXAMPLE:

```
C SOURCE=P1   CULIB=myculib1,   INLIB=temp;

C SOURCE=P2   CULIB=myculib2,   INLIB=temp;

LIB CU   INLIB1=myculib2;

LINKER   P1  INLIB=myculib1;
```

❑

Starting from the compile unit containing the entry point main automatically converted into P1, the LINKER resolves the references to external objects. If one of the still unresolved objects is of the procedure type and belongs to a different compile unit, the LINKER searches for the definition of this object in the compile unit libraries according to the order previously indicated. The LINKER analyzes the new compile unit as it did the previous one, and the process loops until all references are resolved.

Three types of errors can be detected during this kind of processing. The messages are supplied for information.

- CONFLICT BETWEEN REF/DEF ATTRIBUTES

  The definition of the object and that of the reference are not the same. For procedures or functions, the conflict bears on the type of the return value. No checks are performed on the number and/or the type of arguments.

  Note that this also applies to inter-language calls if the different language types are in conflict.

- UNRESOLVED REFERENCE or NO MATCHING DEF.

  No definition matching this reference can be found in the library path supplied (for example, INLIB and INLIB1). Manipulating this object causes an exception at execution time (FAULT DATA DESCRIPTOR or FAULT BASE REGISTER).

- THIS CATALOGED ENTRY ALREADY EXISTS.

  The same object is defined in several compile units.

All these errors are indicated in the linkage report. The LINKER then usually ends with a non null severity. A load module is still produced although errors can occur at execution if used in this condition.

4.4.3 Inter-Language Calling

4.4.3.1 Correspondence Between Data Types

It is possible to call sub-routines and to reference data items coded in all languages other than C supported by GCOS 7. The "correspondence table" below shows the correspondence between the base types for C and those of other languages.

Note the following comments:

- External data cannot be manipulated from other languages.
- Aggregates (arrays, structures and unions) are implemented in the same way as the corresponding objects of the external language, if they exist.
- Arrays are allocated in lines and then in columns. (This is the opposite of FORTRAN.)
- Any correspondence not included in the table indicates constructs that are either difficult or impossible to describe.
- Passing parameters by value is done in the same way as in PASCAL and GPL.

Correspondence Table:

The following is a table showing the correspondences between the C language and other GCOS 7 languages.

C	GPL	PASCAL	FORTRAN	COBOL
short int	FIXED BIN(15)		INTEGER*2	COMP-1
long int	FIXED BIN(31)	INTEGER	INTEGER*4	COMP-2
int	FIXED BIN(31)	INTEGER	INTEGER*4	COMP-2
char	LOGBIN(8) BYTE	CHAR	CHARACTER*1	
float	FLOAT BIN(21)		REAL	COMP-9
double	FLOAT BIN(53)	REAL	DOUBLE PRECISION	COMP-10
short unsigned	LOGBIN(16) BYTE			
unsigned	LOGBIN(32) BYTE			
long unsigned	LOGBIN(32) BYTE			
pointer	POINTER			
function	ENTRY RETURNS	FUNCTION	FUNCTION	

Aggregates can correspond with each other. The following rules help do this:

- Elements in a structure are allocated in the same order.
- Arrays are allocated first by lines, then by column. This is true in several languages, but not Fortran.

In STANDARD (or GCOS 7) mode, the type promotion is very important when passing parameters. For example, float data is promoted to double; integer, short, long, and character data is promoted to long.

However, in ANSI (or GANSI) mode, the argument is assigned the type of corresponding formal parameter (if any) only in the function prototype. Otherwise, the process is the same as in STANDARD.

The following are examples of aggregate correspondence in C and COBOL.

In C:

```
struct s {int a; double t(2);} s;
```

In COBOL:

```
01 S
02 A COMP-1
02 T COMP-10 OCCURS 2
```

4.4.3.2 Passing Parameters Between Languages

Passing by Reference

By default, the passing of parameters is done by value in C, not by address.

For example, the program below does not pass the object i to p. It passes only a copy of the object i, which contains the value 3.

```
i = 3;
p(i);
```

Any modification to the parameter in p has no influence on the argument i. When returning from p into the caller, the value of i is still 3, no matter what the code of p.

However, there are some procedures that need to modify a parameter. To do this, C passes a pointer to the object, rather than passing the object itself, as in the following example:

```
p (&i)
```

In this example, p is declared:

```
void p (*int)
```

This changes the interface slightly, because this passes a pointer to int, not the int itself.

Most languages other than C pass parameters by reference, which means that a modification to the called parameter affects the argument that the caller passes. (Some exceptions to this are non VAR parameters in PASCAL and arguments explicitly passed by value in GPL.)

The C language has an extension that can communicate with other languages. With this extension, the C language can declare or define a function that accepts a parameter passed by reference. This function is declared as follows:

```
extern a (&);
```

Only LEVEL=GCOS 7 and GANSI support this extension. This extension is very convenient when the called procedure is written in a language other than C, for example, COBOL, GPL, and FORTRAN.

NOTE:

The pragma BYREF is equivalent to this extension. It has the same semantics and is available on any level.

Another solution is to write a relay procedure, as follows:

```
CALLER    ()                              /* Calling in C */

{
char *point; char zone [13];
extern void CALLED ();

      (void) CALLED (&zone[0]);
}

CALLED: proc (point);                     /* Called in GPL */
DCL point PTR;
DCL obj CHAR(14)  BASED (point) NOMAP;

      obj = "EFFET DE BORD" !!"00"H;

END CALLED;
```

Special syntax:

The function for which all arguments are passed by reference is declared with the character "&" following the first parenthesis of the declarative.

4.4.3.3 Examples of Passing Parameters

This subsection contains several example programs showing parameter passing. These programs are in C, COBOL, and GPL. The following conditions apply to these examples:

- USAGE POINTER and SET ADDRESS are restricted to COBOL-85 LEVEL.
- The & feature is restricted to LEVEL GCOS 7 or GANSI, but the pragma BYREF can be used at any level.
- There are no functions in COBOL, so the C functions that are called or those that call COBOL programs are declared void.
- The minus (-) character is not allowed in C names. Uppercase and lowercase letters in names can lead to error.

EXAMPLE 1: C AND COBOL

In this example, a COBOL program calls a C procedure. The C procedure copies its first parameter into its second one. Both are displayed by the calling program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   CBLMAIN.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PPW      PIC X(10).
LINKAGE SECTION.
01 AA PIC X(10).
PROCEDURE DIVISION  USING AA.
PAR.
    MOVE "ABCDEFGHIJ" TO AA.
    CALL "CPROG" USING AA, PPW.
    DISPLAY AA, PPW UPON TERMINAL.
    STOP RUN.
```

```
void CPROG (& a,b) char a; char b; {
   int i;
   char *p1 = &a;
   char *p2 = &b;

   for (i=0; i<10; i++)
   *p2++=*p1++;
   }
```

❑

EXAMPLE 2: C AND COBOL

In this example, a C program calls a COBOL procedure (CLP4). This procedure in turn calls a C function (clp5). This is done in two programs. In this example, the programs first use the byref (&) feature of the GCOS 7 compiler, where the parameters are then passed by reference. This is followed by a corresponding COBOL program. The first program is as follows:

```
#include <stdio.h>

main ()
{
extern void clp4 (&);
struct s1 {
int   c1, c2, c3;
} ss;
int dd[2];
short tt;
float ff;
double bb;
char kk;
ss.c1 = 2;
ss.c2 = 3;
ss.c3 = 7;
dd[0] = 8;
dd[1] = 9;
tt = 11;
ff = 12;
bb = 13;
kk = 'K';
clp4 (ss, tt, ff, bb, kk, dd);
printf ("CLP: RES=%d\n", dd[1]);
}

void clp5 ( & ss, tt, ff, bb, kk, dd)
struct s3 {
int    c1, c2, c3;
} ss;
short tt;
float ff;
double bb;
char kk;
int dd[2];
{
dd[0] = ss.c3;
ff = tt;
bb = 13;
kk = 'K';
return;
}
```

The corresponding COBOL program writes as follows:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    CLP4.
 DATA DIVISION.
*
 WORKING-STORAGE SECTION.
*
 LINKAGE SECTION.
 01  SS.
      02  C1    COMP-2.
      02  C2    COMP-2.
      02  C3    COMP-2.
 01  TDD.
      02  DD    OCCURS 2   COMP-2.
 01  TT    COMP-1.
 01  FF    COMP-9.
 01  BB    COMP-10.
 01  KK    PIC X.
*
 PROCEDURE DIVISION    USING SS, TT, FF, BB, KK, TDD.
 PAR.
     MOVE 21 TO TT   MOVE 22 TO FF   MOVE 23 TO BB
     MOVE "H" TO KK.
     ADD C1 TO C2 GIVING C3.
     CALL "clp5"  USING SS, TT, FF, BB, KK, TDD
     MOVE FF TO DD (2).
     EXIT PROGRAM.
```

❑

EXAMPLE 3: C AND COBOL (ADVANCED)

This example is like example 2 in that it is also of a C program that calls a COBOL procedure (clp2), which in turn calls a C function (CLP3). In this example, however, it is more difficult to use the C parameter passing conventions and to adapt the COBOL program to deal with it.

```
#include <stdio.h>
main ()
{
externe void clp2 (&);
struct s1 {
int   c1, c2, c3;
} ss;
int dd[2];
short tt;
float ff;
double bb;
char kk;
int * aa;
ss.c1 = 2;
ss.c2 = 3;
ss.c3 = 7;
dd[0] = 8;
dd[1] = 9;
tt = 11;
ff = 12;
bb = 13;
kk = 'K';
aa = &dd[1];
clp2 (ss, tt, ff, bb, kk, dd, aa);
printf ("CLP: RES=%d\n", dd[1]);
}
void clp3 (ss, tt, ff, bb, kk, dd)
struct s2 {
int    c1, c2, c3;
} ss;
short tt;
float ff;
double bb;
char kk;
int dd[2];
{
dd[0] = ss.c3;
ff = tt;
bb = 13;
kk = 'K';
return;
}
```

The corresponding COBOL program is as follows.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    CLP2.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  PPW    USAGE IS POINTER.
 01  TT     COMP-1.
 01  FF     COMP-9.
 01  TDD.
  02 OCCURS 2 COMP-2.
*
 LINKAGE SECTION.
 01  SS.
  02  C1    COMP-2.
  02  C2    COMP-2.
  02  C3    COMP-2.
 01  TTX   COMP-2.
*   Promotion of short to long
 01  FFX   COMP-10.
*    Promotion of float to double
 01  BB    COMP-10.
 01  KKX.
  02  FILLER  PIC XXX.
  02  KK      PIC X.
 01  PP    USAGE POINTER.
*     PP represents the array which is passed as a pointer.
 01  AA       USAGE IS POINTER.
*
 PROCEDURE DIVISION    USING SS, TTX, FFX, BB, KKX, PP, AA.
 PAR.
     MOVE TTX TO TT.
     MOVE FFX TO FF.
     SET ADDRESS OF TDD TO PP.
     ADD C1 TO C2 GIVING C3.
     MOVE DD(1) TO TT.
     SET PPW TO ADDRESS OF TDD.
     MOVE TT TO TTX.
     MOVE FF TO FFX.
     CALL "clp3"  USING BY CONTENT SS, TTX, FFX, BB, PPW.
     MOVE FF TO DD (2).
     EXIT PROGRAM.
```

❑

EXAMPLE 4: C AND GPL:

In these examples, C is the caller.

- Passing parameter by value. The value of i remains 1 after the execution of Q.

```
extern Q();
main () { int i;        with:     Q:PROC(B);
                                  DCL B FIXED BIN(31);
         i=1;
         Q(i);}                   B=100;
                                  END Q;
```

- Passing parameter by reference. The value of i is 100 after the execution of Q.

```
extern Q(&); /* special syntactic feature available only
                at LEVEL=GCOS 7*/

main () { int i;                  Q:PROC (B);
         i=1;         with:       DCL B FIXED BIN(31);
        Q(i);}                    B=100;
                                  END Q;
```

- Passing address by value. The value of i is 100 after the execution of Q.

```
extern Q();
main () { int i;        with:     Q:PROC(P);
                                  DCL P PTR;
                                  DCL B FIXED BIN(31) BASED;
        i=1
          Q(&i);/*address of i*/
                                  P -> B =100;
        }                         END Q;
```

❑

EXAMPLE 5: C AND GPL

In this example, C is called. The value of i is 100 after the execution of F.

```
Q:PROC;                                  F(&b) int b;
DCL I FIXED BIN(31);       with:
DCL F ENTRY (FIXED BIN(31));               {b=100;}
       I=1;
       CALL F(I);
       END Q;
```

❑

4.4.3.4 Restrictions on Use

If the starting entry point of a multi-language application is not the C main function, the C Run Time Package must be initialized before any use of C Run Time primitives. Refer to Chapter 10.3.

4.5 Multitasking

4.5.1 What is Multitasking?

Multitasking is the process that performs several tasks at the same time. A task is a sequence of instructions that can be executed asynchroneously with respect to another task. Generally, a user program executes a single task, and this is a monotask job.

There are facilities to build a multitask job written in C language. There is a set of primitives that start or stop a task or synchronize tasks. The user concepts of task and job are directly mapped in the hardware by process and job respectively.

For more information, see the *System Overview*.

4.5.2 Building a Multitask Application

When building a multitask program, there are three main points to consider:

- Splitting the tasks
- Sharing the memory and file data
- Synchronizing the tasks

4.5.2.1 Splitting Tasks

The GCOS 7 features require that the task splitting is static. This means that when splitting tasks, each task must have a defined entry point specifying the procedure that is called at task initiation. The static LINKER then builds the different tasks. Each task is made of compile units that the corresponding entry point calls, unless specific liner commands are used.

The static hierarchy of tasks is limited to one level. This is a defined main task and a set of secondary tasks. Because only the main task can start secondary tasks, the dynamic hierarchy is the same as the static hierarchy. A secondary task cannot start another secondary task.

A secondary task can be started more than once and can have several running occurrences at a same time. The maximum number of occurrences possible at any one time is defined at link time.

The main task can share only statically-defined tasks. Each task must contain a main function or initialize properly the C Run Time package if any C Run Time primitive is used.

4.5.2.2 Passing and Sharing Data

There must be a means of communication between the tasks of a multitask application. This can be established in the following ways:

- Passing information
- Sharing files
- Sharing data

Passing Information

The C language can pass information in two ways. One way is to pass parameters between tasks. The begin_c_task function passes information from the main task to the new starting task at initiation. This function is recommended over the corresponding C primitive h_begtsk.

The other way to pass information is to send messages attached to a semaphore. This is discussed later in this subsection.

Sharing Files

There are two ways that tasks can share a file. However, for either way, there are no implicit file descriptors that are inherent from the main task to a starting secondary task, except for standard files such as: stdin, stdout, and stderr. The ways to share a file are as follows.

- The tasks share the same file description. The tasks pass the file description as a parameter of the new starting task or share it as part of the data. The file description is the returned pointer to a FILE structure, as described in the fopen function. For more information, see the description of sharing data, below.
- Each task opens the file independently. The SHARE and ACCESS parameters open the file in the tasks that require it. SHARE and ACCESS are described in the fopen function.

Sharing Data

The most efficient means of communication between tasks is sharing memory. For sharing data, there are two kinds of memory: type 2 and type 3. All the processes of the same process group share type 2 memory, but other process groups or jobs cannot. Type 3 memory is private for one process.

Depending on its storage class, the compiler, the LINKER, or the run-time package allocates a block of memory into a segment. For type 2 segments, which are process-group shared, the same segment name in several processes leads to the same memory block. For type 3 segments, the same segment name in different processes leads to different memory areas.

Storage classes include, for example, static, automatic, and dynamic. Each storage class has a default sharing level, which the user can change. Automatic data goes into the stack segment, which has a default level of 3. Static data goes into a static segment, which also has a default level of 3. The static LINKER allocates the static segment. The code goes into a non-writable static segment, which also has a default level of 3.

Dynamic data (that malloc creates) goes into a segment that the run-time package dynamically allocates. Its default sharing is 2, unless no type 2 entry is available, in which case type 3 segments are allocated.

The LINKER commands can change the default sharing level for non-dynamic data. Non-dynamic data includes STACK3, MSEGAT and DSEGAT commands. For dynamic data, allocation primitives can control the sharing level if the default sharing strategy is not correct. For more information, see the *C Language Primitives Reference Manual*.

NOTES:

1. Objects larger than 64Kbytes are allocated in large segments. The SEGTABi command declares these segments at link time with a specific sharing level.
2. When the malloc function allocates memory, it cannot be reallocated or freed in another task.
3. Using a pointer in a process when it is already allocated elsewhere in shared memory and assigned the address of a private area to a process can lead to undefined behavior.

4.5.2.3 Synchronization with Semaphores

A semaphore is a system object with two operations, and it can synchronize tasks under GCOS 7. A semaphore can be with or without a message.

For example, a task can perform a V-operation that resumes the execution of another task, which in turn executes a P-operation.

4.5.3 Differences between GCOS 7 and Unix

This subsection describes some of the differences between GCOS 7 and Unix.

4.5.3.1 Static and Dynamic Hierarchies

Under GCOS 7, the description of the hierarchy is more restrictive. It is a one-level hierarchy, that is a main task starting one or several secondary tasks, and it shall be statically defined. Also, the static and dynamic hierarchies of tasks must be the same.

Under Unix, creating a task is easier with the fork exec mechanism.

4.5.3.2 Sharing Data

Unix has a mechanism of implicit inheritance of data, whereas GCOS 7 provides a data sharing mechanism.

4.5.4 Run time Functions and Primitives

4.5.4.1 begin_task_h header

The begin_c_task function includes this file. It contains the definition of the task_param_t structure type, which the begin_c_task function also uses.

Synopsis

```
#include <begin_task_h>

int begin_c_task(char *task_name, short int occur, task_param_t *param);
```

Description

The begin_c_task function activates a secondary task in a multiprocess application. The pointer task_name points to the name of the starting task. If there are several task occurrences in the application, the occur parameter gives the occurrence number, beginning with 0. The pointer param points to a parameter structure that has the following description and meaning:

```
typdef struct {int size_param;
    char list_param[MAX_TASK_PARAM];
    } task_param_t;
```

In this structure, size_param gives the size of the parameter area that is passed to the launched task. Also, list_param is the parameter area itself. list_param is limited to 10280 bytes.

The C language can describe the starting entry point of the secondary task as follows:

```
main (argc, argv)
```

In this case, the standard semantics apply to the parameters argc and argv. That is, the value assigned to the size_param must correspond to argc, and the value assigned to list_param must correspond to the value of argv. Because argv is a pointer, list_param must be declared as an array of four characters.

Diagnostics

When successful, the begin_c_task function returns 0. Otherwise, it returns 1. An incorrect task name or occurrence number are common causes of failure.

4.5.4.2 Functions

This is a list of Run time functions that deal with multitasking.

begin_c_task	This function begins the execution of a secondary task. For more information, see the subsection directly above.
abort, exit	This function terminates execution of a task. The execution of the return statement of the main function also terminates the task.
argc,argv	Main arguments are interpreted as argc and argv when there are two defined arguments.
atexit	This gives a set of functions that are executed before task termination.
malloc/realloc /calloc/free	These allocate or deallocate memory.
clock	This function gives the time of the task.

4.5.4.3 Primitives

The C language primitives can be useful in multitasking. The following is a list of some of these primitives with a short explanation of each. For more information, see the *C Language Primitives Reference Manual*, especially the section describing task management.

h_begtsk	Starts the execution of a secondary task.
h_abtsk	Stops the execution of a task.
h_testsk	Tests the execution of a secondary task.
h_waitsk	Waits for the termination of a secondary task.
h_crsempool	Creates a semaphore pool
h_dlsempool	Deletes a semaphore pool
h_getsem	Gets a semaphore
h_freesem	Releases a semaphore
h_sep	Executes a P-op on a semaphore
h_sepm	Executes a P-op on a semaphore with message
h_sept	Executes a P-test on a semaphore
h_septm	Executes a P-test on a semaphore with a message
h_sev	Executes a V-op on a semaphore
h_sevf	Executes a V-op on a semaphore with message enqueued FIFO
h_sevl	Executes a V-op on a semaphore with message enqueued LIFO
h_sevtf	Executes a V-test on a semaphore with a message enqueued FIFO
h_sevtl	Executes a V-test on a semaphore with a message enqueued LIFO

4.5.5 LINKER Commands

This subsection describes some of the LINKER commands that deal with multitasking. They are as follows:

ENTRY
TASK
DSEGAT
MSEGAT
SEGTABi
SEMPOOL
STACK3

For more information, see the *Linker User's Guide*.

```
ENTRY=entry_point_name
```

- This command defines the entry point of the main task.

  ```
  TASK=(task_name [, OCNB=p] , START=entry_name)
  ```

- This command defines the secondary tasks. task_name is the starting task's name, which the begin_c_task function gives. p is the maximum number of occurrences of the task that can be started at the same time, and the default is 1. entry_name is the entry point for the task.

  ```
  DSEGAT=({CODESEG|INSTATIC|SEMSEG} [SHRLEVEL=t])
  ```

- The command changes the default sharing level for statically allocated segments.

  ```
  MSEGAT=({cu_name|seg_id|GLOBLSEG|segname} [SHRLEVEL=t])
  ```

- This commands modifies the sharing level of a specific segment.

  ```
  SEGTABi=(SHRLEVEL=t,VSEG=n)
  ```

- This command allocates large objects with a specific sharing level.

  ```
  SEMPOOL
  ```

- This command changes default size of the semaphore pool.

  ```
  STACK3=(TASK={MAIN|task_name} [SHRLEVEL=t])
  ```

- This command changes the default sharing level for a task. The default is 3.

4.5.6 Restrictions

A file can be connected to a terminal only in the main task, if not, the multitasking process issues an abnormal return code. Also, the memory allocated in any given task can be then reallocated or released only in that same task.

4.5.7 Example of a Multitask Program

The following example is the C language adaptation of a multitask program. This program is also described in the *C Language System Primitives Reference Manual*.

The job consists of a main task MAIN_TASK_C, and three secondary tasks FIRST_TASK_C, SECOND_TASK_C, and THIRD_TASK_C. The **sem** semaphore synchronizes the tasks. The semaphore is without message. The value of **sem** is initially set to 0, and its maximum value is 2. Because the values of sem are shared data, sem is declared in a dedicated compile unit (DATA_C) which LINKER allocates in a type 2 segment.

```
MAIN_TASK_C
  10 #include <retcode.h>
  20 #include <jobm.h>
  30 #include <timer.h>
  40 #include <taskm.h>
  50 extern char *sem;            /* pointer to semaphore,      */
  60                             /* shared by all tasks        */
  70 main ()
  80 {
  90    char *message;           /* message into JOR           */
 100    char *sempool;           /* pointer to semaphore pool  */
 110    int wait_time=30000;     /* waiting time used by       */
 120                             /* h_setelt_milsec function   */
 130
 140    /* create a type 2 private semaphore pool,             */
 150    /* it contains one semaphore without message           */
 160    h_crsempool(sempool,0,1,128);
 170    if (!h_testrc(DONE))
 180    {  message="Unable to create semaphore pool";
 190       { h_putjor(message,strlen(message)); }
 200       abort();              /* abort program              */
 210    }
 220
 230    /* get a semaphore without message from pool,          */
 240    /* initial count=0, maximum count=2                    */
 250    { h_getsem(sem,sempool,0,2,SN); }
 260    if (!h_testrc(DONE))
 270    {  message="Unable to get semaphore";
 280       { h_putjor(message,strlen(message)); }
 290       abort();              /* abort program              */
 300    }
 310
 320    /* start "FIRST_TASK_C" to issue a P-operation         */
 330    /* upon semaphore "sem". "FIRST_TASK_C" waits upon     */
 340    /* "sem" whose count is -1                             */
 350    { h_begtsk("FIRST_TASK_C",0,NULL_PTR); }
 360    if (!h_testrc(DONE))
```

```
370    {  message="Unable to start FIRST_TASK_C";
380       { h_putjor(message,strlen(message)); }
390       abort();               /* abort program              */
400    }
410
420    /* start "SECOND_TASK_C" to issue a P-operation         */
430    /* upon semaphore "sem". "SECOND_TASK_C" waits upon     */
440    /* "sem" whose count is -2                              */
450    { h_begtsk("SECOND_TASK_C",0,NULL_PTR); }
460    if (!h_testrc(DONE))
470    {  message="Unable to start SECOND_TASK_C";
480       { h_putjor(message,strlen(message)); }
490       abort();               /* abort program              */
500    }
510
520    /* wait "wait_time" micro-seconds elapse time :         */
530    /* "SECOND_TASK_C" is started after this delay          */
540    h_setelt_milsec(wait_time);
550
560    /* test status of "FIRST_TASK_C" and "SECOND_TASK_C",   */
570    /* should be not yet terminated                         */
580    { h_testsk("FIRST_TASK_C",0); }
590    if (h_testrc(NOTYET))
600    {  message="FIRST_TASK_C not yet terminated";
610       { h_putjor(message,strlen(message)); }
620    }
630    else
640    {  message="FIRST_TASK_C termination error";
650       { h_putjor(message,strlen(message)); }
660       abort();               /* abort program              */
670    }
680
690    { h_testsk("SECOND_TASK_C",0); }
700    if (h_testrc(NOTYET))
710    {  message="SECOND_TASK_C not yet terminated";
720       { h_putjor(message,strlen(message)); }
730    }
740    else
750    {  message="SECOND_TASK_C termination error";
760       { h_putjor(message,strlen(message)); }
770       abort();               /* abort program              */
780    }
790
800    /* start "THIRD_TASK_C" to issue two successive         */
810    /* V-operations upon "sem". "FIRST_TASK_C" and          */
820    /* "SECOND_TASK_C" resume their execution               */
830    /* Count of "sem" is 0. "THIRD_TASK_C" issues then a    */
840    /* P-operation upon "sem": so this task waits upon      */
850    /* "sem" whose count is -1                              */
860    { h_begtsk("THIRD_TASK_C",0,NULL_PTR); }
870    if (!h_testrc(DONE))
```

```
 880    {  message="Unable to start THIRD_TASK_C";
 890       { h_putjor(message,strlen(message)); }
 900       abort();              /* abort program              */
 910    }
 920
 930    h_setelt_milsec(wait_time);
 940
 950    /* test status of "FIRST_TASK_C" and "SECOND_TASK_C",   */
 960    /* should be terminated                                 */
 970    { h_testsk("FIRST_TASK_C",0); }
 980    if (h_testrc(DONE))
 990    {  message="FIRST_TASK_C terminated";
1000       { h_putjor(message,strlen(message)); }
1010    }
1020    else
1030    {  message="FIRST_TASK_C termination error";
1040       { h_putjor(message,strlen(message)); }
1050       abort();              /* abort program              */
1060    }
1070
1080    { h_testsk("SECOND_TASK_C",0); }
1090    if (h_testrc(DONE))
1100    {  message="SECOND_TASK_C terminated";
1110       { h_putjor(message,strlen(message)); }
1120    }
1130    else
1140    {  message="SECOND_TASK_C termination error";
1150       { h_putjor(message,strlen(message)); }
1160       abort();              /* abort program              */
1170    }
1180
1190    /* test status of "THIRD_TASK_C":                       */
1200    /* should be not yet terminated                         */
1210    { h_testsk("THIRD_TASK_C",0); }
1220    if (h_testrc(NOTYET))
1230    {  message="THIRD_TASK_C active";
1240       { h_putjor(message,strlen(message)); }
1250    }
1260    else
1270    {  message="THIRD_TASK_C termination error";
1280       { h_putjor(message,strlen(message)); }
1290       abort();              /* abort program              */
1300    }
1310
1320    /* issue a V-operation on "sem": "THIRD_TASK_C" resumes */
1330    /* execution. Count of "sem" is now 0.                  */
1340    /* Wait until "THIRD_TASK_C" completion.                */
1350    h_sev(sem);
1360    { h_waitsk("THIRD_TASK_C",0); }
1370    if (h_testrc(DONE))
1380    {  message="THIRD_TASK_C terminated";
```

```
1390       { h_putjor(message,strlen(message)); }
1400    }
1410    else
1420    {  message="THIRD_TASK_C termination error";
1430       { h_putjor(message,strlen(message)); }
1440       abort();               /* abort program             */
1450    }
1460 }
```

FIRST_TASK_C

```
  10 #include <jobm.h>
  20 #include <taskm.h>
  30 extern char *sem;
  40 main ()
  50 {
  60    { h_putjor ("FIRST ==> BEGIN",15); }
  70    h_sep(sem);
  80    { h_putjor ("FIRST ==> END",13); }
  90 }
```

SECOND_TASK_C

```
  10 #include <jobm.h>
  20 #include <taskm.h>
  30 extern char *sem;
  40 main ()
  50 {
  60    { h_putjor ("SECOND ==> BEGIN",16); }
  70    h_sep(sem);
  80    { h_putjor ("SECOND ==> END",14); }
  90 }
```

THIRD_TASK_C

```
  10 #include <jobm.h>
  20 #include <taskm.h>
  30 extern char *sem;
  40 main ()
  50 {
  60    { h_putjor ("THIRD ==> BEGIN",15); }
  70    { h_putjor ("THIRD ==> issues a V-op",23); }
  80    h_sev(sem);
  90    { h_putjor ("THIRD ==> issues a V-op",23); }
 100    h_sev(sem);
 110    { h_putjor ("THIRD ==> issues a P-op",23); }
 120    h_sep(sem);
 130    { h_putjor ("THIRD ==> END",13); }
 140 }
```

DATA_C

```
  10 char *sem;
```

The following GCL statements are for compilation and linkage of the program:

```
CLANG MAIN_TASK_C   INLIB=.SLLIB CULIB=.CULIB EXPLIST
                    PRTLIB=.LISLIB LEVEL=GCOS7 XREF;
CLANG FIRST_TASK_C  INLIB=.SLLIB CULIB=.CULIB EXPLIST
                    PRTLIB=.LISLIB LEVEL=GCOS7 XREF;
CLANG SECOND_TASK_C INLIB=.SLLIB CULIB=.CULIB EXPLIST
                    PRTLIB=.LISLIB LEVEL=GCOS7 XREF;
CLANG THIRD_TASK_C  INLIB=.SLLIB CULIB=.CULIB EXPLIST
                    PRTLIB=.LISLIB LEVEL=GCOS7 XREF;
CLANG DATA_C INLIB=.SLLIB CULIB=.CULIB EXPLIST
                    PRTLIB=.LISLIB LEVEL=GCOS7 XREF MAP;
LINK_PG LM=MULTITASK_C INLIB=.CULIB LIB=.LMLIB PRTLIB=.LISLIB
             COMMAND=#CAT('ENTRY=MAIN_TASK_C,'
                          'TASK=(FIRST_TASK_C START=FIRST_TASK_C),'
                          'TASK=(SECOND_TASK_C START=SECOND_TASK_C),'
                          'TASK=(THIRD_TASK_C START=THIRD_TASK_C),'
                          'MSEGAT=(DATA_C,1,SHRLEVEL=2);');
```

The following messages are written in the JOR during the program execution.

```
          LOAD MODULE = MULTITASK_C (16:57 SEP 14, 1988 )
          LIBRARY = FUEL.LMLIB
17:14:47  STEP STARTED XPRTY=8
          CLR00: C Run Time version 20.00    17   -1
          FIRST ==> BEGIN
          SECOND ==> BEGIN
          FIRST_TASK_C not yet terminated
          SECOND_TASK_C not yet terminated
          THIRD ==> BEGIN
          THIRD ==> issues a V-op
          THIRD ==> issues a V-op
          THIRD ==> issues a P-op
          FIRST ==> END
          SECOND ==> END
          TASK FIRST_TASK_C J=0E P=01 COMPLETED
          TASK SECOND_TASK_C J=0E P=02 COMPLETED
          FIRST_TASK_C terminated
          SECOND_TASK_C terminated
          THIRD_TASK_C active
          THIRD ==> END
          TASK THIRD_TASK_C J=0E P=03 COMPLETED
          THIRD_TASK_C terminated
          TASK MAIN J=0E P=00 COMPLETED
```

5. Execution and Debugging

This section describes how executable programs can be executed in batch and interactive mode. The debugging facilities of DPS 7 C are also described.

5.1 Step Execution

An executable program in the load module format is built by the linker. The execution of the load module is accomplished by execution of the associated STEP JCL statement, a full description of which is given in the *JCL Reference Manual*. The following is an example of the use of the STEP statement:

```
STEP MYPROG, MY.LIBRARY
    ,CPTIME =  10000
    ,LINES =  20000
    ,DEBUG =   (A.LIBRARY, SUBFILE=DEBPROG)
    ,OPTIONS =   'CASE1'
    ,REPEAT;
```

The only mandatory parameters are MYPROG and its library description. The other parameters appearing in the example are explained as follows:

MYPROG	Is the load module name, contained in the library member MYPROG.
CPTIME	Limits the use of the CPU time in units of one-thousandth of a minute. Here 10 minutes.
LINES	Limits the number of records written on the SYSOUT file, that is printed lines. Here 20000 lines.
DEBUG	Specifies that step execution is under control of the Program Checkout Facility, and that its commands are on the library member DEBPROG.
OPTIONS	Specifies a character string CASE1, accessible from the executable program by the use of ARGC ARGV. ARGC yields 2 and ARGV[1] points to "CASE 1".
REPEAT	Specifies that the step is to be restarted after a system crash.

5.2 Execution in Batch Mode

Execution in batch mode is illustrated in the following example, where a job is set up (compiled and linked) in interactive mode and then submitted for batch execution.

The program P is assumed to be the one created in the preceding section.

EXAMPLE:

```
S:    LMN SL,LIB=(C.SLLIB,DVC=MS/D500,MD=K104);
C:    EDIT;
R:    A
I:    $JOB JCLP,REPEAT;
I:    C SOURCE=P,INLIB=(C.SLLIB,DVC=MS/D500,MD=K104);
I:    LINKER P;
I:    STEP P,TEMP,CPTIME=5000,LINES=1000;
I:    ENDSTEP;
I:    $ENDJOB;
I: /
R:    Z(JCL) JCLP
R: /
C: /
S:    EJR JCLP LIB=Z2.SLLIB:K104:MS/D500;
```

❑

Note that for batch execution, it is good programming practice to put a limit on the CPU time and the number of printed lines. This will force termination of the program if it gets caught up in an endless loop.

5.3 Interactive Execution in GCL Mode

Execution of a C program is done with the EXEC_PG command. For more information, see the IOF Terminal User's Reference Manual.

EXAMPLE:

```
S: EXEC_PG?;
1/9                     EXEC_PG                                 -->:
execute a user program
PG                      + name of program to be executed

LIB                     program library (none is #LLIB or TEMP)
c.lmlib

LINES                   maximum number of printout lines        1000
CPTIME                  maximum CPU usage time                   500
ELAPTIME                maximum permitted clock-time            2000
DEBUG                   PCF input file (TN is terminal)

REPEAT                  allow checkpoints ?
DUMP                    NO, DATA, ALL
SIZE                    program working set
OPTIONS                 program option string
a b c


+++
2/9                     EXEC_PG                                 -->:

execute a user program

FILE1                   internal file name                  my_file1
ASG1                    file assign parameters
c.buglib
ALC1                    file allocation parameters
DEF1                    define file parameters
OUT1                    output parameters
```

❑

5.4 External Interface

An external interface is established through the option string which is transmitted to 'main' in the C program by the two parameters ARGC and ARGV. ARGC gives the number of literal sequences separated by blanks in the option string incremented by 1. ARGV is an array containing a pointer to a value for the character strings (ARGV [0] contains the name of the program).

```
EXEC_PG   TT OPTIONS='   X Y2T    U ';
yields:  ARGC=4        ARGV[0]="TT", ARGV[1]="X", ARGV[2]="Y2T",
ARGV[3]="U".
```

NOTE:
Each option is a C string ending with 0.

5.5 Batch or Interactive Debugging

The Program Checkout Facility is a system resident facility for debugging executable programs. It is especially useful in interactive mode , as you can start execution, stop at any line modify variables and so on without compiling the program again.

The principal PCF commands are:

- CHANGE to assign a new value to a variable or array element.
- DUMP to print the values of variables or array elements.
- GOTO to start (or restart) execution at some source line.
- PAUSE to stop interactive execution at a specified source line, if a certain condition is verified.
- TRACE to print the labels (or subroutine names) of lines which have been executed.

The operands of all these commands can be designated by the name of subroutines and their source lines. You can designate symbols by their source names as opposed to their physical addresses.

Restrictions: References to identifiers written in lower-case must be between simple quotes. Character strings are printed as binary bytes.

EXAMPLE:

```
1 main (){
2 int a,b,c;
3     a = 3;
4     b = a * 4;
5 l:  c = a + b;
6}


S: EXEC_PG P C.LMLIB DEBUG=TN;

  (((PCF AT ESSC LINE 1 ILN 1 IN P
  ...PCF AT BEGINNING OF MAIN PROCEDURE.
  ...100 D: p at line 3;
  ...110 D: go;
  )))
  (((PCF AT * LINE 3 ILN 3 IN P
  ...100 PAUSE
  ...110 D: d a,b,c;
  ...        a         135528472
  ...        b         135266312
  ...        c         135528680
  ...110 D: c a=10;
  ...        a         10          (135528472)
  ...110 D: p at 1
  ...120 D: go;
  )))
  (((PCF AT 1 LINE 5 ILN 5 IN P
  ...110 PAUSE
  ...120 D: d a,b,c;
  ...        a         3
  ...        b         12
  ...        c         135528680
  ...120 D: go;
  )))
S:
```

❑

5.6 Errors at Execution Time

At execution, errors are detected either inside the program or in the load module.

Errors detected in the load module are written in plain language in the JOR or at the console if executed interactively. The first correspond to an exception.

5.6.1 Errors Inside a Program

Errors inside a program have the following form:

```
FATAL   EX01. EXCEPTION xx.yy: <exception type> IN TASK
        MAIN AT ADDRESS <address>
WARNING
```

Where:

`<address>`

corresponds to the segmented address where the error was detected under the form: stn.ste.sra.

The pair stn.ste enables finding the corresponding segment from the listing produced by linker. The move in the segment is supplied by the sra value. The corresponding source line can be found using the compile listing in the Correspondence Table DATA MAP, LOC:LINE (if the MAP option was requested at compile time).

The main values that can be found for xx.yy. are the following:

06.00	ACCESS OUT OF SEGMENT BOUNDS	incorrect pointer, incorrect array subscript without check
0B.00	R3 STACK OVERFLOW	insufficient stack size, see Section IV
0E.01	FAULT DATA DESCRIPTOR	pointer at NIL external object
0E.02	FAULT BASE REGISTER	not found at linkage
10.00	FLOATING POINT OVERFLOW	errors in floating point arithmetic (real numbers)
10.01	FLOATING POINT UNDERFLOW	
10.02	FLOATING POINT DIVIDE	
11.00	FIXED POINT OVERFLOW	errors in fixed point arithmetic (integers)
11.01	FIXED POINT DIVIDE	
11.02	SUBSCRIPT OUT OF ARRAY RANGE	subscript outside authorized range of adjustable array

Arithmetic exceptions are warnings. The other errors are fatal, meaning that the corresponding step is aborted.

5.6.2 Errors in Load Module at Execution Time

Errors in the RTP (Run Time Package).

All messages returned by the RTP have the following form:

```
CLR<internal number>:<message> [<return code>] opt
```

The complete list of messages is supplied in the following paragraph. The internal error number is insignificant for the user.

Since using C means that it is the user who manages proper or improper functioning of the run-time primitive called by transmitting his own error message, the printing of standard messages can be invalidated through the two macros defined in STDIO_H: set_silent_mode () and cancel_silent_mode (). Otherwise, standard messages are printed by default.

However, any error occurring when the standard files stdin, stdout and stderr are opened produces a message from the RTP since in this particular case the current task is aborted.

The return code is an error code returned by the system.

A message can also be printed with a return code of the 'DONE' type which means that the error indicated does not necessarily cause an erroneous return of the run-time function that was called.

NOTES:

1. A run-time exception may be due to a user error in the employment of some function or other (faulty passed pointer, loss of information necessary to the RTP for memory management, etc). For performance reasons, the RTP does not always check the validity or the number of passed arguments, so it goes without saying that such exceptions may occur.
2. When the C main function is executed, the C Run-Time package checks that no more than 16K stack memory (i.e. auto variables) are requested. This restriction is due to the on-condition mechanism, triggered when an exception occurs. If this limit is reached, an error MAIN FUNCTION'S STACK IS TOO BIG is emmitted. In such a case, the user must rename his main function, for example into:
   main2 ()
   and write a new main function that does nothing but call main2:
   main() { main2(); }

5.7 Run-Time Errors

The following messages are intended to be self-explanatory. If you cannot determine what action to follow, please contact your Service Center.

```
ABNORMAL CONNECTION (CRFD).
ABNORMAL FFLUSH (CLOSE).
ASSIGNATION TO SYSOUT FAILED.
BUFFER VARIABLE ALLOCATION FAILED.
CANNOT ALLOCATE A LARGE OBJECT, USE LK CMD:SEGTAB1=(SHRLEVEL=2,VSEG=N).
CANNOT BE CONNECTED: INCORRECT FILE LITERAL.
CANNOT READ FILE DEFINITION OF H_PR.
CLOSE FAILED (CLOSE).
DOUBLE HAS BEEN TRUNCATED (ETOF).
DYNAMIC ALLOCATION UNSUCCESSFUL (CANNOT CREATE SEGMENT).
DYNAMIC VARIABLE SIZE TOO LARGE: IT CANNOT BE ALLOCATED IN A SEGMENT.
EFN NOT AVAILABLE VOLUME IS NOT ALREADY MOUNTED.
FAPPEND MAY NOT BE USED ON A BFAS DIRECT FILE.
FILE IS ALREADY CONNECTED, PREVIOUS CONNECTION PRESERVED.
FILE LITERAL ERROR: REMOTE FILE NAME > 255 OR = 0 AT INDEX <XX>.
FILE LITERAL ERROR: INPUT ENCLOSURE NAME LONGER THAN 16 AT INDEX <XX>.
FILE LITERAL ERROR:   UNKNOWN OPTION AT INDEX <XX>.
FILE LITERAL ERROR:   WORKING DIRECTORY MAY BE USED WITH CAT FILES ONLY
                      AT INDEX <XX>
FILE LITERAL ERROR:   EMPTY WORKING DIRECTORY AT INDEX <XX>
FILE LITERAL ERROR:   CONFLICTING ATTRIBUTES AT INDEX <XX>.
FILE LITERAL ERROR:   REMOTE FILES NOT IMPLEMENTED.
FILE LITERAL ERROR:   EMPTY STRING OR INVALID AT INDEX <XX>.
FILE LITERAL ERROR:   ILLEGAL CHARACTER AT INDEX <XX>.
FILE LITERAL ERROR:   SITE LENGTH > 8 AT INDEX <XX>.
FILE LITERAL ERROR:   EFN LENGTH > 44 AT INDEX <XX>.
FILE LITERAL ERROR:   SUBFILE LENGTH > 31 AT INDEX <XX>.
FILE LITERAL ERROR:   MEDIA LIST LONGER THAN 10 AT INDEX <XX>.
FILE LITERAL ERROR:   MEDIA NAME > 6 AT INDEX <XX>.
FILE LITERAL ERROR:   ILLEGAL DEVICE CLASS AT INDEX <XX>.
FILE LITERAL ERROR:   LOGICAL VOLUME NAME > 33 AT INDEX <XX>.
FILE LITERAL ERROR:   UNKNOWN ATTRIBUTE AT INDEX <XX>.
FILE LITERAL ERROR:   ATTRIBUTE NOT ALLOWED AT INDEX <XX>.
FILE LITERAL ERROR:   WRONG VALUE FOR THIS ARGUMENT AT INDEX <XX>.
FILE NOT CONNECTED.
FILE NOT OPEN.
FLOAT CONVERSION ERROR;UNEXPECTED RESULT.
FUPDATE UNSUCCESSFUL.
IMPLICIT DEASSIGNATION FAILED.
INTERNAL ERROR, CONSISTENCY CHECK FAILS.
INVALID CONNECTION.
```

```
OPEN FAILED, USE LINK CMD: 'FILE=(FILEORG=XX,NBBUF=2,NUMBER=N)'.
OPEN IN APPEND MODE UNSUCCESSFUL.
OPEN IN INPUT MODE UNSUCCESSFUL.
OPEN IN OUTPUT MODE UNSUCCESSFUL.
OPEN UNSUCCESSFUL (DFLDEF).
OPEN UNSUCCESSFUL (RFLDEF).
OVERFLOW (MAX REAL VALUE ASSUMED) (ETOF).
POINTER HAS A NIL VALUE OR IS UNDEFINED.
PRECISION LOST ON MATH FUNCTION CALL.
PRECISION OF DOUBLE GREATER THAN 18 (18 ASSUMED).
PROCESSING MODE FORBIDDEN FOR TERMINAL (TAM).
REQUIRED OR TOO LONG OPTION STRING.
SEEK FAILED (CLOSE).
SEEK FAILED (GET).
SEEK FAILED (OPEN).
SIZE OF THE REQUIRED MEMORY SPACE IS NEGATIVE.
STANDARD TERMINAL ACCESS METHOD NOT IMPLEMENTED.
TRY TO CONNECT A FILE TO AN UNKNOWN EFN.
UNDERFLOW (MIN REAL VALUE ASSUMED) (ETOF).
UNKNOWN FILESTAT IN A FILE LITERAL.
WRITE FAILED (CLOSE_EWRECORD).
WRITE FAILED (OPEN_EWRECORD).
WRITE FAILED (PUT_EWRECORD).
WRITE FAILED (PUTX_EWRECORD).
WRITE OR READ FAILED (GET_EWRECORD).
WRONG INPUT PARAMETER IN THE STRING-DOUBLE CONVERSION.
WRONG PROCESSING MODE.
```

5.8 An Example of Execution and Debugging

This section contains an example of interactive compilation, link, and execution with debugging. It also includes two erroneous messages, marked with a dollar sign, and their accompanying warning message (two asterisks).

```
  1  /* acker.c */
  2  #include <stdio_h>
130  static acker(m,n)
131  int m,n;
132  {
133   if (m == 0)  return(n+1);
134   else
135      if(n==0) return(acker(m-1,1));
136      else
137      return(acker(m-1,acker(m,n-1)));
138  }
139  main()
140   {
141   int r;
142
143    r = acker(3,6);
144    printf ("acker (3,6) is %i\n", r);
145    printf ("acker (3,5) is %i", acker (3,5));
146  }

S: c acker_hg lsfy.test.cc.sllib

>>>18:20 C  45.01  13 -1
   18:20:24 AUG 19, 1998 X325 .9
   compilation of LSFY.TEST.CC.SLLIB: ACKER_HG
   Error Syntax   ***  at ILN
   378: Unrecognizable statement.
   1 ERROR
   $ 378 . int m,n;
   $     1
   ***   $ 1 C23 Unrecognizable statement.
   CL.10(30.22) summary for ACKER_HG: ***1 ,cu produced.
<<<18:20
```

After error correction:

```
S: lk acker_hg command='LIST=S,STACK3=(INITSIZE=128K,MAXSIZE=256K)'

>>>18:21 LK 90.00  17 -2
   WORKING ON: ACKER_HG
   LK00.(90.00)
   SUMMARY FOR ACKER_HG
   NO ERROR DETECTED .
   OUTPUT MODULE PRODUCED
<<<18:21

S: exec_pg acker_hg
   acker (3,6) is 509
   acker (3,5) is 253
```

6. Programming Considerations

6.1 Portability

The following sections describe some situations to avoid or to be aware of when writing portable C programs.

6.1.1 Lexical and Syntactical Features

Identifiers, internal data, and external data are restricted as follows:

- Identifiers cannot be longer than 31 characters.
- Identifiers do not allow special characters, with the exception of the dollar sign ($) when LEVEL=GANSI or GCOS 7.
- Identifier spelling is different for lower case than upper case.

Do not use highly-nested patterns, in particular the following:

- INCLUDE files (>15)
- Conditional expressions (>10)
- Function calls (>30)
- Block, structures, iterators, switch, conditional compilation, types (>20)
- Array dimensions greater than 6

Do not use hexaliteral values, such as 0xnnnn, that contain 4 hexadecimal digits and the left-most bit set to 1. If you assign the hexaliteral value to int or long, the result depends on the size of int and long.

6.1.2 Data Representation

- Do not use the signed char and do not assume that a plain char is signed.
- Do not assume that the char coding value is equal to either ASCII or EBCDIC. The result of the char expression is different depending on which is used. The following expression shows this:

  ```
  'Z'-'A'
  ```

- Use the sizeof operator to determine the size of an object.
- Do not assume that a pointer equal to 0 is valid because a DPS 7000 pointer contains a segmented address. Such a pointer does not necessarily point to memory that contains 0.
- Avoid the use of bit fields. This is because their allocation and maximum length depend on their implementations.
- Be aware that the precision and range of the float and double values varies, depending in the hardware. Directly testing a floating point value is dangerous, because the two values can never be equal, for example when exiting a loop. It is better to use the range as follows:

  ```
  while (abs (x - x0) > eps)
  ```

6.1.3 Data Allocation

- Do not assume that the relative allocations of data of the same storage class in the same block are always equivalent. For example, do not assume that the value of &a+sizeof(a) and the value of &b are equivalent in the following example:

```
int a;
int b;
```

- The relative allocation of data with the same storage class in different files or with a different storage class is not always into the same segments. On DPS 7000 they fall under different segments, as an example for static versus dynamic or automatic versus dynamic
- Use function names or use the functions from vararg_h (if the number of parameters is variable) for the relative allocation of parameters. The following is an example of this:

```
f(a,b) &a+4 is not equivalent to &b
```

- The relative allocation of dynamic objects is not always into the same segment. The following is an example of this:

```
p1 = malloc (4); p2 = malloc (n);
p1+4 is not equivalent to p2
```

- Do not assume the allocation boundary of objects. For example, do not assume that an object is word aligned. The following is an example of this:

```
p1 = malloc (4); /* p1 assumed to point on a word
                    boundary */
```

6.1.4 Statements and Expressions

- Do not use the symbols ++, --, -=, or += when the operand occurs more than once, as this can cause "side effects". Three examples of this are shown below:

```
(1) f((a=b),a)
(2) a=(i+b)+i++
(3) (b[i]=a[i++])
```

  Side effects occur when a calculation modifies something else in addition to what is intended in the environment. For example, in line (3), above, the expression `i++` sends `i+1` but also modifies the contents of `i`.

- Do not assume the evaluation order of expressions. In lines 1 and 2 below, the first expression is not always equivalent to the second. Do not rely on possible short cuts in the evaluation of expressions. For example, in the first line (1) of the following evaluation, do not assume that j/i will not be evaluated if the first part (i != 0) is yielded "false".

```
(1) if ((i != 0) & (k = j/i))
(2)   if ((i != 0) && (k = j/i))
```

- Shift operations can be arithmetic or logical. They are logical on DPS 7000.
- Be careful when using % with negative values, especially on the sign of the result (the sign of the first operand on GCOS 7).

6.1.5 Pointer Handling

- Do not use if (p) or while (p--) instead of if (p!=NULL), because the NULL value is not necessarily 0.
- Do not use an address before or after data, because the address can point outside of the segment. For example:

```
int t[10]; for (p = t; p < t+10; p++)
```

or

```
int t[10]; for (p = t+9; p >= t; p--)
```

- Do not use indexing outside of the bounds of an array,because an exception will occur when crossing the segment boundary.

```
int t[10]; i = p - t; t[i] = 2;
```

- Be careful when using pointer arithmetic, because on DPS 7000, pointers contain a segmented address.

6.1.6 Library

- To use low and high-level file access on the same file, close the file between accesses.
- Do not make assumptions about character coding on the files. (ASCII versus EBCDIC)
- Do not use the functions signal, getenv, system, fork, wait, because, when implemented, they are highly system dependent. The functions wait and fork do not exist on GCOS 7.
- Avoid redefining any function from the library.
- Include the header file corresponding to the library functions you use (for example `stdio_h` for `printf`).

6.2 The GCOS 7 Preprocessor

The C preprocessor is integrated in the GCOS 7 C compiler. The LEVEL keyword indicates how to process the source text. This subsection discusses some preprocessor commands not yet implemented in all C compilers. The next subsection discusses the differences between the STANDARD and ANSI levels.

6.2.1 #<newline>

If there is nothing between the # character and the end of the line, this preprocessor directive has no effect. In general, it is used to "space" different macro definitions.

6.2.2 defined <identifier>

The two boolean expressions 'defined <identifier>' and 'defined (<identifier>)' are identical. They are preprocessor expressions which may occur in the commands #if or #elif. They return a value of 1 if <identifier> is a defined macro name at evaluation time. Otherwise they return a value of 0.

6.2.3 #elif <constant-expression><new line>

This directive is equivalent to:

```
#else <new line>
#if <constant-expression><new line>
```

The #elif directive is used inside a conditional loop of the preprocessor, between #if and #endif. It may be followed by an #else command corresponding to the <else> clause of the conditional loop containing the directive. You can have several #elif commands in sequence. Each <constant-expression> is evaluated (beginning with that of the initial #if clause) until one of them returns a "true" value (non-zero). Only the lines depending on the "true" clause are taken into account. If none are "true" then the (optional) #else clause is taken into account.

```
#if const_E1
<line_group_1>
#elif const_E2
<line_group_2>
#elif const_E3
<line_group_3>
.
.
.
#elif const_En
<line_group_n>
#else
<last_group>
#endif
```

6.2.4 #error

The #error command is one that is introduced. It produces a compile-time error message (SEV 3). This message includes the string constant that is an argument to the #error. It can detect programmer inconsistencies and violations of constraints during preprocessing, as follows:

```
#if defined(A_THING) && defined(NOT_A_THING)
#error "Inconsistent thing!"
#endif

#if SIZE % 256 !=0
#error "SIZE must be a multiple of 256!"
#endif
```

The error message is as follows :

```
***   $  1  B200  #error : Inconsistent thing!
***   $  1  B200  #error : SIZE must be a multiple of 256!
```

6.2.5 Predefined Macros

The GCOS 7 C compiler provides built-in macros for the programmer. These macros cannot be redefined or undefined, and they do not need a header file to define them. The value of each macro is as follows.

LINE	A decimal integer constant that represents the current line number in the source file. That is the line that uses the macro _LINE_.
FILE	A string constant that represents the name of the source file being compiled.
DATE	A string constant that represents the date of the translation of the source file. For example, "Dec 20 1990".
TIME	A string constant that represents the time at which the translation occurred. For example, "14:22:00".
STDC	A decimal constant 1 to indicate implementation conforming to ANSI C. _STDC_ = 1 for LEVEL = ANSI or GANSI.
GCOS 7	A decimal constant 1 to indicate implementation conforming to GCOS 7. _GCOS 7_ = 1 for LEVEL = GCOS 7 or GANSI.
VERSION	A decimal integer constant that represents the current version of the GCOS 7 C compiler. The current value is 40.

6.2.6 #line

The #line command is one that is developed. It can have one of the following forms:

```
#line integer-constant "filename"
#line integer-constant
#line pp-tokens
```

The third form is a preprocessing directive of the form. The preprocessing tokens after line on the directive are processed just as in normal text. The directive that results after the replacements matches one of the two previous forms and is then processed accordingly.

6.2.7 Macro Definition and Expansion

This subsection discusses how the ANSI level processes the macros.

- A macro appearing in its own expansion must not be expanded again. In this way, a programmer can redefine a function in terms of its old definition, as follows:

  ```
  #define sqrt(x) ((x)<0 ? sqrt(-x) : sqrt(x))
  ```

- An argument replaces its corresponding parameter in the replacement list, unless the parameter is either preceded by a # or ## preprocessing token or followed by a ## preprocessing token. This replacement occurs after all the macros contained in the corresponding argument have been expanded.
- A macro can be redefined if the new definition is identical to the existing definition. This is a "benign" modification. There is no error message.

6.2.8 Stringing and Merging Tokens (# and ## Operators)

The ANSI level limits the amount of control that the programmer has over merging tokens and converting macro parameters into strings.

Within a macro definition, the # character is recognized as an unary string operator that must be followed by the name of a macro formal parameter. During macro expansion, the corresponding argument actually enclosed in string quotations replaces the # and the formal name.

The following statements are an example of this.

```
#define TEST(a,b) printf(#a "<" #b "=%d\n",(a)<(b) )
TEST(0,0xFFFFFFFF);
```

After preprocessing and string concatenation, in this example the source text is as follows:

```
printf("0<0xFFFFFFFF=%d\n", (0)<(0xFFFFFFFF) );
```

A merging operator controls the merging of tokens to form new tokens. The two tokens surrounding any ## operator are combined into a single token before the replacement list is reexamined for more macro names to replace. If the combination is not a legal token, the result is undefined. The resulting token is available for further macro replacement.

The following statements are an example of this.

```
#define glue(a,b)    a ## b
#define xglue(a,b)   glue(a,b)
#define HIGHLOW      "hello"
#define LOW          LOW ", world"
glue(HIGH,LOW);
xglue(HIGH,LOW);
```

After preprocessing and string concatenation, the source text is as follows:

```
"hello";
"hello, world";
```

6.2.9 Preprocessor Output

When the EXPLIB keyword invokes the compiler, an SL library member can be created that is not part of the compilation listing. This member contains the output of the preprocessor, and its name is derived from the source file name, suffixed by "_I". It contains the results of the preprocessing phase, which are as follows:

- Include files are inserted
- #define directives are deleted
- Macros are substituted

This member (file) can be the input for another compilation step, the results of which are the same as those of the compilation of the initial program.

With macro substitution, a line can be longer than 255 characters, a length that an SL-library subfile cannot normally record. The macro substitution truncates the line at 254 characters and terminates the line with an ending back-slash (\). The macro substitution writes the remaining portion of the expanded line on the next record and repeats this splitting process if needed.

6.3 Pre ANSI and ANSI Compilers

This subsection describes the differences between pre-ANSI compilers and ANSI compilers. Pre-ANSI compilers include some of those on UNIX, and the GCOS 7 compiler with a level of STANDARD or GCOS 7.

Because this section describes only the differences, it does not list the compiler extensions, such as new keywords.

6.3.1 Expanding Macro Parameters in Strings

The GCOS 7 C compiler (with STANDARD level) can expand a macro parameter that is in a string. The compiler substitutes the macro parameter with the actual argument.

This is a feature of the GCOS 7 C compiler (with STANDARD level). In this case, macro parameters occurring in a string are substituted by the actual argument.

For example, in the following statement, `a(u)` yields `"u"` in a GCOS 7 C compiler with STANDARD or GCOS 7 level. In all other cases, `a(u)` yields `"x"`:

```
#define a(x) "x"
```

6.3.2 Trigraph Sequences

The ANSI compiler processes a trigraph sequence as a single character. For example, the following trigraph sequence becomes a **"/"** (slash) character:

```
'??/'
```

6.3.3 Octal Digits

A compiler at the ANSI or GANSI level issues a syntax error if either a digit 8 or 9 occurs in an octal coded literal.

6.3.4 Long Float Type

A compiler at the ANSI or GANSI level issues a syntax error if this declaration occurs in the source text. In pre-ANSI compilers, this is a synonym of double.

6.3.5 Constant Strings

ANSI level compilers do not modify constant strings. The MODSTRNG user option makes this modification, and the user can control this feature.

At the pre-ANSI level, a program statement can prevent the compiler from allocating literal strings in a non-writeable segment and then creating space in data segments to contain them. For example, the following statements do this:

```
char *p = "qwerty";
 ...
*p = 'a';
```

6.3.6 Separating Assignment Operators

At the ANSI level, a blank space cannot separate assignment operators. However, this is possible at the GCOS 7 C compiler level. Assignment operators include the following:

```
+=   -=   *=   /+   <<=   >>=   %
```

6.3.7 Empty Declarations

The ANSI level compiler does not accept empty declarations. They lead to a syntax error. Empty declarations include the int and extern declarations.

6.3.8 Linkage

For identifiers with external linkage, the GCOS 7 C compiler uses a lexical scope. It uses internal definition only for the lines that follow this definition.

The ANSI standard is the opposite, in that it uses a file scope. In this way, a non-local definition is valid for the entire file in which it is contained. Most of the UNIX compilers also use the file scope.

The compiler level determines how the function named f will be called in main. In the example program shown below, this is as follows:

- At STANDARD or GCOS 7 level, an externally defined function with name f will be called in main
- At ANSI or GANSI level, the internally defined function with name f will be called in main.

EXAMPLE PROGRAM:

```
main () {
   f();}
static f () {
   }
```

❑

6.3.9 Conversions

The ANSI level compiler uses sign extension to convert from `int` to `unsigned` with increasing length. The STANDARD level compiler does not use sign extension.

6.3.10 Sizeof

The ANSI level compiler does not allow `sizeof` on functions or bit-fields. The `sizeof` parameter in a boolean expression returns 4 at the ANSI level and returns 1 at the STANDARD level.

6.3.11 Bit-Fields

The ANSI level compiler does not initialize unnamed bit fields, while the STANDARD level does. At the ANSI level, bit-fields are not an argument of the `sizeof` or `address` operator.

6.3.12 Pointers to Functions

The ANSI level compiler does not allow pre-increment, post-increment, pre-decrement, and post-decrement on pointers to functions.

6.3.13 Constant Expressions

The ANSI level compiler does not allow comma expressions in constant expressions. These expressions include array bounds, case selectors, and static initializers.

6.3.14 Preprocessor Features

The ANSI level compiler does not allow recursive macros. However, it does allow lexical rescanning.

In the example shown below, the STANDARD level compiler scans the `"a.1"` as a single token that yields the real value 1.1. The ANSI or GANSI level compiler scans it as two separate tokens, 1 and .1. The example is as follows:

```
#define a 1
a.1;
```

In this case, a binary "merging" operator can change this scanning on the ANSI compiler level.

With LEVEL = GCOS 7 or STANDARD, the integer-constant after the preprocessor command #line is the new number of the current source line. With LEVEL = ANSI or GANSI, the integer-constant is the new number of the following source line.

6.4 Performance Considerations

The following notes help improve performance. For more information, see the section describing optimization.

- Avoid extensive use of bitfields.
- Avoid using signed char and signed bitfields.
- Declare strings as "not modifiable" (MODSTRNG=0 or ANSI/GANSI parameters)
- Avoid using complex conditional expressions, such as the following:

  ```
  x=f().(a?b:c).
  ```

  In this case the following is preferable:

  ```
  if (a) x=f().b; else x=f().c;
  ```

- Include the header files, if they exist. These retrieve builtin functions.
- Avoid extensive use of setjmp/longjmp.
- Use memxxx functions rather than xxxbuf. Use memxxx functions rather than string functions. For example, use memcpy rather than strcpy or cpybuf.
- Use the buffering mechanism.
- Use the read and write functions if high performance is needed on some I/Os. However, note that these functions do not belong to the ANSI standard.
- Use direct access on UFAS files with fixed size records.
- Avoid using SSF format if it is not necessary to edit the file.

7. GCOS 7 Specific Considerations

7.1 Size and Limits

GCOS 7 places restrictions on programs for maximum allowed values and minimum requirements. The following is a list of the maximum values that GCOS 7 allows in a program. The term "at the most" indicates it is a fixed limitation for the current implementation. A program in GCOS 7 can have:

- At most, 30 nesting levels of compound statement, iteration control structures and selection control structures.
- At most, 50 nesting levels of conditional inclusion.
- At most, 32 significant initial characters in an internal or external identifier or a macro name.
- At most, 128 parameters in one function definition.
- At most, 128 arguments in one function call.
- At most, 128 parameters in one macro definition.
- At most, 128 arguments in one macro invocation.
- At most, 512 characters in a character string literal.
- At most, 15 nesting levels for #included files.
- At most, 20 levels of nested structure or union definitions.

The following is a list of minimum conditions that GCOS 7 requires of programs. The term "at least" indicates conformity with the ANSI standard, and means that there is no maximum limit imposed. It is context dependent. A program in GCOS 7 must have:

- At least 12 pointer, array, and function declarators (in any combination) that modify an arithmetic, a structure, a union, or an incomplete type in a declaration. However, only 6 dimensions for an array and 19 indirections for a pointer expression.

- At least 31 nesting levels of parenthesized declarators within a full declarator.
- At least 32 nesting levels of parenthesized expressions within a full expression.
- At least 511 external identifiers in one translation unit.
- At least 127 identifiers with block scope declared in one block.
- At least 1024 macro identifiers simultaneously defined in one translation unit.
- At least 509 characters in a logical source line.
- At least 32767 bytes in an object.
- At least 257 case labels for a swith statement.
- At least 127 members in a single structure or union.
- At least 127 enumeration constants in a single enumeration.

7.2 Implementation-defined Features

7.2.1 Translation

The compiler produces diagnostics that can be characterized by:

Location	A line number, internal line number, and column position
Severity	A warning, observation, or error
Message	A brief description of the error and a reference to a specific paragraph about error messages.

7.2.2 Environment

The arguments of the main function can be referenced by names different from argc and argv. Main arguments are interpreted as argc and argv when there are two defined arguments for this function, no matter what their types. If the type does not correspond to the expected type for either argc or argv, the behavior is undefined.

For the interactive device, only files with the device specifier TN are an interactive device. The standard files stdin,stdout and stderr are interactive devices when executing a program under IOF.

7.2.3 Identifiers

An identifier has up to 31 significant characters, either with or without external linkage. Upper or lower cases are significant for all identifiers, even with external linkage.

7.2.4 Characters

The source and execution character sets are EBCDIC. Adding multibyte characters through the localization facilities can extend the execution character set.

Wide characters are represented as integers (32 bits), and each constant character can not have more than one wide character. More than one character in a constant character is detected as an error at compile-time.

A plain character has the same range of values as unsigned char. The unsigned char type describes the plain character.

For more information, refer to the localization chapter.

7.2.5 Integers

The representation of an integer is a two's-complement representation. The integer type values are as follows:

short int	[-32768,+32767]
int,long int	[-2147483648,+2147483647]
unsigned int, unsigned long int	[0,4294967295]

Shifts left are logical shifts. The sign of the remainder with signed operandi is the sign of the first operand (-3%-2 --> -1, 3%-2 --> 1).

7.2.6 Floating Point:Internal Representation

This subsection describes the internal representation of floating point data for float types, and double types. This includes the value, format, sign S, characteristic C, and mantissa M.

7.2.6.1 Float Type Data

The float type corresponds to a simple precision floating point datum.

Value:

The following formula defines the value of a real datum.

$$V = (-1)^{S} * 16^{E} * .M$$

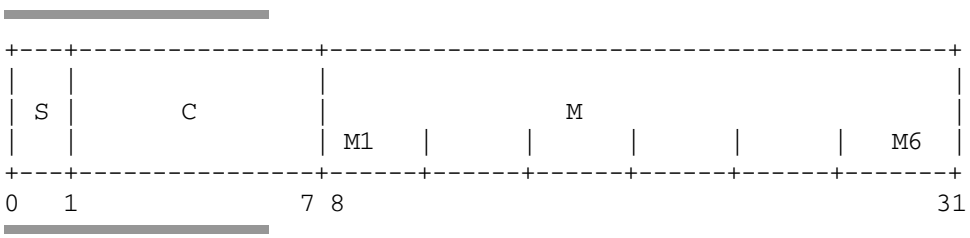
The following values apply to this formula:

E	is C-64
S	is the sign
E	is the exponent
C	is the characteristic
M	is the mantissa of the real datum

A real datum with the mantissa equal to zero represents the value zero. The value of true zero is represented either by a real datum with all 32 bits equal to zero, or by a real datum with the left hand bit set to one and the 31 right hand bits equal to zero.

Format:

A real datum occupies four consecutive bytes. The format is as follows:



Sign S:

The sign S of a real datum is contained in bit 0. The value of S is as follows:

0	Positive sign
1	Negative sign

Characteristic C, Exponent E:

The bits 1 through 7 contain the characteristic C of a real datum. Its range is as follows:

```
0 <= C <= 127.
```

The exponent E is the power to which 16 is raised when calculating the value of the real datum. The value of the exponent E is as follows:

```
E = C-64.
```

Mantissa M

The mantissa M is the hexadecimal number that bits 8 through 31 contain. It is 6 hexadecimal digits. The radix point is at the left of the high order digit position, so that the following is true:

$$\sum_{i=1}^{6} M_i \times 16^{-i}$$

7.2.6.2 Double Type Data

The double type corresponds to a double precision floating point datum.

Value:

The following formula defines the value of a double precision datum:

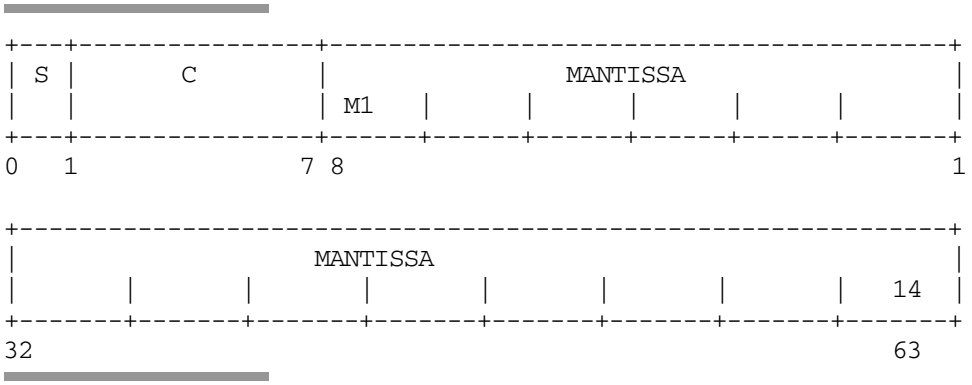
$V = (-1)^{S} x 16^{E} x .M$

The following values apply to this formula:

E	is C-64
S	is the sign
E	is the exponent
C	is the characteristic
M	is the mantissa equal to zero. A value of true zero is represented by a double precision datum with 63 right bits equal to zero.

Format:

A double precision datum occupies eight consecutive bytes. The format is as follows:



Sign S:

The sign S of a real datum is contained in bit 0. The value of S is as follows:

0	Positive sign
1	Negative sign

Characteristic C, Exponent E:

The bits 1 through 7 contain the characteristic C of a real datum. Its range is as follows:

```
0 <= C <= 127.
```

The exponent E is the power to which 16 is raised when calculating the value of the floating point number. The value of the exponent E is as follows:

```
E = C-64.
```

Mantissa M:

The mantissa M is the hexadecimal number that bits 8 through 63 contain. It is 14 hexadecimal digits.

$$\sum_{i=1}^{14} M_i \times 16^{-i}$$

NOTE:
The options of truncation and rounding are under reserve.

7.2.7 Arrays and Pointers

The integer type to hold the maximum size of an array is an unsigned long int.

Assignments can be made between an integer and a pointer. However, because this is not always done, the value of an integer expression assigned to a pointer does not always lead to a valid pointer representation, which can then be used to reference a memory location. The value of an integer expression remains unchanged after the assignment operation.

For more information about the unsigned long int and ptrdiff_t type definition, see the section describing the stddef.h standard file.

7.2.8 Registers

The register attribute for an object allows the user to access an object more rapidly. To do this, only the optimizer and objects allocator strategy are required.

7.2.9 Structures, Unions, Enumerations, and Bit-Fields

When a member of a union object is accessed by a member with a different type, the resulting behavior can be undefined or can even contain exceptions.

If necessary, successive non-zero width bit-fields of structures or unions can overlap adjacent units. The order allocation of bit fields in the same unit is high-order to low-order.

Sometimes padding is necessary to represent successive bit-field and non-bit-field members.

A plain int bit-field is treated as an unsigned int bit-field.

The values of an enumeration type have the same representation as the int type.

7.2.10 Qualifiers

By definition, a volatile-qualified type object is a non-optimizable object that is evaluated at each access.

7.2.11 Declarators and Statements

For the maximum number of declarators that may modify an arithmetic, structure or union type and the maximum number of case values in a switch statement, see the subsection describing the size and limits of the implementation, earlier in this section.

7.2.12 Preprocessing Directives

The value of a single character constant in a constant expression that controls the conditional inclusion matches the value of a member of EBCDIC character set. This value is always positive.

For more information about locating and naming an include file, see the subsection describing the searching rules for include files, in the section about compilation.

For more information about recognized pragmas, see the section describing preprocessor pragmas.

7.2.13 Library Functions

The following notes pertain to the library functions.

- The NULL macro expands in the same way as the 0 integer constant.
- The functions beginning with "is" (for example, isprint) use the standard EBCDIC character set. However, they also support any character sets specified in the current available localization.
- The last line of a text stream requires a newline.

When space characters for a text stream are written out immediately before a newline, they appear as variable size records when read in a GCOS 7 file. If not, they do not appear at all.

- The maximum number of null characters that can be appended to data written to a binary stream is as follows:

```
1 - (the maximum size of a record of the file)
```

- The file position indicator of an append mode stream is initially positioned at the end of the file.
- When writing a text stream, truncation occurs only at open time in the following processing modes:

```
w/w+/wb/w+b
```

- The library functions support only line and full buffering. Refer to the section describing General I/O Considerations for more precise details.
- A zero-length file can exist. In this case, the ssf format file contains only its control record.
- The appendix describing file and volume syntax contains the file names syntax.
- A file can be successfully opened several times, depending on its processing mode, its type of access and its ability to be shared. for more information, see the chapter titled General I/O Considerations.
- An open file cannot be removed.
- A file name cannot duplicate an existing one.
- The printf and scanf function describe the output and input of the %p specifier. For more information, see the section describing general I/O considerations.
- There is no distinction for the '-' character in the scanlist of the %[conversion in the fscanf function. This is true even when it is not the first nor the last character of the scanlist.
- It is possible to request zero-size memory when using allocation functions. This request allocates 4 bytes.
- If the tmpfile function opens any files, the abort function does not remove nor flush remaining open files. In this case, the system closes the open files.

7.3 Size of Data Basic Types

TYPE	SIZE	MIN.VALUE	MAX.VALUE
char	1	0	255
unsigned char	1	0	255
int	4	-2147483648	+2147483647
long int	4	-2147483648	+2147483647
short int	2	-32768	+32767
unsigned int	4	0	4294967295
unsigned long int	4	0	4294967295
unsigned short int	2	0	65535
float	4	10^{-78}	10^{+76}
double	8	10^{-78}	10^{+76}
pointer	4		

- The above sizes are given in bytes (1 byte = 8 bits).
- Le number of significant digits is 5 for a "float" and 14 for a "double".
- All types are byte aligned.

7.4 Pointer Specific Behavior

The NULL pointer is represented in GCOS 7 by the integer value zero. (That is, all 32 bits are set to 0).

All other languages under GCOS use the value -1 to represent a NULL pointer. Consequently, you must take account of this incompatibility for every interface between the C language and another language.

Also, the manipulation of pointers in the C language (notably for address calculations) does not take ring evaluation into account. Any program, written in another language, which returns a pointer to a program written in C, must reset to zero the bits of that pointer which describe the ring. Otherwise, you may obtain incoherent results.

7.5 Allocation and Segmentation

The static variables of a C program are allocated in one or more DPS 7000 data segments. The order of allocations is unknown in advance.

For example, if an array is allocated at the beginning of the segment and we execute the following program:

```
static int TAB [4];
  main ()
              int p,
              for (p=&TAB[3]; p>=&TAB[0];p--)...
```

At this point, an exception 06.00 ACCESS OUT OF SEGMENT BOUNDS occurs on the last pass of the "for" loop at the evaluation of the output conditions. This error is irremediable.

7.6 Implementation-defined Behavior

7.6.1 Identifier Spelling

The following rules apply to identifiers:

- The number of significant characters of an identifier with an external link is 31.
- Both upper and lower case characters are significant in an external identifier.
- The GCOS 7 and GANSI levels use a dollar sign ($) to extend identifier spelling. For example, the following are valid identifiers:

```
Abc$1_
$32$
```

7.6.2 Characters

The character code used by the compiler is EBCDIC.

The type "char" is treated as the type "unsigned char".

7.6.3 Arrays and Pointers

The maximum size of an array is 4 Megabytes. The SIZEOF operator is of type "unsigned". The type "int" can be assigned to a pointer, but this latitude is considered as NON-PORTABLE.

7.6.4 Registers

The "register" attribute is accepted by the syntax, but has no semantic value. All the "register" variables are allocated in the local data zone.

7.6.5 Structures, Unions and Bit Fields

The members of a structure are allocated in ascending order of address.

The alignment of each member of a structure is that of its type.

The alignment of a structure containing only bit fields is that of an "int".

The bit fields are not signed and behave as "unsigned". They are allocated contiguously in memory in ascending order of address.

7.6.6 Line Command

The #line preprocessor command changes the current and following line numbers of the reported listing at the end of compilation. This command does not have the same effect on the error messages given by the C compiler as it has in standard UNIX.

7.6.7 #include Command

The preprocessor can change the file name in an #include preprocessor command. This makes it easier to import a file from UNIX. The following conditions apply to the name change:

- Underscores replace any periods in the file name.
- Minus signs replace slashes.

For example, for a file name #include <stdio.h>, the preprocessor changes the name to STDIO_H for the search in the include libraries.

7.7 Calling from another Language

The main function can be called outside a C program from another language with the name of its SYMDEF generated at compilation time (the name of its source member). However, this call can be done only once, if you manipulate the standard files. Remember that all files are closed at the end of execution of the main function.

8. Building Packages

8.1 What is a C/GCOS 7 Package?

A C application consists of one or more source files needing compilation and their include files. Each source file can contain one or more C functions. A package is the regrouping of several source files, and one or more packages form an application. A package is located between the application and the source file and is the compilation unit.

The package is characterized by its interface. The interface determines the objects that are visible from the exterior and the externally-defined objects that it needs.

Packaging enhances programs in the following ways:

- Encapsulating data that limits the visibility of certain objects.
- Shortening the amount of time needed to call functions.

8.2 Why Package an Application?

8.2.1 Encapsulation

The C language uses the following two types of function visibility:

- Static, which is local to the source file.
- External, which is visible to all other functions.

In the case of large applications, the C language can verify and define the objects that are common to some functions but inaccessible to others. The packaging facility restricts the visibility of the C objects that are manipulated from outside the package as well as those that are manipulated from the inside.

Any object defined in the package and not exported out of the package can be manipulated only within the package. Similarly, any object defined on the exterior of the package and not imported into the package cannot be manipulated from within the package. The package construction restricts object visibility to avoid confusion (for example, name conflicts). However, the package must clearly and exhaustively define all the objects so that it can best function as an interface between the package interior and exterior.

8.2.2 Performance

The packaging of an application improves program performance. External function calls are relatively costly, and transforming external calls into internal calls brings about a significant gain in performance. Because the compiler has greater visibility of the source, the compilation becomes even more efficient. This is because the compiler can include a whole group of files in the compilation. This is particularly evident in the case of the inliner, which functions only under this condition. In addition to the performance gain, the segmentation is more efficient because only one compile unit (CU) is generated for a file group.

The first step in building a package is to identify the files that it comprises. To ensure a performance gain for calls within the package (intra-package calls), include the functions that frequently call one another together in one package. There is a small penalty for calling a function from an exterior package when the function is not recursive and a large penalty when the function is recursive.

For a function containing three non-recursive parameters, an estimated 20 intra-package calls are equivalent to one call from the exterior. For a function containing recursive parameters, there is no performance gain and no cost increase for an intra-package call, but the estimated cost is double for a call from the exterior.

If an application is not very large, it is advisable to compile it as a single package that has neither exported nor imported objects.

It can benefit performance to duplicate some functions in other packages. If an application is divided into several packages and two or more of these packages have a critical need of one function, there is a significant performance gain if the function exists in both packages without being exported. On the condition that it is not recursive, this function can be defined in both packages and still optimize well.

8.3 Pragmas

A pragma is an ANSI feature that introduces target dependencies in a standard program. It has the following form:

```
#pragma <pragma command>
```

8.3.1 GCOS 7 Pragmas

The GCOS 7 C compiler recognizes the set of pragma listed below. If the compiler does not recognize a pragma command, it ignores it and issues a warning message.

For the GCOS 7 C compiler, a pragma has the following form:

```
#pragma PRAGMANAME <parameters>
```

8.3.2 PACKAGE and Related Pragma

The following pragma set deals with program packaging:

```
#pragma PACKAGE membername[,membername]...
#pragma AUTOPACKAGE membername[,membername]...
#pragma IMPORT objectname[,objectname]...
#pragma EXPORT objectname[,objectname]...
```

8.3.3 ALIGN Pragma

The ALIGN pragma controls the alignment of data in memory. This affects only main variables, not aggregate elements. The alignment constraint applies to the variables of a given list, to the names variables, to the (main) variables in a given storage class, or to all variables. This is shown as follows:

```
#pragma ALIGN <alignment> = <variable list>
#pragma ALIGN <alignment> = <storage class>
#pragma ALIGN <alignment>
```

In this pragma, the alignment can be halfword, word, doubleword, or quadrupleword. Storage class can be auto or static.

In the following example, b, s, and t are allocated a 4-bytes boundary, and no alignment is specified for a. The 0-length bit fields can manage alignment within a structure.

```
#pragma ALIGN word = b, s, t
static char b, a, t;
static struct St {int x:1; :0; int y:1} s;
```

8.3.4 BYREF Pragma

The BYREF pragma has the same semantics as the syntactical features that are & specific in the function declaration. This pragma indicates a set of functions for which parameters must be passed by reference rather than by value. It is used to communicate between languages. The form of the BYREF pragma is as follows:

```
#pragma BYREF functionname[,functionname]...
```

8.3.5 INLINE and OUTLINE Pragma

The INLINE and OUTLINE pragma is a set of pragma related to the optimizing function of procedure merging.

The INLINE pragma indicates a set of functions that must be in-line inserted. This is required when the INLINER option is off, and the OPTIMIZE level is less than 4. The OUTLINE pragma indicates a set of functions that must not be in-line inserted. This is required when the INLINER option is on or if the OPTIMIZE level is equal to 4.

These pragma are as follows:

```
#pragma INLINE functionname[,functionname]...
#pragma OUTLINE functionname[,functionname]...
```

8.4 What Comprises a Package

8.4.1 The Aim of the #pragma PACKAGE

A package regroups functions to provide the best performance. These functions are often contained in several sources. The #pragma PACKAGE is a directive that indicates which sources must be regrouped. For example, a package can be built of all the functions needed to execute a specific processing or of all the functions handling a specific data structure.

Syntax of the Directive

If the functions are taken from the file members 1 through n, with all members in the same library, the directive is as follows:

#pragma PACKAGE member1, member2, ... member n, or with several directives:
#pragma PACKAGE member1
#pragma PACKAGE member2
#pragma PACKAGE membern

A #pragma package directive can be located anywhere in a file. The file compilation that contains a PACKAGE directive also compiles all the other sources named in that directive. This compilation generates only one CU. The name of the CU is the same as the compiled file.

Some Restrictions

The following restrictions apply to packages:

- All the source files that make up a package must be in the same library. If any file is not found, the following error message (SEV 3) is displayed:

  ```
  INCLUDE TEXT NOT FOUND IN SPECIFIED LIBRARY, IGNORED
  ```

- If one file appears several times in the package order, the following warning message (SEV 2) is displayed and the package order is ignored:

  ```
  FILE NAME ALREADY USED IN PACKAGE PRAGMA ORDER
  ```

- The PACKAGE directives can appear in only one file, the name of which is sent to the compiler. If they appear more than once, the following warning message (SEV 2) is displayed:

  ```
  PACKAGE PRAGMA ORDER CANNOT BE INVOKED IN PACKAGE'S
  SUBFILES
  ```

- A function with a variable number of arguments can not be part of a package. For more information, see the section describing the STDARG Functions.

8.4.2 The EXPORT Directive

To call or reference any C object (function or variable) from the exterior of a package, the object must be specified in an export directive. This applies only to objects that are visible externally. This directive has no effect on static or automatic objects.

An object that is externally visible, but is not exported cannot be referenced from outside the package. However, this object does remain visible to all files making up the package. It works as if the externally visible, but non-exported object is transformed into a static object of the package. The object retains its external visibility to the member in which it is defined, but not to the other members that make up the package.

The syntax of an EXPORT directive is as follows:

#pragma EXPORT name1, name2, ..., namex

The following points apply to the EXPORT directive.

- The EXPORT directive can appear anywhere in one of the files in the package, not necessarily in the first. This is unlike the PACKAGE directive.
- There is no limit to the number of EXPORT directives allowed. There can be one for each exported object or one that references all the objects to export. The list of objects to export can continue over several lines as long as the last character of each line is the backslash character (\), as used in #define.
- As in standard mode, the compiler renames a C function that is called main with the file name given to the compiler at start time. The compiler automatically exports this name.
- Without an error resulting, an object can appear several times in an EXPORT directive or in several directives. However, in order to ease code readability and maintenance, it is better not to export the same object several times.
- Any object appearing in an EXPORT directive and called in at least one other file must be defined in one of the package files. If not, the following error message (SEV 3) is displayed:

  ```
  AMBIGUOUS LINKAGE OF THIS IDENTIFIER, <name>: EXPORTED
  ALTHOUGH NOT DEFINED
  ```

IMPORTANT:
An object must be exported if its address is assigned to a pointer that is passed (directly or indirectly) outside of the package.

8.4.3 The IMPORT Directive

To call or reference any C object (function or variable) from the interior of a package, the object must be specified in an import directive. This applies only to the external objects that are not defined in one of the package members. The syntax of an import directive is as follows:

#pragma IMPORT name1, name2, ..., namex

The following points apply to the IMPORT directive.

- The IMPORT directive can appear anywhere in one of the files in the package, not necessarily in the first. This is unlike the PACKAGE directive,
- There is no limit to the number of IMPORT directives allowed. There can be one for each imported object or one that references all the objects to import. The list of objects to import can continue over several lines on the condition that the last character of each line is the backslash character (\), as used in #define.
- Without an error resulting, an object can appear several times in an IMPORT directive or in several directives. However, in order to ease code readability and maintenance, do not import the same object several times.
- If any object defined in the package appears in an IMPORT directive, an error occurs. Because the compiler automatically imports the standard functions of C run time, a function having the same name as a standard run time function cannot be defined in the package. In both of these cases, the following error message (SEV 3) is displayed:

  ```
  AMBIGUOUS LINKAGE OF THIS IDENTIFIER, <name>: IMPORTED
  ALTHOUGH NOT DEFINED
  ```

8.4.4 Building the Package

8.4.4.1 Packaging at Design Time

Packaging large applications is best done in a modular fashion, using any method. The definition of the modules and the module interfaces directly constitute the packages forming the application.

8.4.4.2 Packaging an Existing Application

There are two ways to package an existing application.

The first way to package an existing application is to use the AUTOPACKAGE directive and then build a set of packages that make the application. The advantage is that it builds packages quickly because the semantics of C is fully preserved. The disadvantage is that the level of encapsulation is as low as that of the original C application.

The second way to package an existing application is to use the PACKAGE and IMPORT/EXPORT directives. The advantage is that the level of encapsulation is higher than that of the original C application. The disadvantage to this is that it requires more effort when building packages.

To begin packaging an existing application, the following must be identified:

- The files that make up the application packages. This is also discussed above in the subsection on performance.
- The objects to be imported and exported. This can be difficult and a method for identification is described below.

To identify the IMPORT statements, compile the package without import or export directives. To compose the list of imports, scan the compilation listing. For each object that requires an IMPORT order, the compiler sends the following message (where xxxx is the name of the object to import):

```
AMBIGUOUS LINKAGE OF THIS IDENTIFIER <xxxx>. REFERENCED
ALTHOUGH NOT IMPORTED
```

To identify the EXPORT statements, compile all the application sources. Delete all the compilation units from the CULIB that correspond to the package sources, and do a LINK. All the EXPORT objects are together in the LINKER listing, marked as NOLINK objects. The objects to export are the intersection of NOLINK and the objects defined in the package.

Because these identification methods can be involved, it is best to place all the package orders in an independent file and compile it. In this way, the application sources are not modified. When the package is established and tested, the file containing the packaging directives can be integrated into one of the package sources.

It is necessary to ensure the coherence of object type when they appear in several files, notably those functions called without being defined (the definition implicit as a function returning an integer). The linker does not detect this type of error during separate compilation. However, in the case of package, the compiler sends a SEV 3 error.

EXAMPLE:

If the file F1 contains a call to the function func but does not declare it, the code is as follows:

```
F1:
     main (){
     int i;
     func(i);
     }
```

If the function func is defined in F2, the code is as follows:

```
F2:
     double func(i)
     int i;
     {
     return ((double) i);
     }
```

When compiling the file PACK_ORDER as follows:

```
PACK_ORDER:
#pragma PACKAGE F2, F1
```

The compiler sends the following error message (SEV 3):

```
THE TYPE SPECIFIER IN DECLARATION IS NOT THE SAME AS IN
THE PREVIOUS ONE. UNDEFINED BEHAVIOR
```

If F1 and F2 are compiled separately, there is no problem in the link between these two files. This type of error depends on the order that the files are given and the PACKAGE order.

To avoid this type of error, add the definition of func in an extern expression as much as possible in PACK_ORDER. The following is an example of this:

```
PACK_ORDER
#pragma PACKAGE F2, F1
extern double func ();
```

The function func then returns a double for all the package files. In F1 there is no longer an implicit definition, but the precedent definition (the one from PACK_ORDER) remains. Also, there is no longer any type incoherence.

❑

8.4.5 The AUTOPACKAGE Directive

The AUTOPACKAGE directive is very similar to the PACKAGE directive. However, all the external definitions are automatically exported and all the external references are automatically imported.

The syntax of the directive and the restrictions on the directive are the same as for the #pragma package.

8.5 One-file Packages

When an application has only one file, it is always most efficient to make a package. The C compiler has an option that avoids having to add in the source the preprocessor order concerning the package. The following shows how to call this package with the compiler keyword:

```
C MYFILE MYSLLIB PACKAGE;
```

This option is equivalent to the following:

- A #pragma PACKAGE order on the file to be compiled.
- A #pragma EXPORT order for all objects defined in the file.
- A #pragma IMPORT for all the objects that are called but not defined.

This provides an improvement in performance without modifying the source or the object visibility (internal or external). By contrast, when the application does not define any single function and calls the functions only from the main (a self-containing application except for the C Run Time Package), it is more rigorous to insert the following order into the source:

```
#pragma PACKAGE X
```

This restricts the visibility of the internal objects and adds to security.

The size of the produced object code is always greater in an automatic package than when constructed in a #pragma package.

8.6 Summary

8.6.1 Pragma Syntax

The syntax of the pragma is as follows:

- #pragma PACKAGE membername[,membername]...
- #pragma EXPORT membername[,membername]...
- #pragma IMPORT membername[,membername]...

8.6.2 Object Visibility

The following table shows the visibility from the package of an object C, with external visibility in the package member. In this table, the term declared means that the object is declared but not defined.

OBJECT	EXPORT		IMPORT		NOTHING	
	Internal	External	Internal	External	Internal	External
Declared	Error	Error	Visible	Visible	Error	Error
Defined	Visible	Visible	Error	Error	Visible	Visible

The visibility of a static object (package or not) is limited to the member in which the object is defined.

For example, when compiling PACK_ORDERS:

```
#pragma PACKAGE F1, F2
#pragma IMPORT f
#pragma EXPORT g
```

With F1:

```
extern int f(), g(), h();
int g() {...}
int h() {...}
int l() {
    int m;
  ....
}
static int n() {...}
```

With F2:

```
extern int f(), g(), h();
...
```

This results in the following visibility:

- f	is defined in another package and used in the package.
- g	is defined in F1 and is visible inside and outside the package.
- h	is defined in F1 and visible only at the interior of the package.
- l	is defined in F1 and visible only at the interior of the package.
- m	is local to the function l.
- n	is a function defined in f1 and visible only in the file.

8.6.3 Application Packaging Steps

The steps for defining a package are as follows:

1. Identify the files to be regrouped in the package.
2. Compose the list of imports and exports.
3. Verify the coherence of the object types.
4. Analyze recursivity problems and function calls to avoid a penalty in program inefficiency.

9. Optimizing with C

9.1 Introduction

9.1.1 The Goals of the Optimizer

The C language, like other high-level programming languages (for example Fortran 77, PASCAL, and GPL), allows programmers to compose algorithms using concepts that are more abstract than those of the assembly code, 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 code. 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 cannot.

```
int a [100], b [100]
int i, j;

for (i=0; i <= 100; i++)
  for (j=i; j< = 100; j++)
a [i+j] = b [j-i];
```

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 * j + 4 * i
address (b [j-i]) = address (b) + 4 * j - 4 * i
```

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.

9.1.2 The Local Optimizer

In its first few version, the C compiler had two optimization levels, the instruction source optimization level and the extended linear sequence optimization level. These levels are still available, however, the global optimizer introduces new levels. The first level is automatically activated when the DEBUG option is on.

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 or a basic block, that are 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.

9.1.3 The Global Optimizer

The global optimizer 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 data flow that it handles allows the compiler to operate an elaborate optimization through the manipulations that the optimization functions perform on an internal representation of the source code. These manipulations include code deletion, insertion, and motion. These global optimization functions are as follows:

4. Anticipation and temporization.
5. Deleting partially redundant expressions.
6. Removing invariant expressions from loops.
7. Strength reduction 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 (or in-line insertion).

NOTE:
The optimization functions apply to the procedure level. There are no inter-procedural optimizations.

There are two types of optimization improvements:

1. Increased speed in program execution.
2. 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 the 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.

9.1.4 Optimization Levels

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

OPTIMIZE=0	No optimization
OPTIMIZE=1	Local optimization, limited to the source statement (the instruction source).
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, in which case it is the default.

9.2 Global Optimizer Functions

This section describes the different functions of the global optimizer and gives an example of each in the C 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. Therefore, 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.

9.2.1 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 expressions, as long as those expressions are valid.

```
a = 1;

if (valid)
      x = a + 3;
else
      x = a + 1;
```

This gives the following, after optimization:

```
a = 1;

if (valid)
      x = 4;
else
      x = 2;
```

The optimizer uses the basic elementary operations (arithmetic, logical, and comparative) for constant folding and copy propagation. 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.

9.2.2 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 = a + b + c;
if (a>b)
    x = 10;
else
    x = 20;
y = a + b + b;
```

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);
    x = 10;
else
    x = 20;
y = t + d
```

The intermediate variable, t, must be interpreted here as the value of the already-memorized "a + b" sub-expression. The compiler can keep it in a machine register.

9.2.3 Deleting Code

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

```
{
  int a, b;

  a = 1;
  b = a - 1;
  if (a < b)
     c = b;
  else
     c = a;
  c = c * 2;
}
```

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

```
{
  int a, b;

  a = 1;
  b = 0;
  if (1 < 0)
     c = 0;
  else
     c = 1;
  c = c * 2;
}
```

Deleting Useless Code

When the optimization functions evaluates the above example, it creates some useless code, a and b. Because a and b are local to the block, their assignment is not necessary. Deleting the useless code results in the following:

```
{
  int a, b;

  if (1 < 0)
     c = 0;
  else
     c = 1;
  c = c * 2;
}
```

Deleting Inaccessible Code

Managing and developing constant expressions can also reveal some inaccessible code. The previous example, which shows this, is reduced to the following:

```
{
  int a, b;

  c = 1;
  c = c * 2;
}
```

9.2.4 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)
   {
      x = a + b;
      a = u;
   }
else
   {
      x = a + b;
      b = v;
   }
```

This yields the following after optimization:

```
x = a + b

if (u > v)
   a = u;
else
   b = v;
```

The following is an example of temporization:

```
if (u > v)
   {
      a = u;
      x = a + b;
   }
else
   {
      b = v;
      x = a + b;
   }
```

This yields the following after optimization:

```
if (u > v)
   a = v;
else
   b = v;
x = a + b
```

9.2.5 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)
   x = a + b;
else
   a = 1;
x = a + b;
```

In the example above, the assignment 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)
   x = a + b;
else
   {
     a = 1;
     x = a + b;
   }
```

9.2.6 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:

```
for (i = 1; i <= 10; i++)
  x [i] = a + b;
```

❑

EXAMPLE 2:

```
for (i = 1; i <= j; i++)
  x [i] = a + b;
```

❑

EXAMPLE 3:

```
for (i = 1; i <= 10; i++)
  if (y > 0)
   x [i] = sqrt (y);
```

The optimization function evaluates all the invariant expressions outside of the loop. This transformation is successful only when the evaluation takes place in the same path where it was performed 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;
for (i = 1; i <= 10; i++)
  x [i] = t;
```

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 in order to allow the optimization function to remove the loop invariant.

Example 2, from above, after rearrangement:

```
if (j > 1)
  for (i = 1; i <=j; i++)
   x [i] = a + b;
```

Example 2, after optimization:

```
if (j > 1)
  {
   t = a + b;
   for (i = 1; i <= j; i++)
     x [i] = t;
}
```

❑

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.

9.2.7 Strength Reduction and Processing Loop Control Variables

9.2.7.1 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, and has 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
```

The product of k * c is evaluated at compile time.

An example of this optimization function is:

```
for (i = 1; i <= 10; i += 2)
  x = x + 4 * i;
```

After optimization:

```
t = -4;
for (i = 1; i <= 10; i += 2)
  {
   t = t + 8;
   x = x + t;
}
```

9.2.7.2 Processing of Loop Control Variables

If the compiler can know the number of iterations of a loop, the loop control test is substituted by an equivalent one and is expressed using one of the intermediary variables that the strength reduction function created.

The example from above (after the strength reduction) can be reformulated by hand 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)
    go to 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)
    go to 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)
    go to lab;
```

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

```
for (i = 1; i <= 25; i++)
  {
    k = 25 * (i - 1);
    for (j = 1; j <= 25; j++)
      x [k + j] = j;
}
```

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:

```
for (i = 1; i <= 25; i++)
  {
     k = 25 * (i - 1)
     for (j = 1; j <= 25;)
       {
           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;
   }
}
```

This program can then be optimized using the algorithms described above.

9.2.9 Procedure Merging

The optimization function of in-line insertion substitutes the calls to procedures or functions with their corresponding code after arguments replace formal parameters. This speeds the program execution time, by both deleting the call sequence and augmenting the window on which to perform the optimization.

Generally speaking, when a call to a procedure is inserted depends on the user options and the optimizer criteria. The users options are on the OPTIMIZE level, the INLINER parameter, and pragmas.

It is possible to merge some calls to a procedure while others not. The decision of when to merge procedure is based upon the following criteria.

- A recursive call is not inserted if a procedure is directly or indirectly recursive. There is only one level of insertion, shown in the example below.
- The procedure size will remain reasonable after the procedure merging.

AN EXAMPLE OF MERGING:

In this example, a file contains the following functions:

```
    f()   { a=1;       g(); b=2;}
    g()   { c=3;       f(); d=4;}
main ()   { f();}
```

After procedure merging, these functions are equivalent to the following:

```
    f()   { a=1;       g(); b=2;}
    g()   { c=3;       f(); d=4;}
main ()   { a=1; c=3; f(); d=4; b=2;}
```

In this example, the second call to f is not inserted. This second call comes from the g that is inserted.

❑

9.3 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, and it is automatically switched on. 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 merged 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.

10. Run-Time Environment

10.1 RUN-TIME Header Subfiles

Run-time functions may need to use some special header subfiles contained in the system library SYS.C.INCLUDE. The SYS.C.INCLUDE subfiles used for run-time functions are:

<STDIO_H>	See section 12, 13, 14, 16, 18, 22, and 24
<SETJMP_H>	See section 17
<ASSERT_H>	See section 21
<CTYPE_H>	See section 15
<STDLIB_H>	See section 14, 22, and 24
<STRING_H>	See section 16
<MATH_H>	See section 18
<TIME_H>	See section 19 and 24
<STDARG_H>	See section 13 and 20
<ERRNO_H>	See section 23
<LOCALE_H>	See section 24
<SIGNAL_H>	See section 22
<STDDEF_H>	See section 25

10.2 Accessing Run-time Functions

The standard header subfiles listed above contain a list of #define statements. These #define statements get the actual entry point of the run time package from the usual name of the function. If the corresponding header is not included, the name is not replaced. The entry point of the run-time package is then called through a compile relay that is linked automatically to the user's program. In order to prevent renaming, add the following statement:

```
#undef <function name>
```

If the header file is included and you want to have your own implementation of one of the standard library functions, adding this statement prevents renaming.

NOTES:

1. Because direct access to the run-time package is more efficient, it is best to include the header file.
2. If the header file is included, the compiler can automatically insert the code of some library functions in your own code, which improves efficiency.
3. If you use the BUILTIN option of the C compiler, you cannot redefine the library function, even by using #undef. For more information, see section 3.1.2.19 and 3.2.2.26.

The current release does not support the _RTP_NO_CU_RELAIS macro.

10.3 Run Time Initialization

A C program consists of a set of functions, one of which is known as the main function and called first. The main function automatically calls the C Run Time package before the first executable statement in order to initialize its own working area. In the same way, the main function calls the C Run Time package after its last executable statement, in order to flush buffers and close files (except when an explicit return statement is executed).

Nevertheless, it is possible to initialize the C Run Time package without going through the main function, by invoking explicitly the H_CLR_EPROLOG function. The H_CLR_EPILOG function must be used to clear the Run Time working area at the end of the program. The respective prototypes are:

- * EXTERN VOID H_CLR_EPROLOG ():
- * EXTERN VOID H_CLR_EPILOG ():

The respective prototypes are:

* extern void H_CLR_EPROLOG (int)
* extern void H_CLR_EPILOG ():

and H_CLR_EPROLOG must be called with a parameter value of 0 (zero).

Subroutines written in other languages supported by GCOS 7 can call these functions For instance, in GPL:

```
DCL H_CLR_EPROLOG ENTRY (FIXED BIN(31));
...
CALL H_CLR_EPROLOG(0)
```

10.4 Portability Levels of the Run-time Functions

Each of the Run Time macros or functions has an associated portability level. The synopsis reflects this level. The three portability levels and their definition levels are as follows:

ANSI	Part of the X3J11 ANSI standard
XOPEN	Part of the XPG3 XOPEN standard
GCOS 7	Specific to the GCOS 7 library

The ANSI level functions are the most portable because they can be used at the ANSI level, the XOPEN level, and the GCOS 7 level. Whereas the XOPEN functions can be used on the XOPEN level and the GCOS 7 level, and the GCOS 7 level functions can be used only on the GCOS 7 level. For example, abs is a function that is part of ANSI, XOPEN and the GCOS 7 library. Then, its attached level is ANSI.

The GCOS 7 compiler can conform to the ANSI standard.

10.4.1 The ANSI Level Functions

The following is a list of the ANSI level functions. These functions are the most portable and can be used at the ASNI level, the XOPEN level, and the GCOS 7 level.

abort	ANSI
abs	ANSI
acos	ANSI
asctime	ANSI
asin	ANSI
assert	ANSI
atan	ANSI
atan2	ANSI
atexit	ANSI
atoi	ANSI
atol	ANSI
bsearch	ANSI
calloc	ANSI
ceil	ANSI
clearerr	ANSI
clock	ANSI
close	ANSI
cos	ANSI
cosh	ANSI
difftime	ANSI
div	ANSI

exit	ANSI
exp	ANSI
fabs	ANSI
fclose	ANSI
feof	ANSI
ferror	ANSI
fflush	ANSI
fgetc	ANSI
fgetpos	ANSI
fgets	ANSI
floor	ANSI
fmod	ANSI
fopen	ANSI
fprintf	ANSI
fputc	ANSI
fputs	ANSI
fread	ANSI
free	ANSI
freopen	ANSI
frexp	ANSI
fscanf	ANSI
fseek	ANSI
fsetpos	ANSI
ftell	ANSI
fwrite	ANSI
getc	ANSI
getchar	ANSI
getenv	ANSI
gets	ANSI
getw	ANSI
gmtime	ANSI
isalnum	ANSI
isalpha	ANSI
iscntrl	ANSI
isdigit	ANSI
isgraph	ANSI
islower	ANSI
isprint	ANSI
ispunct	ANSI
isspace	ANSI
isupper	ANSI
isxdigit	ANSI
labs	ANSI
ldexp	ANSI
ldiv	ANSI
localeconv	ANSI
localtime	ANSI
log	ANSI
log10	ANSI
longjmp	ANSI
malloc	ANSI

mblen	ANSI
mbstowcs	ANSI
mbtowc	ANSI
memchr	ANSI
memcmp	ANSI
memcpy	ANSI
memmove	ANSI
memset	ANSI
mktime	ANSI
modf	ANSI
perror	ANSI
pow	ANSI
printf	ANSI
putc	ANSI
putchar	ANSI
puts	ANSI
putw	ANSI
qsort	ANSI
raise	ANSI
rand	ANSI
realloc	ANSI
remove	ANSI
rename	ANSI
rewind	ANSI
scanf	ANSI
setbuf	ANSI
setjmp	ANSI
setlocale	ANSI
setvbuf	ANSI
signal	ANSI
sin	ANSI
sinh	ANSI
sprintf	ANSI
sqrt	ANSI
srand	ANSI
sscanf	ANSI
strcat	ANSI
strchr	ANSI
strcmp	ANSI
strcoll	ANSI
strcpy	ANSI
strcspn	ANSI
strerror	ANSI
strftime	ANSI
strlen	ANSI
strncat	ANSI
strncmp	ANSI
strncpy	ANSI
strpbrk	ANSI
strrchr	ANSI
strspn	ANSI

strstr	ANSI
strtod	ANSI
strtok	ANSI
strtol	ANSI
strtoul	ANSI
strxfrm	ANSI
subbuf	ANSI
system	ANSI
tan	ANSI
tanh	ANSI
time	ANSI
tmpfile	ANSI
tmpnam	ANSI
tolower	ANSI
toupper	ANSI
ungetc	ANSI
va_arg	ANSI
va_end	ANSI
va_start	ANSI
vfprintf	ANSI
vprintf	ANSI
vsprintf	ANSI
wcstombs	ANSI
wctomb	ANSI

10.4.2 XOPEN Level Functions

This is a list of the XOPEN level functions. These functions can be used at both the XOPEN and GCOS 7 levels.

_tolower	XOPEN
_toupper	XOPEN
close	XOPEN
cmpstr	XOPEN
cpystr	XOPEN
creat	XOPEN
fcvt	XOPEN
gcvt	XOPEN
index	XOPEN
log2	XOPEN
lseek	XOPEN
notstr	XOPEN
open	XOPEN
prefix	XOPEN
read	XOPEN
scnstr	XOPEN
substr	XOPEN
write	XOPEN

10.4.3 GCOS 7 Level Functions

The following is a list of the GCOS 7 level functions. These functions are the least portable and can be used at only the GCOS 7 level.

begin_c_task	GCOS 7
cancel_multibyte_mode	GCOS 7
cancel_record_mode	GCOS 7
cancel_silent_mode	GCOS 7
cancel_ssf_fmt	GCOS 7
cmpbuf	GCOS 7
cpybuf	GCOS 7
ecvt	GCOS 7
etof	GCOS 7
etoi	GCOS 7
etol	GCOS 7
fill	GCOS 7
h_reopen	GCOS 7
lenstr	GCOS 7
lock	GCOS 7
notbuf	GCOS 7
scnbuf	GCOS 7
set_multibyte_mode	GCOS 7
set_record_mode	GCOS 7
set_silent_mode	GCOS 7
set_ssf_fmt	GCOS 7
setprompt	GCOS 7
settsp	GCOS 7
sexit	GCOS 7
test_multibyte_mode	GCOS 7
test_silent_mode	GCOS 7
unlock	GCOS 7

11. General I/O Considerations

11.1 C Files and GCOS 7 Files

This subsection describes the C files and the GCOS 7 files that work with the I/O functions.

11.1.1 C Files

A C file is a simple logical entity called a stream. It is associated at open time with an external file name, which in turn represents the physical entity of a file. The physical entities include disk and terminal files. When a pointer points to a structure of FILE type, the programmer can "see" this stream. Most of the I/O functions reference this pointer as an argument.

The standard inclusion file STDIO_H defines the FILE type. It holds information to control I/O operations. The ANSI standard requires at least the following information:

- A file position indicator
- A pointer to the associated buffer
- An error indicator
- An end-of-file indicator.

The structure of a C file can be either one of the following:

- A set of lines, with each one composed of printable characters and ending with a newline character.
- A sequence of characters that are in the execution character set.

The smallest unit of a C file is the character.

11.1.2 GCOS 7 Files

A GCOS 7 file is a system object that its external file name (EFN) accesses. This name is used as constant character string for the first argument of the fopen function. The syntax of this name is described in the appendix about file and volume syntax.

I/O operations that work on a GCOS 7 file associate this EFN with an internal file name (IFN). JCL or GCL commands use the IFN to assign a file outside a source program. In C/GCOS 7 library implementation, the IFN is a character string. The IFN values of the standard stdin, stdout and stderr files are respectively STDIN,STDOUT and STDERR. Otherwise, the character string represents numerical values from 3 to 255.

C/GCOS 7 libraries accept only files that the following methods can manage:

- UFAS sequential or relative access method.
- BFAS sequential, DMU access method.
- The terminal standard access method.

The structure of a GCOS 7 file is a set of records with fixed or variable size. These can be accessed sequentially, directly, or with keys, depending on the organization of the file. The smallest unit of a GCOS 7 file is the record. Note that the V6 release does not support keys on BFAS files.

The character set for GCOS 7 is EBCDIC.

11.2 Stream Types, Data Formats, and Modes

11.2.1 Text and Binary Streams

The C ANSI standard uses text streams and binary streams to separate user data. Some processors, for example, editors, can then treat user data as text, and other data as binary. This process allows any system to support a C file in the same way.

What is a Text Stream?

A text stream is a logical mapping of a GCOS 7 file to a set of lines. Each line contains zero or more characters and terminates on a newline character. Under C/GCOS 7, the last line of a text stream associated with a file always has a newline character at the end, except for an SSF file in which no explicit newline has been written on that line.

In C/GCOS 7, data read from a text stream is equivalent to data written earlier to that text stream if the data meets the following conditions that ANSI imposes:

- The data consist of only printable characters, the horizontal tab and newline control characters.
- No newline character is immediately preceded by space characters.

These conditions assume that the applications handling text streams are portable.

Depending on the GCOS 7 file with which the stream is associated, the following exceptions apply to the above ANSI conditions:

- For variable-size record files, the newline character can be preceded by space characters. However, for fixed-size record files, padding space characters preceding a newline character can be written temporarily or permanently in a record. It is not possible to distinguish these space characters preceding a new line from those to be written out.
- Characters that are not printable can be written out and read in from a text stream. Text streams can be associated with GCOS 7 files whose contents are actually binary data.

What is a Binary Stream?

A binary stream is a logical mapping of a GCOS 7 file to an ordered sequence of any characters.

In C/GCOS 7, binary streams must conform to the following condition that ANSI applies:

Data read in from a binary stream is equivalent to the data written out earlier to that stream.

Fixed-size record GCOS 7 files can have, at most, RECSIZE null characters appended. (Where RECSIZE is the size of a record.)

IMPORTANT:
If binary streams end with null characters, due to the null character padding, the behavior of fseek or fread functions is undefined when using Fixed-size record GCOS 7 files.

A binary stream may be associated with a GCOS 7 file whose contents are in fact text data.

EXAMPLE OF BINARY STREAM USAGE

When a GCOS 7 file contains binary data, it can be useful to interpret EBCDIC value 15H as the corresponding opcode in the instruction set instead of as the newline character.
❑

11.2.2 SSF and SARF Formats

The SSF and SARF formats are GCOS 7 standard formats. The SSF format records user data mixed with information for specific processors, for example, editors. The SARF format records only user data.

The SSF format is not a specific property of a file seen as a system object. It is a convention between GCOS 7 processors that handles user data contained in a file.

The following C/GCOS 7 macros set the SSF and SARF formats:

```
set_ssf_fmt
cancel_ssf_fmt
```

SSF or SARF formats are associated with the GCOS 7 files. They are independent of the text or binary stream associated with this file. It is possible to associate a

text stream with a file in SARF format, even if, conceptually, a binary stream is the natural mapping for such a file.

IMPORTANT:

Writing operations on a file with SSF format use internal SSF information. The coherence of a file can be destroyed if it is as input to other GCOS 7 processors. This is especially true for specific commands of editors, if the file can be edited.

11.2.3 Line-Record and Stream-Mode Files

The line record and stream modes separate GCOS 7 files into two categories: text files and other files. These modes were created before the ANSI standard, and are now part of C/GCOS 7 file processing in order to be compatible with previous releases.

The line-record mode is used mainly to allow GCOS 7 editors to read a C file created in this mode. It assumes that each file record is a C line. The end of the record is considered to represent the ending newline character of the line. By extension, this mode can be set (and have the same semantics) for files that the editors cannot handle. Files that are not in line-record mode are in stream mode.

The line record and stream modes characterize the manner in which a GCOS 7 file is seen by the application and by the other processors in the GCOS 7 environment. Because they take into account that newlines, which separate text lines, are not visible by a GCOS 7 standard editor, they try to map a line on a record. The record is the smallest GCOS 7 file unit.

UNIX prefers to map a file in the STREAM mode. It physically writes all characters in a file through C primitives. This includes newline characters.

The following C/GCOS 7 macros set these modes:

```
set_record_mode
cancel_record_mode.
```

LINE_RECORD mode implies that the stream associated with a file is a text stream, even if a binary stream was requested at open file time.

IMPORTANT:
If a written line is larger than the current size record, the behavior after LINE_RECORD is undefined when the file is read in with the same mode.

11.2.4 Buffering

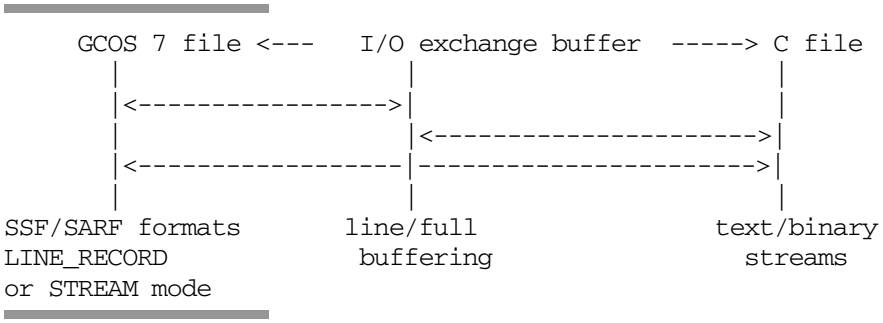
A buffer is a resource that is attached to the stream associated with a GCOS 7 file. The buffer manages the I/O operations. This buffer is a kind of window through which data is exchanged between the stream and the file. The buffering characteristics of a stream control the data flow transmitted to or from the file.

The definitions for stream buffering are as follows:

- A no-buffering or unbuffered stream is ready to receive characters directly from the source or to send characters directly to the destination as soon as possible.
- A fully-buffered stream accumulate and transmits characters to or from the file as a block when the buffer is filled.
- A line-buffered stream accumulates and transmits characters to or from the file as a block when a newline character is encountered.

C/GCOS 7 does not support unbuffered streams. This is because the smallest I/O exchange unit for a file is the record. C and GCOS 7 do support line or full buffering, even when having to bypass buffering characteristics. This bypassing allows the I/O control flow to adapt between a GCOS 7 file and its associated stream. For more information, see the overriding rules, described below.

The following figure summarizes the different notions seen above. Vertical lines show which properties are associated with which objects (above the vertical line). Horizontal lines show how objects are linked both physically and in terms of data exchange during I/O operations.



11.2.5 Default Positioning on GCOS 7

This subsection describes how the SSF/SARF format and LINE_RECORD/STREAM types are set, and the buffering attribute of a file.

At open time, only the SARF/SSF format can be explicitly specified. If it is not specified, it takes the default setting. The LINE_RECORD/STREAM type and buffering attribute take the default settings.

In a second step, the user can override these settings between open time and the first invocation of an I/O library function. This can be done as follows:

- The following two non-ANSI macros can set or reset the SARF/SSF format:

```
set_ssf_fmt
cancel_ssf_fmt
```

- The following two non-ANSI macros can explicitly set or reset the LINE-RECORD or STREAM modes:

```
set_record_mode
cancel_record_mode
```

- The following two ANSI primitives can modify line-buffering or full buffering, as follows:

```
setbuf
setvbuf
```

Finally, the implemented strategy of C/GCOS 7 library functions overrides these settings, as follows:

Terminal files:

- Are always associated with a text stream
- Have SARF format
- Have LINE_RECORD mode.

SYSOUT/SYSIN files:

- Are always associated with a text stream
- Have SSF format
- Have LINE_RECORD mode.

Disk files:

- Have the SARF format set if the file is not UFAS sequential, or BFAS sequential.
- Are always associated with a text stream if the LINE_RECORD is set.

All supporting files are associated with a fully buffered stream when both the LINE-RECORD mode and SSF format are not set, and the file does not have variable-size records.

The open and create functions establish the following default settings:

- A stream associated with a GCOS 7 terminal file uses the SARF format, LINE_RECORD mode, and line buffering.
- A stream associated with a SYSOUT/SYSIN file uses the SSF format, LINE-RECORD mode, and line buffering stream.
- Other files with fixed size records use the SARF format, STREAM mode, and full buffering stream.
- Subfiles with variable size records use the SSF format, LINE-RECORD mode, and line buffering stream.
- Other files with variable size records use the SARF format, STREAM mode, and full buffering stream.

The low-level functions (open and create) open a file as if it were associated with a binary stream and the default settings.

11.3 Standard Files

During execution of a C program (the entry point of which is a 'main' function), the run-time package (RTP) opens three files called standard files. These files can be used as such until the user 'main' function is completely executed unless they have been closed by the close or fclose functions.

Description:

- Names of pointers on FILE structure: stdin, stdout, stderr
- Names of IFNs (for external assignment): STDIN, STDOUT, STDERR
- Numbers of file descriptors for accessing them with low-level primitives: 0,1,2 for stdin, stdout, stderr respectively. Low-level primitives are equivalent to system functions under UNIX.

According to usage mode:

- IOF:
  - Not statically assigned: the standard file is a terminal file handled by the Terminal Access Method (TAM). For more information, see the paragraph on files handled by TAM.
  - Statically assigned: this is the assigned GCOS 7 file.
- Batch:
  - Not statically assigned: for stdout and stderr, it is the SYSOUT. The stdin file is a dummy file. The 'dummy' is not the DUMMY type associated with certain files under GCOS, although it does simulate their functions.
  - Statically assigned: this is the assigned GCOS 7 file.

IMPORTANT:
These files can be opened only when the 'main' function is called.

11.4 Non-standard Files

The following functions can open or create a file:

fopen
freopen
tmpfile
open
create

The following rules apply when a file does not already exist and when the used function allows the creation of the file.

- For a library subfile, a subfile has the same physical attributes as the library file.
- For a temporary file, the file has the standard characteristics unless the tmpfile function creates it. If the tmpfile creates a temporary file, it has the standard characteristics and the $TEMPRY attribute of a temporary GCOS 7 file. The standard characteristics are described below.
- A terminal file has the attributes described in the terminal I/O paragraph, later in this section.

A standard C file has the following characteristics:

organization type:	UFAS
data format:	SARF
record size:	256
cisize:	2048
record type:	FIXED BLOCKED
allocated space:	1 cylinder
increment:	1

The creation of a file by the RTP implies that space be allocated for the GCOS 7 object. The user is responsible for the file that is thus created and must be capable of retrieving it to destroy it if necessary. It is recommended that you use the attribute $CAT for cataloging such a file or the attribute $TEMPRY for creating a temporary one. By default, it is created RESIDENT with a warning message sent to the JOR.

The FILE structure that can be accessed by the user for opening a file with fopen is initialized on the first invocation of one of the I/O functions, so that:

- The size of the buffer associated with the file, its write index and number of characters remaining to be read (depending on the opening mode) can be consulted.
- The macros described in STDIO_H can consult or manipulate the position of different flags with respect to the state of the file.
- The buffer address.
- Specific functions can set or manage a position indicator, but its use is prohibited elsewhere. The specific functions include fseek and ftell. The position indicator gives:
  - The current byte position for a binary stream
  - The current line and column in the line for a text file

Files assigned or pre-allocated by JCL or GCL are subject to the restrictions of the type of organization selected by the user.

When a file is created it is advisable that you specify the private catalog under which the file is to be associated. This is to avoid any problems with access rights; and also to enable you to find again the name of the MEDIA and the DEVICE.

Trying to create a sub-file in a non-existent library is forbidden. Creating OBJ1.OBJ2. OBJN..FILE1, if one of the OBJ does not exist, actually creates a system object OBJ1.OBJ2. OBJN, or no object at all if any problems with access rights occur.

11.5 Terminal I/O

The RTP processes only the part in text mode supplied by the communications interfaces that can be used with synchronous and asynchronous terminals during a session under IOF.

A typical feature of this mode is that a data item is considered as a continuous stream of information. This mode processes only the simplest communications interface, i.e. the one accessible to a user with no previous experience with the type of manipulation performed on his file. This is the SARF data format.

IMPORTANT:

A terminal I/O is always line buffered. For example, this means that a `putc (c);` statement displays that character `c` at the terminal only when a newline character `\n` is output.

Only the following characteristics for terminal handling are supported:

- A file can be statically assigned to a terminal using the parameter DVC=TN of the JCL command: ASSIGN (or the file literal::TN for GCL).
- A program can connect several IFNs to the same terminal.
- Three standard files are available to the user in IOF. The prefix I:b indicates data on the STDIN file (where b denotes a blank character) and 0:bbb indicates data on the files STDOUT and STDERR (where bbb denotes three blank characters). For non standard files, the file descriptor number returned by the RTP is the prefix.
- The transparent mode of the standard terminal access method is not supported. For more information about how to use the transparent mode on terminals, refer to the setprompt and setrsp functions.
- The JCL command DEFINE is the responsibility of the user.
- In read mode, the end of file is marked by EOS <CR> at the beginning of the line (<CR>= carriage return).
- Opening of a terminal file is valid only in write or read modes.

11.6 Static Assignment of C Files

Static assignment of a C file under GCOS 7 requires an EFN and an IFN linked to the file.

The IFN is created by the RTP when the file is opened. Except for the three pre-defined files, it is best to use the IFN extension in the mode parameter of the fopen functions. The IFN is stdin, stdout, and stderr (upper case) for the three pre-defined files. For the other files, IFN is a character string that represents a numerical value in the range of 3 to 255.

Static assignment is not a standard feature of input/output operations of C. It is merely a facility added by the RTP to simulate re-direction of the file as under UNIX. Consequently, the use of this facility is your own responsibility.

EXAMPLE IN JCL:

```
   .
   .
step TOTO TEMP;
ASG 3 LSFY.CC.USER SUBFILE = FIC0;
ASG 4 LSFY.CC.USER SUBFILE = FIC1;
   .
   .
   .
END STEP;
   .
```

❑

EXAMPLE IN THE C PROGRAM:

```
#include <stdio_h>
 main()
   FILE * p0, * p1;
    .
    .
    .
   p0 = fopen ("BIDON$TEMPRY", "w;IFN=3"); /* first opening */
   p1 = fopen ("TEMPO$TEMPRY", "w;IFN=4"); /* second opening */
    .
    .
    .
```

❑

The IFN of BIDON$TEMPRY is 3 and the IFN of TEMPO$TEMPRY is 4.

NOTE:

In the case of static assignment, the efn given in the fopen function has no effect on the first call.

IMPORTANT:

It is possible that after closing the file to which P1 points, the statement p1 = fopen ("TEMPO$TEMPRY","w") does not re-open the redirected file (FIC1), but re-opens the file designated by the GCOS 7 file name supplied by the primitive fopen, i.e., TEMPO$TEMPRY (temporary file). The C program execution opens the subfile FIC0 for the file to which PO points and the subfile FIC1 for the file to which P1 points. If the JCL does not contain ASG 3... and ASG 4..., the opened files are BIDON$TEMPRY and TEMPO$TEMPRY, respectively. If another file is already using an IFN, the opening request is rejected.

Closing a file automatically deassigns it. Any failure in opening a file can make the IFN of another statically assigned file unpredictable.

For standard files, the following are examples in GCL:

```
exec-pg toto file1 = STDIN  asg1 = mylib..input
            file2 = STDOUT asg2 = mylib..output
            file3 = STDERR asg3 = mylib..error;
```

In the above case, the three standard files are re-directed onto three library subfiles.

11.7 Direct Access

Accessing a file at a given position or finding the current position of a file are called seek actions. The fseek, lseek, rewind and fsetpos functions access a file directly. The ftell and fgetpos functions give the current position in a file. However, the SYSOUT files, SYSIN files, and terminals do not support these seek actions.

In files that do support seek actions, the stream contains an available file position indicator. The stdio_h header file defines the fpos_t type that describes this indicator.

Text streams support seek actions with the following conditions:

- The file position indicator holds a record identification and the displacement in this record.
- Direct access on a stream is possible only from the beginning of the file and at a position that the ftell or fgetpos functions have already returned. The first position in the file is zero and can be accessed by the following call, in which p is the pointer on the stream associated with the file:

```
fseek (p,0,0);
```

Binary streams support seek actions with the following conditions:

- The file position indicator holds a byte offset position relative to the beginning of the associated file.
- For direct access on a stream, the value of a byte offset can be added to the beginning of the file, the current position of the file, or the end of the file that is associated with that stream.

When the append mode opens a text or binary stream, the position indicator is at the end of file. Therefore, the stream supports seek actions, but data is still written. Therefore, any actions are not very useful.

For empty files, a file position is possible at the beginning of file.

When the update mode opens a stream, the action is either reading or writing following repositioning.

The fgetpos and fsetpos functions directly access very large files. The ftell and fseek functions do not always successfully access files.

11.8 GCOS 7 Specific Features

11.8.1 Extensions of the Open Mode

As an argument of the fopen function, the open processing mode extends itself to accept two types of information about the file to be opened. The first is the file type (SSF/SARF), and the second is the IFN.

11.8.1.1 File Type (SSF/SARF)

The user indicates the SARF or SSF type of records in the fopen command. In the following example, SARF is in an expression specifying that there are no special records or headers added to the data that are recorded in the my_file file.

```
FILE *p;
    p = fopen ("my_file", "w;SARF");
```

In the following example, SSF is in an expression specifying that the re-written text_file contains some special information that the GCOS 7 editors need.

```
FILE *p;
   p = fopen ("my_sl_lib...text_file", "r;SARF");
```

The example below shows that the user wants to read a text file that includes special information, such as special records and header records. A GCOS 7 editor can modify the text file.

Keywords like SSF, SARF, and IFN must be in capital letters.

11.8.1.2 Examples

In all the following examples, the function that copies files record by record is called my_func_copy. The fgets and fputs functions implement this function.

Copying an SSF File to an SSF File Text Member of a Library

```
p = fopen (file1,"r;SSF");
```

or:

```
p = fopen (file1,"r");
set_ssf_fmt (file1);
set_record_mode (file1);       /* can be omitted if file1 has
                                  variable-sized records */

p = fopen (file2,"w;SSF");
```

or:

```
p = fopen (file2,"w"); set_ssf_fmt (file2);
set_record_mode (file2);        /*same remark as above */
my_func_copy (FROM_FILE1,TO_FILE2);
```

All lines of file2 have the value zero for their line number. GCOS 7 editors can edit this file.

Copying an SSF File to an SARF File Text Member of a Library

```
p = fopen (file1,"r;SSF");
```

or:

```
p = fopen (file1,"r"); set_ssf_fmt (file1);
set_record_mode (file1);           /* same remark as above */
p = fopen (file2,"w;SARF");
```

or:

```
p = fopen (file2,"w");cancel_ssf_fmt (file2);
set_record_mode (file2);           /* same remark as above */
```

File2 cannot be edited by GCOS 7 editors.

Copying an SSF File (Seen as an SARF File) to an SARF File Text Member of a Library

```
p = fopen (file1,"r;SARF);
p = fopen (file2,"w;SARF);
```

or:

```
p = fopen (file2,"w"); cancel_ssf_fmt (file2);
set_record_mode (file1);           /* same remark as above */
set_record_mode (file2);           /* same remark as above */
my_func_copy (FROM_FILE1, TO_FILE2);
```

GCOS 7 editors can edit file2. After this function, file2 is a real SSF file. The line numbers in file2 respect the numbering of those in file1.

Copying an SARF File to an SSF File Text Member of a Library

```
p = fopen (file1,"r;SARF);
p = fopen (file2,"w;SARF);
```

or:

```
p = fopen (file2,"w"); set_ssf_fmt (file2);
set_record_mode (file1);           /* same remark as above */
set_record_mode (file2);           /* same remark as above */
my_func_copy (FROM_FILE1, TO_FILE2);
```

GCOS 7 editors can edit file2. All lines of file2 have the value zero for their line number.

11.8.1.3 Giving the IFN

You can specify the internal file name (IFN) to attach to the file. This is useful with static assignment of a file, either in the JCL or GCL procedures that start the user application.

As under Unix, the IFN is an integer in the range of 3..255. The first three values are reserved for stdin, stdout and, stderr.

11.8.1.4 Examples

```
FILE *p;
p = fopen ("my_file","w;SARF;IFN=15");
```

In this example, the user can reference my_file with the IFN of 15 to assign it to a file with another external file name that is in the JCL or GCL procedure. Note that fileno(p) returns the value 15 for this file. This is the only way to use a file that has been statically assigned, that is through GCL or JCL.

```
FILE *p;
p = fopen ("dummy", "r;IFN=7");
```

The program invocation is as follows (in GCL):

```
EXEC_PG MYPROG FILE1=7 ASG1=MY_FILE;
```

The efn given in the first fopen call has no effect and the static file assignment is used. Note that if the file is closed, the previous (static) assignment is lost. The file is closed using either fclose or freopen.

11.8.2 Access and Share Extensions

The user can specify the types of access and sharing that is assigned to a given GCOS 7 disk file at open time. The user does this upon the invocation of the fopen function, as follows:

```
 p = fopen ("myfile","r;ACCESS=WRITE;SHARE=ONEWRITE");
```

The file named myfile is opened for reading, but its assignation allows one writer and n readers, simultaneously.

For more information about access and sharing, refer to the specific documentation of UFAS disk files.

NOTE:

Update modes and positioning requests are not supported for terminal, SYSOUT, SYSIN, and tape files. For example, fseek is a positioning request.

12. File Processing

12.1 stdio_h Interface

The use of a standard input/output function and of any RTP function under GCOS 7 (except for functions requiring the header files <setjmp_h>, <assert_h>, <ctype_h>, <stdlib_h>, <string_h>, <math_h> or <time_h>) automatically requires "including" the header file <stdio_h> (see Section 8).

```
NAME    STDIO_H (in upper or lower case)
```

Synopsis

```
#include <stdio_h>   (or: #include <stdio.h>)
```

Description

- The high-level functions sue getc or putc. These functions include gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, and fwrite.
- A file and its associated buffer is called a stream and is declared as a pointer to a FILE TYPE.
- Fopen creates descriptive data for a stream and returns a pointer to designate the stream in further input/output operations. There are normally three streams open when a C 'main' function is activated. Their associated pointers have standard names:

stdin	standard input file
stdout	standard output file
stderr	standard error file

  For more information see the section on standard files.
- An EOF integer (=-1) is returned at end of file or upon an error by functions that handle streams. It is defined as a macro in the file STDIO_H.

The following functions are implemented as macros:

feof	fileno
ferror	set_record_mode
clearerr	cancel_record_mode
getchar	set_ssf_fmt
putchar,	cancel_ssf_fmt

Redefining these functions can be dangerous as the results are unpredictable.

The file <stdio_h> also defines a certain number of specific macros for the GCOS 7 system. Some reserved external variables are also declared for RTP use only.

12.2 Stream Status Macros

Name

```
feof, ferror, fileno, clearerr, set_record_mode,
cancel_record_mode, set_ssf_fmt, cancel_ssf_fmt
```

Synopsis

```
#include <stdio_h>

    feof (p)
    FILE *p;

    ferror (p)
    FILE *p;

    fileno (p)
    FILE *p;

    clearerr (p)
    FILE *p;

    set_record_mode (p)
    FILE *p;

    cancel_record_mode (p)
    FILE *p;

    set_ssf_fmt (p)
    FILE *p;

    cancel_ssf_fmt(p)
    FILE *p;

    set_silent_mode()

    cancel_silent_mode()
```

Description

feof	Returns a non-zero value when the end of file is reached on the file designated by p: otherwise returns zero.
ferror	Returns a non-zero value when read or write error occurs on the file designated by p or in the case of incorrect use of LINE-RECORD type: otherwise returns zero.
clearerr	Resets the read/write error flag of the file designated by p.
fileno	Returns the descriptor of the file designated by p.
set_record_mode	Informs the RTP that input/output on the file designated by p are performed in LINE_RECORD type. This mode has the characteristic that - a line C (i.e., a character string ending with a newline) is considered as a file record. You must therefore know the record size of your file, which you can find by consulting the field bufsize of the FILE structure associated to the file. If it is greater than the written line C, spaces are added at the end of line. Conversely, if the record size is less than the line C that is to be written, the line is split over two records of the file. When the file is re-read, two lines are read instead of one. This macro must be used immediately before the first input/output operation is performed on the file.
cancel_record_mode	Indicates to the RTP that input/output operations on the file designated by p are performed in the STREAM type. This mode has the characteristic that a C file is considered as any sequence of bytes. A C line is written on the file as is, including the newline character. Interpreting the newline is the responsibility of the user. This macro must be used immediately before the first input/output operation is performed on the file.

set_ssf_fmt	Informs the RTP that input/output operations are performed in an SSF data format on the file designated by p. This macro must be used before the first input/output operations are performed on the file.
cancel_ssf_fmt	This indicates that the operations on the stream to which p points are done in SARF format. This macro must precede the first input/output operation on the associated file.
set_silent_mode	This macro suppresses the printing of standard messages.
cancel_silent_mode	This macro enables the printing of RTP error messages.

Diagnostics

The validity of the pointer included in the parameters when these macros are called is not checked.

12.3 Standard File Processing (High-Level Primitives)

The C language does not provide pre-defined functions for file processing. These form a package and as such make up a sub-set of the RTP. This section discusses implementing these functions on the GCOS 7 system. The two main primitives in the package are getc and putc. They are both used by most of the other primitives.

12.3.1 fopen

Synopsis

```
# include <STDIO_H>

FILE *fopen(const char *, const char *);
FILE * fopen (filename, type)

    char * filename, * type;
```

Description

The fopen function opens a file. The filename pointer points to the name string of the file and associates a stream with it. The argument type points to a string described formally as follows:

```
type ::= <mode> [ ';' <exts> ]
mode ::= r | r+ | rb | r+b | rb+
         | w | w+ | wb | w+b | wb+
         | a | a+ | ab | a+b | ab+
exts ::= { <data_fmt> | <ifn> | <access_or_share> } ';' <exts>
data_fmt :: = SARF | SSF
ifn ::= IFN=<xxx>
xxx ::= numerical value from 3 to 255
<access_or_share>::= ACCESS=<access_val> | SHARE=<share_val>
<access_val> ::= ALLREAD | SPREAD | SPWRITE | WRITE | READ |
                 RECOVERY
<share_val> ::= MONITOR | FREE | ONEWRITE | DIR | NORMAL
```

Each mode, data format, ifn, access and share value appears only once in the string.

The following is an example of the fopen string:

```
    p = fopen ("my_file","ab+;IFN=12;SARF");
```

The semantics of the mode values are:

r	Open text file for reading.
w	Truncate to zero length or create text file for writing.
a	Append, open, or create text file for writing at end of file.
rb	Open binary file for reading.
wb	Truncate to zero length or create binary file for writing.
ab	Append, open, or create binary file for writing at end of file.
r+	Open text file for update (reading and writing).
w+	Truncate to zero length or create text file for update.
a+	Append; open or create text file for update, writing at end of file.
r+b or rb+	Open binary file for update (reading and writing).
w+b or wb+	Truncate to zero length or create binary file for update.
a+b or ab+	Append; open or create binary file for update, writing at end of file.

The following table specifies the capabilities and actions associated with the various opening modes (Y=yes,-=NO). It is also available for modes concerning binary streams.

	r	w	a	r+	w+	a+
File must exist before open :	Y	-	-	Y	-	-
old file contents discarded on open :	-	Y	-	-	Y	-
stream can be read :	Y	-	-	Y	Y	Y
stream can be written :	-	Y	Y	Y	Y	Y
stream can be written only at end :	-	-	Y	-	-	Y

NOTES:

1. Opening a file with append modes forces all subsequent writes to the file to be made at the current end of file. This is not affected by intervening calls to the fseek function.
2. Opening a file with append modes may initially set the file position indicator for the stream beyond the data last written. This is because of null character padding. This applies only to fixed-size record GCOS 7 files.

3. Opening a file with update modes allows reading and writing actions on the associated stream. **However, input cannot directly follow output without an intervening call to the fflush, fseek, fsetpos, or rewind functions.**
4. **Output cannot directly follow input without an intervening call to fseek, fsetpos, or rewind functions, unless the input operation encounters the end of file.**
5. When opening a file, the error and end of file indicators for the stream are cleared.
6. For more information about full/line buffering, data format, mode or real stream associated with the opened file, refer to the section that describes default positioning on GCOS 7.
7. The append "a" mode does not support either the tape device or the SYSOUT files.

Diagnostics

The fopen function returns a pointer to the object controlling the stream. If the open operation fails, a null pointer is returned.

12.3.2 freopen

Synopsis

```
#include <stdio_h>

FILE *freopen(const char *, const char *, FILE *);
FILE * freopen (filename, mode, stream)

char *filename, *mode;

FILE *stream;
```

Description

The freopen function opens the file identified by filename. freopen then associates with that file the stream to which the stream parameter value points. The mode argument specifies the opening processing mode. This is the same as in the fopen function, except that a specified IFN is not taken in account.

First, the associated buffer is flushed, then freed. Then the file associated with stream is closed. Failure to close this file successfully is ignored. If the user cannot re-open the file, the error is signaled immediately. The error and end-of-file flags for the stream are both cleared.

Diagnostics

A null pointer is returned if the open operation fails. Otherwise, the value of the stream is returned.

12.3.3 h_reopen

Synopsis

```
#include <stdio_h>

FILE * h_reopen (stream , mode)

FILE *stream;

char * mode;
```

Description

The h_reopen function quickly re-opens a file that is in another processing mode. The h_reopen function uses the opening processing mode specified in the mode argument to re-open the file associated with stream.

The file is first closed without deassigning it. It is then opened with the new processing mode, the value of which is restricted to "r", "w" or "a". The SARF, SSF, IFN, access, and shared indications are not taken into account.

NOTE:

h_reopen does not apply to stdin, stdout, or stderr streams.

Diagnostics

A null pointer is returned if the re-open operation fails or if the stream is stdin,stdout or stderr. Otherwise, the value of stream is returned.

12.3.4 fclose

Synopsis

```
#include <STDIO_H>

int fclose(FILE *);
    fclose (stream)

    FILE * stream;
```

Description

The fclose function causes the stream pointed to by stream to be fflushed and the associated file to close. The stream is disassociated from the file. If the associated buffer is allocated automatically, it is freed. If the tmpfile function creates the file, it is removed automatically.

Diagnostics

Returns EOF (= -1 defined in <STDIO_H>) when there is an error.

NOTE:

The file is deassigned when closed so that any static assignment using JCL is no longer valid if it is re-opened.

12.3.5 fflush

Synopsis

```
#include <STDIO_H>

int fflush(FILE *);
    fflush (stream)

    FILE * stream;
```

Description

Empties the buffer associated with the file identified by stream. If the stream is a null pointer, the fflush function works on all output or update streams.

Diagnostics

Returns EOF when there is an error. Flushing a read-only stream is considered to be an error.

NOTE:

Stream pointer validity is not checked.

12.3.6 gets, fgets

Synopsis

```
#include <STDIO_H>

char *gets(char *);
    char * gets (s)

    char * s;

char *fgets(char *, int, FILE *);
    char * fgets (s,n,stream)

    char * s;

    FILE * stream;
```

Description

The gets function reads characters from the input stream pointed to by stdin. It reads them into the array pointed to by s, until the end of file is encountered or a newline character is read.

The gets function discards any newline character, and it writes a null character after the last character read into the array.

The fgets function reads at most n-1 characters from the stream pointed to by stream. It reads them into the array pointed to by s. No character is read after a newline character (which is transmitted to s) or after the end of file are encountered.

A null character is written after the last character read into the array.

Diagnostics

Both these functions return s, if successful. If end of file is encountered and no characters have been read into the array, its contents remains unchanged and a null pointer is returned.

If a read error occurs during the operation the array contents are indeterminate and a null pointer is returned. An example of this is if "we" operates on a write-only stream.

NOTE:

Unlike fgets, gets does not retain the newline character "\n".

12.3.7 getc, getchar, fgetc, getw

Synopsis

```
#include <STDIO_H>

int getc(FILE *);
    int getc (stream)

    FILE * stream;

int getchar(void);
int getchar ()

int fgetc(FILE *);
int fgetc (stream)

    FILE * stream;

int getw (stream)

    FILE * stream;
```

Description

The getc function is equivalent to fgetc function when a function call invokes it. If the getc function is invoked as a macro, its argument (stream) should never be an expression with side-effects, or else the behavior is undefined. When the getc function is invoked as a macro, either the stdio_h file has been included and getc has not been undefined or redefined, or parentheses do not surround the getc call.

The getchar function is identical to getc with stream=stdin.

The fgetc function obtains the next character, if it exists, as an unsigned char. The input stream pointed to by stream converts the unsigned char to an integer (which has no sign extension), and sets the file position indicator for the stream, when it is defined.

Fgetc invocation is always a call to a function. It is never a macro call replacement.

The getw function obtains the four next characters (in the same way as fgetc) from the stream pointed to by stream. This is in order to compose an integer-type object, which is obtained by concatenation of read characters.

Diagnostics

The following are results when this function is successful:

- The functions getc,getchar and fgetc return the value of read character as described in fgetc function.
- The function getw returns the value of the obtained object.

The following are the results when this function is not successful:

- If the end of file is encountered, the end of file indicator for the stream is set and the EOF value is returned
- if a read error occurs, the error indicator for the stream is set and the EOF value is returned.

IMPORTANT:
The pointer stream that identifies the file is not tested.

12.3.8 ungetc

Synopsis

```
#include <SDTIO_H>

int ungetc(int, FILE *);
    ungetc (c, stream)

     char c;

    FILE * stream
```

Description

The ungetc function pushes the character specified by c back onto the input stream pointed to by stream. The character specified by c is converted to an unsigned character. Subsequent reads on that stream return the pushed-back characters in the reverse order of their pushing.

Only one character of pushback is guaranteed. Too many calls to ungetc function on the same stream can fail if they are without an intervening read or file positioning operation.

A successful intervening call to fseek, fsetpos, or rewind functions on that stream, discards any pushed-back characters for that stream.

Pushed-back characters never change the contents of the external storage corresponding to the file associated with the stream.

If the value of c argument equals that of the macro EOF, the operation fails and the input stream is unchanged.

If the ungetc function is successful, it clears the end of file indicator for the stream.

The value of the file indicator for the stream after reading or discarding all pushed-back characters is the same as before the characters were pushed back. However, for a text stream, this indicator is unspecified until all pushed-back characters are read or discarded. For a binary stream, it is decremented by each successful call to the ungetc function. If its value was zero, it is undefined after the call.

Diagnostics

The ungetc function returns the converted argument c. If the operation fails, it returns EOF. For example, either c is EOF or else it does not read the stream.

The validity of argument stream is not checked.

12.3.9 puts, fputs

Synopsis

```
#include <STDIO_H>

int puts(const char *);
    char * s;

    puts (s)

int fputs(const char *, FILE *);
fputs (s, stream)

    char * s;

    FILE * stream;
```

Description

The puts function writes the string pointed to by s onto the standard stdout stream. It also appends a newline character ('\n').

The function fputs writes the string pointed to by s to the stream pointed to by stream.

For both functions,the null terminating character is not written.

Diagnostics

Both functions return EOF if a write error occurs. If not, a non-negative value is returned.

NOTE:

Puts appends the character newline; fputs does not. Stream is not checked.

12.3.10 putc, putchar, fputc, putw

Synopsis

```
#include <STDIO_H>

int putc(int, FILE *);
  int putc (c,stream)

      char c;

      FILE * stream;

int putchar(int);
  putchar (c)

int fputc(int, FILE *);
  FILE * stream;
  fputc (c,stream)

  FILE * stream;
  putw (w,stream)
```

Description

The putc function is equivalent to fputc function when a function call invokes it. If the putc function is invoked as a macro, its argument (stream) should never be an expression with side-effects, or else the behavior is undefined. When the putc function is invoked as a macro, either the stdio_h file has been included and putc has not been undefined or redefined, or parentheses do not surround the putc call.

The putchar function is identical to putc with stream=stdout.

The fputc function obtains the next character, if it exists, as an unsigned char. The input stream pointed to by stream converts the unsigned char to an integer (which has no sign extension), and sets the file position indicator for the stream, when it is defined.

The character is appended to the output stream if either the file cannot support positioning requests (as in terminal files) or if the stream was opened with one of the append modes.

fputc invocation is always a call to a function, never a macro call replacement.

The putw function writes the first argument w to the stream pointed to by stream arguments. It does this as if it was seen as the concatenation of four characters. The write operation does not support special alignment in the file.

Diagnostics

The following results are when this function is successful:

- The functions putc,putchar and fputc return the character written.
- The function putw returns the w value.

The following results are when this function is not successful:

If a write error occurs, the error indicator for the stream is set and the EOF value is returned.

12.3.11 fread, fwrite

Synopsis

```
#include <stdio.h>
size_t fread (ptr,size,nitems,stream)
char *ptr;
size_t size;
size_t nitems;
FILE *stream;

size_t fwrite (ptr,size,nitems,stream)
char *ptr;
size_t size;
size_t nitems;
FILE *stream;
```

Description

The fread function reads nitems elements from the stream pointed to by stream. It reads these elements into an array pointed to by ptr, whose size is specified by size.

The fwrite writes up to nitems elements from the array pointed to by ptr to the stream pointed to by stream. The size of the elements is specified by size.

For both functions:

- The file position indicator for the stream (if defined) is advanced by the number of characters successfully read or written.
- The file position indicator for the stream is indeterminate, if a read or write error occurs.

Diagnostics

Both functions return the number of elements successfully read or written. This can be less than nitems if a read or write error occurs, or, for fread only, when it encounters the end of file.

Furthermore, the fread function returns zero if size or nitems is zero. The contents of the array and the state of the stream remain unchanged.

12.3.12 fseek

Synopsis

```
#include <STDIO_H>

FILE * stream;

long offset;

int fseek(FILE *, long int, int);
fseek (stream,offset,ptrname)
```

Description

The fseek function sets the file position indicator for the stream pointed to by stream.

For a binary stream, the new position is the value of the offset added to the position that whence specifies. This is measured in characters from the beginning of the file.

The specified position can have one of three values. It is the beginning of the file if whence is SEEK_SET. It is the current value of the file position indicator if whence is SEEK_CUR. It is the end of file if SEEK_END. The SEEK_xxx are macros defined in stdio_h.

For a text stream, the offset is either zero, or a value returned by an earlier call to the ftell function on the same stream. Whence is SEEK_SET. If not, the operation fails.

A successful call to the fseek function clears the end of file indicator for the stream and undoes any effects of the ungetc function on that stream.

Diagnostics

The fseek function returns nonzero only when the operation fails. For more information, see the restrictions on direct access in the general considerations on files.

NOTE:

fseek does not work on a stream connected to a terminal, SYSOUT, SYSIN, or tape file.

12.3.13 ftell

Synopsis

```
#include <stdio.h>

    long int ftell (stream)
    FILE *stream;
```

Description

The ftell function obtains the current value of the file position indicator for the stream pointed to by stream.

For a binary stream, the value is the number of characters from the beginning of file.

For a text stream, it is an unspecified value that only the fseek function uses. It sets the file position indicator to its position at the time of the ftell call. The difference between two such return values is not the measure of the number of characters written or read.

Diagnostics

The current value of the file position indicator is returned, if the function is successful.

If it is not successful, the ftell function returns -1L and stores the value ENOSR or ENOFAULTPTR in errno. This is if the current position cannot be delivered or if the stream pointer is invalid.

An error can cause the file associated with the stream not to support a position request. Or an error can cause the file to be at a position that cannot be represented as a long integer.

12.3.14 rewind

Synopsis

```
#include <stdio.h>

    void rewind ( stream );
    FILE * stream;
```

Description

The rewind function sets the file position indicator for the stream pointed to by stream at the beginning of the file. It also clears the error indicator for the stream. This function is equivalent to the following:

```
(void) fseek(stream, 0L, SEEK_SET)
```

```
clearerr ( stream ).
```

Diagnostics

No value is returned.

12.3.15 fgetpos

Synopsis

```
#include <stdio.h>
    int fgetpos ( stream, pos)
    FILE *stream;
    fpos_t *pos;
```

Description

The fgetpos function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos.

The value stored contains information that only the fsetpos function uses. It uses it to reposition the stream to its position at the time of the call to the fgetpos function.

Diagnostics

If it is successful, the fgetpos function returns zero. Otherwise, it returns a nonzero value. Either ENOSR or ENOFAULTPTR is stored in errno, according to the current position, or the pos pointer is invalid.

12.3.16 fsetpos

Synopsis

```
#include <stdio.h>
    int fsetpos ( stream, pos)
    FILE *stream;
    fpos_t *pos;
```

Description

The fsetpos function sets the file position indicator according to the value of the object pointed to by pos. An earlier call to the fgetpos function on that stream gives this value. When this function is successful, the end of file indicator of stream is cleared and any effects of ungetc function are undone.

Diagnostics

When this function is successful, the fsetpos function returns zero. Otherwise, it returns a nonzero value. Either ENOFSETPOS or ENOFAULTPTR is stored in errno according the reposition or the pos pointer is invalid.

12.3.17 setprompt

Synopsis

```
#include <stdio.h>
    int setprompt ( stream, prompt, size)
    FILE *stream;
    char *prompt;
    int size:
```

Description

The setprompt function modifies the standard prompt for a file assigned to a terminal. Calling setprompt with size = 0 causes suppression of the prompt.

Diagnostics

If successful, the setprompt function returns zero. Otherwise, it returns -1 (this happens in particular if the file pointed by stream is not assigned to a terminal).

12.4 getc and putc Macros

The getc and putc functions help to improve the efficiency of a program that includes the stdio.h header file. The getc and putc functions are implemented as macros in the header file However, they are also used as functions.

The only restriction for these two macros is that side effects are not allowed on their arguments.

To use getc and putc as functions, not macros, or to have your own implementation, do one of the following:

- specify #undef with either getc or putc
- use fgetc/fputc (which are always functions)

12.5 Non Standard File Processing (Low-level Primitives)

The primitives described in the following pages constitute a set of functions that must assure a minimum modifying capability for programs from the UNIX system run under GCOS 7. These primitives do not perform identical functions in all respects to those of UNIX. Some general features are not taken into account when implemented under GCOS 7, such as non-buffering of input/output, modification or status references of a file considered as a system object (creat).

12.5.1 open

Synopsis

```
#include <stdio_h>

open (name,mode)

char * name;
```

Description

The open function opens the name file in read when in mode 0 or in write if in mode 1 or in both if in mode 2. Name is the address of the EBCDIC character string representing the name of the file to be opened. It ends with the character ' 0'. The file is positioned at the beginning (byte 0). The file descriptor number returned is the one that must be used for all input/output operations on the file.

As long as no stream is associated with a file opened by the open function, the I/O operations behave as though a pseudo binary stream with default setting for data format and mode were associated with them. This means the file can be either text or binary.

Diagnostics

The value -1 is returned if the name of the file to be opened is not valid or if the file does not exist and is opened in read mode, or more generally, if the file opening process fails.

12.5.2 creat

Synopsis

```
#include <stdio_h>

    creat (name, mode)

    char *name;
```

Description

This function allocates space for a non-existing file before opening it. This function also opens the file if it already exists. When this function is successful, the file is open for writing and the old file contents are discarded. The mode argument has no meaning under C/GCOS 7.

Diagnostics

The value -1 is returned if the attempt to open or allocate the file fails.

12.5.3 close

Synopsis

```
#include <stdio_h>

close (fildes)
```

Description

With a file descriptor number like that returned by open or creat, calling the close function closes the associated file. All files are automatically closed on exit although it is recommended that you close the files yourself. All files are closed at the end of a process.

Diagnostics

The value -1 is returned if an unknown fildes file descriptor number is supplied or if an abnormality occurs when the file is effectively closed by the system.

12.5.4 read

Synopsis

```
#include <stdio_h>

read (fildes,buffer,nbytes)

char*buffer;
```

Description

Buffer is the address in which the contiguous nbytes are placed after reading.
The number of characters read is returned.
The value zero is returned when the end-of-file is reached.

Diagnostics

If the reading fails, the returned value is -1.

An error may be caused by any of the following:

- A physical input/output error
- An erroneous buffer address
- A file descriptor number not corresponding with an input file.

12.5.5 write

Synopsis

```
#include <stdio_h>

write (fildes,buffer,nbytes)

char*buffer;
```

Description

Buffer is the address of contiguous nbytes written on the output file. The number of characters actually written is returned. If the number is not identical to nbytes, this is considered as an error.

Diagnostics

The value -1 is returned in case of error: erroneous descriptor number, incorrect buffer address or physical write error.

12.5.6 lseek

Synopsis

```
#include <stdio_h>

long lseek (fildes, offset, whence)

long offset;
```

Description

The file descriptor number (fildes) is for an open file for reading, writing , or both. The offset value is the number of bytes. The current position in the file is modified according to the following whence values:

0	The new position is set to offset bytes with respect to the beginning of the file.
1	The new position is set to the current position plus offset.
2	The new position is set to the end of file location plus offset (if offset is greater than zero, the operation fails).

The new calculated position includes the default setting of SSF data format and LINE_RECORD mode for the current file. For more information, see the section describing general I/O considerations. The lseek function provides the following:

- The new position is never set at a location of specific SSF information data.
- If the default sets the LINE_RECORD mode, the new calculated position does not take into account the newline character corresponding to the end of records.

The following is an example of the contents of a variable-size record:

```
Record 1 ---->   This is a test
Record 2 ---->   of lseek function.
```

The LINE_RECORD mode interprets these as:

```
This is a test\n of lseek function\n
```

The lseek function (fildes,14,0) call places the new position to the first character of the second record and not on the newline character of the first record. The call attempts to access the 15th character from the beginning of file whose descriptor is fildes. The first character is a space character, and the newline character is not physically written.

Diagnostics

The value -1 is returned for an undefined file descriptor or for a search located before the beginning of a file (whence = 0 and offset < 0) or for any access beyond the file.

12.6 Buffering

The setvbuf and setbuf functions are used for buffering files.

NOTE:
When files are fully buffered, do not use a `return` statement to exit the `main` procedure unless you have flushed the buffers and closed the files.

12.6.1 setvbuf

Synopsis

```
#include <stdio_h>

int setvbuf(FILE *, char *, int , size_t );
int setvbuf (stream, buf, mode, size)

   FILE *stream;

   char *buf;

   int mode;

   int size;
```

Description

- The setvbuf function is used only in the following two ways:
  - After the stream to which stream points is associated with an open file.
  - Before any I/O operations on this file. These operations include specific GCOS 7 macros, for example set_ssf_fmt.
- If the buf argument is not a null pointer, the array to which it points is the buffer associated with the stream. The mode argument specifies the mode for the associated buffer. For more information, see the section describing general I/O considerations for files handling.
- The mode argument accepts only the value for _IOLBF or for _IOFBF that is defined in stdio_h header file. The _IOLBF value is line buffered and the _IOFBF value is fully buffered.
- The size of buf is given by the size argument.
- If buf has automatic storage duration, the length of its lifetime must be equal to or greater than that of the opened stream.

- The contents of this array at any time are indeterminate.
- If buf is a null pointer, a buffer with the specified size and mode is dynamically allocated to the stream.

Diagnostic

The setvbuf function returns zero on success. It returns a non-zero value if an invalid mode is given or if the request cannot be honored as described below:

- A dynamic buffer cannot be allocated.
- A line-buffered value (_IOLBF) is specified in a mode argument for a file that has fixed size records.
- The size value is too small.

12.6.2 setbuf

Synopsis

```
#include <stdio_h>

void setbuf(FILE *, char *);
void setbuf ( stream, buf)

FILE *stream;

   char *buf;
```

Description

- The setbuf function is equivalent to the setvbuf function specifying the value _IOFBF (fully buffered mode) for mode and BUFSIZ for size.
- If the buf argument is a null pointer, the request is not honored because non-buffered streams are not supported.
- If BUFSIZ does not respect the specific rules of the size argument of setvbuf, the request is not honored.

Diagnostics

The setbuf function does not return a value.

12.7 Global File Operations

12.7.1 remove

Synopsis

```
#include <stdio.h>
    int remove (char * filename)
```

Description

This function performs the following operations:

- It removes a file. If the file is cataloged, then the catalog is updated after the removal.
- It removes both file and file-linked files. If the specified file has associated files links, as associated with the file link notion in a GCOS 7 environment, then both file and file-linked files are removed.
- It deletes subfiles. If the specified filename refers to a subfile of a library, then the subfile is deleted.
- It performs no action if the file is open and -1 is returned.

Diagnostics

This function returns 0 if removal is normally completed or -1 if an error occurs.

12.7.2 rename

Synopsis

```
#include <stdio.h>
    int rename (char * oldfilename, char * newfilename)
```

Description

This function renames a file. If the file is cataloged, the relevant catalog is updated. This function does nothing if a file named newfilename exists prior to the call to rename function, or if the file oldfilename is open.
In case of library subfiles, the library name specified in both names should be equal.

Diagnostics

This function returns 0 if renaming is normally completed or -1 if an error occurs.

12.7.3 tmpfile

Synopsis

```
#include <stdio.h>
    FILE * tmpfile (void)
```

Description

This function creates a file that is removed automatically when it is closed or at program termination, whether or not the program terminates normally.
This function returns a pointer to the stream of the file or a null pointer if the file cannot be created.

Diagnostics

The following are some of the error messages sent if the file cannot be created:

- "INTERNAL FILE NAME GENERATION ERROR"
- "ERROR IN OPENING TEMPORARY FILE"
- "ERROR IN TEMPORARY FILE DESCRIPTOR"

12.7.4 tmpnam

Synopsis

```
#include <stdio.h>
    char * tmpnam (char * filename)
```

Description

This function generates a string that is a new file name. A different file name is generated each time tmpnam is called. The generated file name is unique.

The `tmpnam` function generates a file name as input to the fopen function, and this file must be explicitly removed after use.

If the argument is a null pointer, the tmpnam function leaves its result in the variable tmpnam_ptr defined in the STDIO standard header. In this case, the function returns a pointer to that object, and a subsequent call to the tmpnam function when the argument is a null pointer modifies the same object.

If the argument is not a null pointer, it is assumed to point to an array of at least L_tmpnam chars; the tmpnam function writes its result in that array and returns the argument as its value.

Diagnostics

The tmpnam function does not modify the variable errno. The error message is send if the function is called more than TMP_MAX times.

13. Formatting I/O

13.1 fprintf

Synopsis

```
#include <stdio_h>

int fprintf(FILE *, const char *, ...);
int fprintf(FILE * stream, char * format, ...);
```

Description

The fprintf function writes output to the stream pointed to by stream, under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

A format string contains two types of object: plain characters, which are copied unchanged to the output stream, and conversion specifications, each of which results in fetching zero or more subsequent arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The fprintf function returns when the end of the format string is encountered.

Each conversion specification is introduced by the character %.

After the %, the following appear in sequence.

1. Zero or more flags that modify the meaning of the conversion specification.
2. An optional decimal integer specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

3. An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and % conversions, the number of digits to appear after the decimal point for e, E and f conversions, the maximum number of significant digits for the G conversions, or the maximum number of characters to be printed from a string in a conversion. The precision takes the form of a period (.) followed by an optional decimal integer, if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.
4. An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value must be cast to short or unsigned short before printing); an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; or an optional L specifying that a following e, E, f, g, or G conversion specifier applies to a long double argument. If an h, l, or L appears with any other conversion specifier, it is ignored.
5. A character that specifies the type of conversion to be applied.
6. A field width, field precision, or both may be indicated by an asterisk * instead of a digit string. In this case, an int argument supplies the field width or precision. The arguments specifying field width or precision must appear before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if it were missing.

The flag characters and their meanings are as follows:

- The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a plus or minus sign.

space If the first character of a signed conversion is not a sign, a space will be provided for the result. If both the space and + flags appear, the space flag is ignored.

The result is to be converted to an "alternate form". For c, d, i, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a non-zero result will have 0x (or 0X) added to the end of it. For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeros will not be removed from the result, as they normally are.

The conversion specifiers and their meanings are:

d,i,o,u,x,X — The int argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f — The double argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal point appears. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e, E — The double argument is converted in the style [-]d.dde+dd, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The value is rounded to the appropriate number of digits. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be printed is greater than or equal to 1E+100, additional exponent digits will be printed as necessary.

g, G — The double argument is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style e will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision (the default value is 6). Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

c — The least significant byte of the int argument is converted to a character and printed.

s — The argument is taken to be a (char *) pointer to a string. Characters from the string are written up to, but not including, the terminating NULL, or until the number of characters indicated by the precision are written. If the precision is missing it is taken to be arbitrarily large, so all characters before the first NULL are printed.

p — Although the expected argument is one that points to void (void*), any typed pointer is converted.

With one exception, if an argument is a union, aggregate, or points to an object of aggregate type, the behavior is not defined. The exception is for an array of character type when using %s conversion or a pointer when using %p conversion.

The conversion specifier 'p' provides the following output for a GCOS 7 pointer:

```
<TAG>"/"<RING>"/"<STN>"."<STE>"."<SRA>
```

where:

```
<TAG> :: = "DT" |"FT" |"IT" |"RS"
<RING>:: = "R0" |"R1" |"R3"
```

and <STN>, <STE>, and <SRA> are numbers in hexadecimal notation.

n	The argument is taken to be an (int *) pointer to an integer into which is written the number of characters written to the output stream so far by this call to fprintf. No argument is converted.
%	A % is printed. No argument is converted.

If the conversion specifier is a lower-case letter that is not described above, the behavior is undefined. If the conversion specifier is any other character that is not described above, the behavior is implementation-defined.

If any argument is or points to an aggregate (except for an array of characters using %s conversion or any pointer using %p conversion), the behavior is undefined.

In no case does a nonexistent of small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Diagnostics

The fprintf function returns the number of characters transmitted, or a negative value if an output error was encountered.

The following example shows how to print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to strings:

```
#include <stdio_h>

fprintf (stdout, "%s, %s %d, %.2d:%.2d\n",
    weekday, month, day, hjour, min);
```

13.2 printf

Synopsis

```
#include <stdio_h>

int printf (char *format, ...);
int printf(const char *, ...);
```

Description

The printf function is equivalent to fprintf with the argument stdout interposed before the arguments to printf.

Returns

The printf function returns the number of characters transmitted, or a negative value if an output error was encountered.

13.3 sprintf

Synopsis

```
#include <stdio_h>

int sprintf(char *, const char *, ...);
int sprintf (char *s, char *format, ...);
```

Description

The sprintf function is equivalent to fprintf, except that the argument s specifies an array into which the generated output is to be written, rather than to a stream. A NULL character is written at the end of the characters written; it is not counted as part of the returned sum.

Diagnostics

The sprintf function returns the number of characters written in the array, not counting the terminating NULL character.

13.4 fscanf

Synopsis

```
#include <stdio_h>

int fscanf(FILE *, const char *, ...);
int fscanf(FILE *stream, char *format, ...);
```

Description

The fscanf function reads input from the stream pointed to by stream, under control of a format string that specifies how input text is converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient argument pointers for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

The format can contain any number of spaces, horizontal tabs, or new-line characters, which cause input to be read up to the next non-whitespace character.

The format can contain an ordinary character (not %), which must match the next character of the input stream.

The format can contain conversion specifications, consisting in sequence of the character %, an optional assignment suppressing character *, an optional decimal integer that specifies the maximum field width, an optional h, l, or L indicating the size of the receiving object, and a conversion specifier.

A conversion specification requires the following steps:

1. The process ignores (skips) input white-space characters, unless the specification includes a], c or n specifier. The isspace function defines white-space characters.
2. The process reads an input item from the stream, unless the specification includes an *n* specifier. It does not read the first character after the input item. If the length of the input item is zero, the execution of the directive fails.
3. The process converts the input item to a type appropriate to the conversion specifier. If it is not a matching sequence, the execution of the directive fails. The process places the conversion result in an object pointed to by the first available argument after the format argument. If this object does not have the appropriate type, or if the result of the conversion cannot be represented the provided space, the behavior is not defined.

The following conversion specifiers are valid:

d	A decimal integer is expected; the subsequent argument must be a pointer to integer. The expected matching sequence is the same as that of the strtol function with the value 10 for the base argument.
i	An integer is expected; the subsequent argument must be a pointer to integer. The input is interpreted as an integer constant, with an optional sign prefix and an optional integer suffix. If the input field begins with the characters 0x or 0X, the field is taken as a hexadecimal integer. Otherwise, if the input field begins with the character 0, the field is taken as an octal integer. Otherwise, the input field is taken as a decimal integer. The expected matching sequence is the same as that of the strtol function with the value 0 for the base argument.
o	An octal integer is expected; the subsequent argument must be a pointer to integer. The expected matching sequence is the same as that of the strtoul function with the value 8 for the base argument.
u	An unsigned decimal integer is expected; the subsequent argument must be a pointer to integer. The expected matching sequence is the same as that of the strtoul function with the value 10 for the base argument.
x	A hexadecimal integer is expected; the subsequent argument must be a pointer to integer. The expected matching sequence is the same as that of the strtoul function with the value 16 for the base argument.
s	A character string is expected; the subsequent argument must be a pointer to char which points to an array large enough to accept the string and a terminating NULL, which will be added automatically. The input filed is terminated by a space, a horizontal tab, or a new-line, which is not part of the field.
c	A character is expected; the subsequent argument must be a pointer to char. The normal skip over white-space characters is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument must refer to a character array, and the indicated number of characters is read.
e, f, g	A floating point number is expected; the subsequent argument must be a pointer to floating. The input format for floating point numbers is an optionally signed sequence of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer. The expected matching sequence is the same as that of the strtod function.

n — No input is consumed; the subsequent argument must be a pointer to integer into which is written the number of characters read from the input stream so far by this call to fscanf. This is not counted as a match input item.

[— A string that is not to be delimited by spaces is expected; the normal skip over white-space characters is suppressed in this case. The subsequent argument must be a pointer to char just as for %s. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex (^), the input field consist of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. A NULL character will be appended to the input.

If the conversion specifier begins with [] or [^], the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification. There is no restriction for a - character in the scanlist.

% — A single % is expected in the input at this point; no assignment occurs.

p — Use this format to match an input sequence that is the same as the output sequence produced by the printf function with the %p conversion specifier.

If the conversion specifier is a lower-case letter that is not described above, the behavior is undefined. If the conversion specifier is any other character that is not described above, the behavior is implementation-defined.

The conversion specifiers d, i, o, u, and x may be preceded by l to indicate that the subsequent argument is a pointer to long int rather than a pointer to int, or h to indicate that it is a pointer to short int instead. Either the h or the I precedes the conversion specifiers o, u, and x. The h precedes it if the corresponding argument points to unsigned short int, and the I precedes it if it is a pointer to unsigned long int. Similarly, the conversion specifiers e and f may be preceded by l to indicate that the subsequent argument is a pointer to double rather than a pointer to float, or by L to indicate a pointer to long double.

The conversion specifiers x, e and g may be capitalized. However, the use of upper-case has non effect on the conversion process and both upper-case and lower-case input is acceptable.

If end-of-file is encountered during a conversion, the conversion terminates. If conversion terminates on a conflicting input character, the offending character is left unread in the input stream. Trailing white space (including a new-line) is left unread unless matched in the control string. The success of literal matches and suppressed assignments is not directly determinable other than via the %n conversion.

Diagnostics

The fscanf function returns the number of assigned input items, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format, or EOF if end-of-file is encountered before the first conflict or conversion. Otherwise, fscanf returns when the end of the format string is encountered.

13.5 scanf

Synopsis

```
#include <stdio_h>

int scanf(const char *, ...);
int scanf (char *format, ...);
```

Description

The scanf function is equivalent to fscanf with the argument stdin interposed before the arguments to scanf.

Diagnostics

The scanf function returns the number of assigned input items, which can be zero in the event of an early conflict between an input character and the format, or EOF if end-of-file is encountered before the first conflict or conversion. Otherwise, scanf returns when the end of the format string is encountered.

13.6 sscanf

Synopsis

```
#include <stdio_h>

int sscanf(const char *, const char *, ...);
int sscanf (char *s, char *format, ...);
```

Description

The sscanf function is equivalent to fscanf, except that the argument s specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the fscanf function.

Diagnostics

The sscanf function returns the number of assigned input items, which can be zero in the event of an early conflict between an input character and the format. Otherwise, sscanf returns when the end of the format string is encountered.

13.7 vfprintf, vprintf, AND vscanf

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
 int vfprintf (FILE *stream,const char *format,va_list arg);
 int vprintf (const char *format,va_list arg);
 int vsprintf (char *s,const char *format,va_list arg);
```

Description

The vfprintf function is equivalent to the fprintf with the variable argument list replaced by arg parameter. The va_start macro, and some subsequent va_arg calls initialize the arg parameter. The value of arg after vfprintf call is indeterminate.

The vfprintf function is comprised of vprintf and vsprintf, and the fprintf function is comprised of printf and sprintf.

The vfprintf function returns the number of written characters or a negative value if an output error occurs. The vsprintf function returns the number of characters written in the array s, not including the terminating null character. If copied objects overlap, the results are not defined.

EXAMPLE:

```
#include <stdarg.h>
#include <stdio.h>
  main() {  f("%s %s","string1","string2"); }
  f( char *format,...)
  { va_list p;
    va_start (p,format);
    vprintf (format,p); /* print 'string1 string2' on stdout */
  }
```

❑

Diagnostics

The diagnostics are the same as those of FPRINTF, PRINTF, and SCANF respectively.

14. The Use of STDLIB_H

14.1 The <STDLIB_H> Header Subfile

This is the standard header subfile to be "included" for the conversion function (etof, etoi, etc.), the memory allocation functions (malloc, etc.), the environment functions (abort and system) and the random number generator functions (rand and srand).

14.2 Memory Allocation

The memory allocation functions are malloc, calloc, realloc, and free.

Synopsis

```
#include <stdlib_h>

void *malloc(size_t);
char *malloc (size)

unsigned size;

#include <stdlib_h>

void *calloc(size_t, size_t);
char *calloc (nelem, elsize)

unsigned nelem, elsize;

#include <stdlib_h>

void *realloc(void *, size_t);
char *realloc (ptr, size)

char *ptr;

unsigned size;

#include <stdlib_h>

free (ptr)
char *ptr;
```

Description

Malloc and free are the two primitives used for dynamically allocating or deallocating memory space. Malloc returns a pointer to a block whose size is at least equal to the value specified in the size parameter.

The argument for free is a pointer to a block allocated by malloc. The freed space can be used again but its contents remain unchanged.

Malloc searches for the free space that is most suitable to the required size in bytes. Allocation is optimized to avoid too much garbage. Contiguous free spaces are concatenated when a block is freed.

Realloc	Changes the size of a block pointed to by ptr to a block of size bytes and returns a pointer to the block with the modified size. The block can be moved in the list of allocated blocks (ptr is changed) but its contents remain unchanged.
Calloc	Allocates space for an array of nelem elements whose size is equal to elsize. The space is initialized to zero.

Diagnostics

- Malloc, realloc, calloc return a null pointer if no more space can be allocated or if an error occurs when searching for a block to allocate. (You have probably tried to write outside the bounds of the allocated block.)
- The pointer of a block to be freed by free is tested. If the value is incoherent, -1 is returned.
- Allocating blocks of a size greater than 65511 bytes require the use of a LINKER option (see the section on linking)
- Allocating objects of null size is valid and a non-null pointer is therefore returned. This feature is not portable.
- The difference between GCOS 7 and UNIX lies in their memory management algorithms.

14.3 Conversions

The conversion functions are as follows:

ecvt, fcvt, gcvt
etof, etoi, etol

14.3.1 ecvt, fcvt, gcvt

Synopsis

```
#include <stdlib_h>
    char *ecvt (value, ndigit, decpt, sign)
    double value;
    int ndigit, *decpt, *sign;
#include <stdlib_h>
    char *fcvt (value, ndigit, decpt, sign)
    double value;
    int ndigit, *decpt, *sign;
#include <stdlib_h>
    char *gcvt (value, ndigit, buf)
    double value;
    char *buf;
```

Description

ecvt	Converts value into an EBCDIC character string terminating with ' \0' and returns a pointer to this string. The position of the decimal point with respect to the beginning of the string is returned via decpt. A negative value means that the point is located to the left. If the resulting sign is negative, the integer pointed to by sign is other than zero. Otherwise, it is equal to zero. The last digit is rounded off.
fcvt	Is identical to ecvt except that the rounded off digit is computed according to ndigits and that the output corresponds to the FORTRAN format F.
gcvt	Converts value to an EBCDIC character string terminating with ' \0' in buf and returns a pointer to buf. It attempts to supply significant ndigits in FORTRAN format F and otherwise outputs in format E. End zeroes can be deleted.

14.3.2 etof, etoi, etol

Synopsis

```
#include <stdlib_h>
   double etof (nprt)
   char *nptr;
#include <stdlib_h>
   etoi (nptr)
   char *nptr;
#include <stdlib_h>
   long etol (nptr)
   char *nptr;
```

Description

These functions convert a string pointed to by nptr to a floating point number, an integer and a long integer, respectively. The first unrecognizable character ends the string.

Etof recognizes an optional string of tabulations and spaces, then an optional sign, then a string of digits possibly containing a decimal point, then an optional 'e' or 'E' followed by an optionally-signed integer.

Etoi and etol recognize an optional string of tabulations and spaces, then an optional sign, then a string of digits.

14.3.3 strtod, strtol, strtoul

Synopsis

```
#include <stdlib_h>
    double strtod (const char *nptr, char **endptr);
#include <stdlib_h>
    long int strtol (const char *nptr, char **endptr, int base);
#include <stdlib_h>
    unsigned long int strtoul (const char *nptr, char **endptr, int base);
```

Description

The strtod function converts the initial portion of the string to which nptr points into double representation. The form of the input string is a real constant. It is sometimes preceded by space characters that the isspace function specifies, and it is followed by unrecognized characters. The function returns the converted value. If no conversion is possible, zero is returned.

The strtol function performs the conversion to long int representation. If base is 0, the form of the input string is an integer constant. This can be preceded by spaces, and followed by unrecognized characters. If base is not 0, its value must be between 2 and 36, and the letters from A (or a) to Z (or z) are ascribed the values 10 to 35.

The strtoul function is the same as strtol, but with an unsigned long int result.

For these three functions, a pointer to the first unrecognized character is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

14.4 The Environment Functions

The environment functions are as follows:

exit
sexit
abort

14.4.1 exit, sexit

Synopsis

```
#include <stdlib_h>
void exit(int);
    exit (status)
    sexit (status);
void abort(void);
    abort ();
    system (s)
    char *s;
```

Description

Exit terminates program execution with a status return code. Implicitly, all programs terminating normally use exit (0). More generally, status supplies the status of the end of a STEP. The status is linked to severity:

Status	Severity
0.99	0
100.999	1
1000.9999	2
10000.19999	3
20000.32767	4
other	5

All files are closed before output after deferred input/output operations are terminated.

The command sexit terminates a program in the same way as exit except for the closing of files.

The use of this function (sexit) is not recommended because GCOS closes all files, whether or not their status is unstable.

14.4.2 atexit

Synopsis

```
#include <stdlib_h>
    int atexit (void (*func)(void));
```

Description

The atexit function registers the function to which func points, to be called without arguments at normal program termination. This is either the execution of the last executable statement in main if not a return, or the execution of exit(0). Successive calls to the atexit function register an ordered list of functions to be called in the order of registration upon termination. Up to 33 functions can be registered in this way. The returned value is zero if the registration succeeds, and 1 if it fails.

14.4.3 abort

Abort terminates a step in the process called 'abort task'.

14.4.4 getenv, system

Synopsis

```
#include <stdlib_h>
    char *getenv (const char *name);
#include <stdlib_h>
    int system (const char *string);
```

Description

The getenv function returns a pointer to the given GCL variable name. This GCL variable is either a user global variable or a system variable. If the variable contains a list, a space separator links the list elements. The returned pointer is null if the name is unavailable. This function works both in IOF and batch modes.

EXAMPLE:

```
   #include <stdio.h>
   #include <stdlib.h>
   char *p;
   main () {
   p = (char *)malloc (256);
    printf (" %s\n", getenv ("#YES"));
    printf (" %s\n", getenv ("MYGLOB"));
    printf (" %s\n", getenv ("Unknown"));
   }
```

```
S: GLOBAL MYGLOB TYPE=CHAR;
S: LET MYGLOB 'contents of MYGLOB';
S: EXEC_PG TGETENV;
    (YES Y 1)
    contents of MYGLOB
  *CLR251 : NO SUCH GCL VARIABLE. .RC=98081870->GCL    8,NAMEERR
    (null)
```

❑

The system function executes a GCL directive given in a string. The string contains a valid GCL directive that is a GCL command in the H_NOCTX domain. If the command is syntactically correct, the returned value is 0. Otherwise it returns -1. The execution of the directive returns no result, apart from this status, to the calling C program.

If the pointer is NULL, the function returns 1, as a command processor is available. This function works in IOF and batch modes. Example:

```
system ("DS EX");
```

For more information about the H_NOCTX domain, refer to the *IOF Terminal User's Reference Manual*.

14.5 Random Number Generator

14.5.1 rand

Synopsis

```
#include <stdlib_h>

int rand(void);
   int rand();
```

Description

Successive calls to rand return integer values in the range 0 to 32767 that are the successive results of a pseudo-random number generator.

Diagnostics

If called with arguments rand returns the code -1.

14.5.2 srand

Synopsis

```
#include <stdlib_h>

void srand(unsigned int);
void srand (seed)

unsigned int seed;
```

Description

The srand function uses the argument as a seed for a sequence of pseudo-random numbers to be returned by subsequent calls to rand. If rand is then called with the same seed value, the sequence of pseudo-random numbers will be repeated.

If rand is called before any calls to srand have been made, the same sequence is generated as when srand is called with a seed value of 1.

Diagnostics

If srand is called with an incorrect number of arguments the seed value remains unchanged.

14.6 bsearch, qsort

Synopsis

```
#include <stdlib_h>
void *bsearch (const void *key, const void *base, size_t nmemb,
               size_t size, int (*compar)(const void *, const void *));

#include <stdlib_h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *));
```

Description

The bsearch function searches an array of nmemb objects for an element that matches the object pointed to by key. The initial element of the array is pointed to by base. The size of each element of the array is specified by size. Compar points to the comparison function that is called with two arguments pointing to the key object and to an array element, in that order. The comparison function returns an integer less than, equal to, or greater than 0 if the key object is considered, respectively, to be less than, to match, or to be greater than the array element. It is assumed that the array is sorted in ascending order. The bsearch function returns a pointer to a matching element of the array, or a null pointer if no match is found.

Compar can be a null pointer, especially for GCOS 7. In this case, the comparison is done according to the EBCDIC collating sequence.

The qsort function sorts an array of nmemb objects, and base points to the initial element. The size of each object is specified by size. The sort is in ascending order.

Compar can be a null pointer, especially for GCOS 7. In this case the sort is done according to the EBCDIC collating sequence.

14.7 abs, div, labs

Synopsis

```
#include <stdlib_h>
    int abs (int j);
#include <stdlib_h>
    div_t div (int numer, int denom);
#include <stdlib_h>
    long int labs (long int j);
```

Description

abs returns the absolute value of its integer operand.

Limitation

The absolute value of i must be less than or equal to 2147483647 (which is 2^{31} -1).

Diagnostic

If the argument is equal to the most negative integer abs returns the value 0 and errno is set to EDOM.

The div function performs the euclidian division of numbering by denomination and returns a structure of type div_t, comprising both the quotient and the remainder. Denom cannot be zero.

The labs and ldiv functions are similar to abs and div, with arguments and returned values of type long int.

15. Character Handling

15.1 The <CTYPE_H> Header Subfile

This header file contains the different macros describing the functions used for character handling. When this file is included, it brings a 256-character static table back to the source.

15.2 EBCDIC Character Subsets

The following macros define the subsets of the EBCDIC set of characters:

isalnum	isxdigit
isalpha	isprint
isdigit	isspace
islower	ispunct
isupper	iscntrl
	isgraph

Synopsis

```
#include <ctype_h>

  isalpha (c)
        .
        .
        .
  iscntrl

int isalpha(int)
        .
        .
        .
int iscntrl(int)
```

Description

These macros define the sub-sets of the EBCDIC set of characters. Each macro is a predicate belonging to the sub-set that it defines. They return a non-zero value if the c character is a member of the sub-set. If not, they return zero.

isalnum (c) — Returns a non-zero value if the c character is a member of the following set of alphanumeric characters. Otherwise, it returns 0.

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T
U V W X Y
a b c d e f g h i j k l m n o p q r s t
u v w x y z
```

isalpha (c): — Returns a non-zero value if c is a character of the following alphabet set. Otherwise, returns 0.

```
A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z
a b c d e f g h i j k l m n o p q r s t
u v w x y z
```

isdigit (c): — Returns a non-zero value if c is one of the following digits. Otherwise, returns 0.

```
0 1 2 3 4 5 6 7 8 9
```

islower (c): — Returns a non-zero value if c is a lower-case letter. Otherwise, returns 0.

isupper (c): — Returns a non-zero value if c is an upper-case letter. Otherwise, returns 0.

isxdigit (c): — Returns a non-zero value if c is a member of the following set of hexadecimal digits. Otherwise, returns 0

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
```

isprint (c):	Returns a non-zero value if c belongs to the following set of printing characters. Otherwise, returns 0.
	
	`SP (space) ! " # $ % & ' ( ) * + , - . /`
	`0 1 2 3 4 5 6 7 8 9/; < = > ?`
	`@ A B . . . . Z [ ] _ a b c . . . z |`

isspace (c):	Returns a non-zero value if c is a character of the following set. Otherwise, returns 0. space formfeed horizontal tab newline carriage return vertical tab.
ispunct (c):	Returns a non-zero value if c is a printing character (see isprint) except for spaces and alphanumeric characters (see isalnum). Otherwise, returns 0.
iscntrl (c):	Returns a non-zero value if c is a control character, which is any non-printing character (see isprint).
isgraph (c):	Returns a non-zero value if c belongs to the set of printing characters shown in isprint, except for the space character. Otherwise, it returns 0.

15.3 Converting to Lower and Upper Case

The following macros convert characters to lower and upper case.

tolower	_tolower
toupper	_toupper

Synopsis

```
#include <ctype_h>

tolower (c)
int tolower(int)

char c;

toupper (c)
int toupper(int)

char c;

_tolower (c)

char c;

_toupper (c)

char c;
```

Description

tolower:	Converts c to lower-case if c is an upper-case letter. Otherwise, c remains unchanged.
toupper:	Converts c to upper-case if c is a lower-case letter. Otherwise, c remains unchanged.
_tolower:	Is identical to tolower except that result is undefined if c is not an upper-case letter.
_toupper:	Is identical to toupper except that result is undefined if c is not a lower-case letter.

NOTE:

_tolower and _toupper are faster than tolower and toupper; therefore, it is preferable to use them when you are sure that c is an upper or lower-case letter.

16. The Use of STRING_H

16.1 The <STRING_H> Header Subfile

This is the standard header subfile to be "included" for the functions used for manipulating character strings, for buffer management and memory management.

16.2 String Handling

The string handling functions are:

```
strcpy     cmpstr    strchr    notstr    prefix
strncpy    cpystr    strcat    strspn    scnstr
strcmp     lenstr    strcspn   substr
strncmp    strlen    index     strpbrk
strncat    strcoll   strxfrm   strrchr
strchr     strtok    strerror
```

Synopsis

```
#include <stdio_h>

char *strcpy(char *, const char *);
        char *strcpy (s1,s2)
              char *s1, *s2;

char *strncpy(char *, const char *, size_t);
        char *strncpy (s1,s2,n)
              char *s1, *s2;
              int n;

int strcmp(const char *, const char *);
        strcmp (s1,s2)
              char *s1, *s2;
```

```
int strncmp(const char *, const char *, size_t);
        strncmp (s1,s2,n)
              char *s1, *s2;
              int n;

        cmpstr (s1,s2)
              char *s1, *s2;

        cpystr (ds, arg1, arg2, ..., null)
              char *ds, *arg1, *arg2, ...;

        lenstr (s)
              char *s;

size_t strlen(const char *);
        strlen (s)
              char *s;

char *strchr(const char *, int);
        char *strchr (s,c)
              char *s, c;

char *strcat(char *, const char *);
        char *strcat (s1,s2)
              char *s1, *s2;

char *strncat(char *, const char *, size_t);
        char *strncat (s1,s2,n)
              char *s1, *s2;
              int n;

        char *index (s,c)
              char *s, c;

        notstr (p,s)
              char *p, *s;

size_t strspn(const char *, const char *);
        strspn (p,s)
              char *p, *s;

size_t strcspn(const char *, const char *);
        strcspn (p,s)
              char *p, *s;

char *strpbrk(const char *, const char *);
        strpbrk (p,s)
              char *p, *s;

        prefix (s1,s2)
              char *s1, *s2;
```

```
        scnstr (s,c)
              char *s, c;

        substr (s,p)
              char *s, *p;

#include <string_h>
        char *strncat (char *s1, const char *s2, size_t n);
#include <string_h>
        int strcoll (const char *s1, const char *s2);
#include <string_h>
        size_t strxfrm (char *s1, const char *s2, size_t n);
#include <string_h>
        char *strrchr (const char *s, int c);
#include <string_h>
        char *strstr (const char *s1, const char *s2);
#include <string_h>
        char *strtok (char *s1, const char *s2);
#include <string_h>
        char *strerror (int errnum);
```

Description

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcpy	Copies string s2 to s1, stopping after the null character has been moved. strncpy copies exactly n characters (where n is the number of characters), truncating or null-padding s2; the target may not be null-terminated if the length of s2 is n or more. Both return s1.
strcmp	Compares its arguments and returns an integer greater than, equal to, or less than 0. This is according to whether s1 is lexicographically greater than, equal to, or less than s2.
strncmp	Makes the same comparison but for, at most, n characters, where n is the number of characters.
cmpstr	Compares two strings, character by character, for equality. The first string starts at s1 and is terminated by a null ' 0'; the second is likewise described by s2. The strings must match through and including their terminating null characters. The value returned is 1 if the strings are equal, otherwise 0.

cpystr	Concatenates a series of strings into the destination string ds. Each string begins at argx and is terminated by a null '\0'. The first character of arg2 is placed just after the last character (before the null) copied from arg1, etc. The series of string arguments is terminated by a null pointer argument. A null is appended to the final destination string to terminate it properly. The value returned is a pointer to the terminating null in the destination string.
strlen (lenstr)	Returns the number of non-null characters in s.
strchr	Locates the first occurrence of a specific character c in the string pointed to by s. The terminating null is considered to be part of the string. strchr returns a pointer to the located character, or a null pointer if the character does not occur in the string.
strcat	Appends a copy of string s2 to the end of string s1.
strncat	Copies, at most, n characters, where n is the number of characters. Both strcat and strncat return a pointer to the null-terminated result. The results are unpredictable if the two string arguments overlap in memory.
index	Returns a pointer to the first occurrence of a specific character c in string s, or the null pointer if c does not occur in the string.
notstr	Scans the null terminated string starting at p for the first occurrence of a character not in the null terminated set starting at s. notstr returns the offset of the first character in p not contained in the set s, or the index of the terminating null if all are in s.
strspn	Is the same function as notstr.
strcspn	Is the same function as notstr, except that it starts at p and scans the null terminated string for the first occurrence of a character in the null terminated set that starts at s.
strpbrk	Is the same function as strcspn, except that it returns a pointer to the first character in p that is contained in the set s. It returns a null pointer if there are no characters in s.

prefix	Compares two strings, character by character, for equality. The first string starts at s1 and is terminated by a null '\0'; the second is likewise described by s2. The strings must match up to but not include the null that terminates the second string. That is, s2 must be a prefix of s1. The value returned is 1 if s2 is a prefix of s1, else 0.
scnstr	Looks for the first occurrence of a specific character c in a null terminated target string s. scnstr also returns the offset of the first character that matches c, or the offset of the terminating null if none matches.
substr	Scans the string starting at s, and looks for the first occurrence of the substring at p. The value returned is the index in s of the leftmost character in the sub-string if substr is successful; otherwise, the index of the terminating null is returned. These functions do not check for the validity of any parameter strings.

The strncat function appends not more than n characters to the end of the string pointed to by s1. The characters are from the array to which s2 points. Null characters and subsequent characters are not appended. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. If overlapping objects are copied, the behavior is undefined. The strncat function returns the value of s1.

The strcoll function compares the strings to which s1 and s2 point. It compares according to the collating sequence that the current LC_COLLATE category defines. The returned integer is greater than, equal to, or less than zero, depending on whether the string to which s1 points is greater than, equal to, or less than the string to which s2 points.

The strxfrm function transforms the string to which s2 points into a string to which s1 points and then returns the length of the transformed string. The transformation is done according to the LC_COLLATE category of the current locale category, which is also described in the chapter on localization. It copies no more than n characters. The function returns the length of the transformed string, which can be greater than n. strxfrm(null,0) returns the size needed to hold the transformed string.

The strrchr function returns a pointer to the last occurrence of c in the string to which s points, or a null pointer if c is not found.

The strstr function returns a pointer to the first occurrence of the string to which s2 points in the string to which s1 points. It returns a null pointer if the string is not found.

The strtok function is called in a sequence. This is in order to break the string to which s1 points into a sequence of tokens delimited by a character contained in the string to which s2 points. The first call searches the s1 string for the first character that is not contained in the s2 string. If this character does not exist, a null pointer is returned, otherwise the function returns a pointer to the first token. The function then searches for a character that is contained in the current separator string. If this character does exist, a null character (\0) overwrites it. Each subsequent call that has a null pointer as the first argument (and possibly a different separator string pointed to by s2) searches for a character that is contained in *s2 and continues as described above.

This example contains the following program fragment:

```
static char str[]= "?a???b,,,#c";
char *t;
   ...
t=strtok (str, "?");
t=strtok(null, ",");
t=strtok(null, "#,");
t=strtok(null, "?");
```

After successive calls to strtok, this leads to the following results:

```
#call          str value              t value
1st            ?a\0??b,,,#c\0           a
2nd            ?a\0??b\0,,#c\0          ??b
3rd            ?a\0??b\0,,#c\0          c
4th            ?a\0??b\0,,#c\0          null
```

The strerror function returns a pointer to an error message string. That string corresponds to the error number errnum that is an errno value.

16.3 Buffer Management

The buffer management functions are:

```
fill          cmpbuf
subbuf        scnbuf
cpybuf        notbuf
```

Synopsis

```
#include <stdio_h>

  fill (s,n,c)
        char *s, c;
        int n;

  subbuf (s,ns,p,np)
        char *s, *p;
        int n;

  cpybuf (s1,s2,n)
        char *s1, *s2;
        int n;

  cmpbuf (s1,s2,n)
        char *s1, *s2;
        int n;

  scnbuf (s,n,c)
        char *s, c;
        int n;

  notbuf (p,n,s)
        char *p, *s;
        int n;
```

NOTE:

The buffer management functions do not check for the validity of parameter strings.

Description

fill	Floods the n-character buffer (where n is the number of characters) starting at s with fill character c. fill returns n.
subbuf	Scans the buffer starting at s of size ns, and looks for the first occurrence of the substring at p of size np. The value returned is the offset in s of the leftmost character in the sub-string if subbuf is successful; otherwise, ns is returned.
cpybuf	Copies the first n characters (where n is the number of characters) starting at location s2 into the buffer beginning at s1. The value returned is n, the number of characters copied.
cmpbuf	Compares two text buffers, character by character, for equality. The first buffer starts at s1, the second at s2. Both are n characters long, where n is the number of characters. s1 and s2 are said to be equal if the n characters in s1 and s2 are identical. The value returned is 1 if the buffers are equal, otherwise 0.
scnbuf	Looks for the first occurrence of a specific character c in an n character buffer starting at s. scnbuf returns the offset of the first character that matches c, or n if none.
notbuf	Scans the n-character buffer starting at p for the first instance of a character not in the null terminated set starting at s. If the null character is to be part of the set, it must be the first character in the set. notbuf returns the offset of the first character in p not contained in the set s, or the value n if all buffer characters are in the set.

16.4 Memory Management

16.4.1 The memcpy Function

Synopsis

```
#include <string_h>

void *memcpy(void *, const void *, size_t);
char *memcpy (s1,s2,n)
      char *s1,*s2;
      int n;
```

Description

The memcpy function copies n characters from the array pointed to by s2 to the array pointed to by s1.

The memcpy function returns the value of s1.

16.4.2 The memset Function

Synopsis

```
#include <string_h>

void *memset(void *, int, size_t);
char *memset (s,c,n)
      char *s,c;
      int n;
```

Description

The memset function copies the value of c (cast to unsigned char) into each of the first n bytes of the array pointed to by s.

The memset function returns the value of s.

16.4.3 The memcmp Function

Synopsis

```
#include <string_h>

int memcmp(const void *, const void *, size_t);
  int memcmp (s1,s2,n)
        char *s1,s2;
        int n;
```

Description

The memcmp function compares the first n bytes of the array pointed to by s2 to the array pointed to by s1. The memcmp function returns an integer greater than, equal to, or less than zero, according as the array pointed to by s1 is lexicographically greater than, equal to, or less than the array pointed to by s2.

16.4.4 The memchr Function

Synopsis

```
#include <string_h>

void *memchr(const void *, int, size_t);
  char *memchr (s,c,n)
        char *s,c;
        int n;
```

Description

The memchr function locates the first occurrence of c in the initial n characters of the array pointed to by s. The memchr function returns a pointer to the located character, or a null pointer if the character does not occur in the array.

16.4.5 The memmove Function

Synopsis

```
#include <string_h>
    void *memmove (void *s1, const void *s2, size_t n);
```

Description

The memmove function copies n characters from the object pointed to by s2 into the object pointed to by s1. Unlike memcopy, memmove supports overlapping between the sending and receiving areas. Its performance is less efficient than memcpy, although it can perform a single copy on GCOS 7.

17. Non-Local Jump

17.1 The <SETJMP_H> Header Subfile

This subfile declares two functions and one object type used for calling one of the functions. Calling the longjmp or setjmp function requires inclusion of the subfile.

17.2 setjmp, longjmp

Synopsis

```
  #include <setjmp_h>

int setjmp(jmp_buf);
 int setjmp (env)
       jmp_buf env;

  #include <setjmp_h>

void longjmp(jmp_buf, int);
 longjmp (env, val)
       jmp_buf env;
       int val;
```

Description

setjmp saves its stack environment in env (whose type, jmp_buf, is defined in the <setjmp_h> header file) for later use by longjmp. It returns the value zero (0).

longjmp restores the environment saved by the last call of setjmp with the corresponding env argument. After longjmp is completed, program execution continues as if the corresponding call of setjmp (the caller of which must not itself have returned in the interim) had just returned the value val. longjmp cannot cause setjmp to return the value 0. If longjmp is invoked with a second argument of 0, setjmp will return 1. All accessible data have values as of the time longjmp was called.

Application Usage

If longjmp is called even though env was never primed by a call to setjmp, or if the last such call was in a function which has since returned, the result is unpredictable and can possibly cause damage. If the call to longjmp is in a different function from the corresponding call to setjmp, local variables may have unpredictable values.

IMPORTANT:
These two functions are not guaranteed in a shared module (TPR) context.

18. Mathematical Package

18.1 The <MATH_H> Header Subfile

This standard subfile must be "included" for all mathematical functions. In particular, it defines two macros EDOM and ERANGE which should be used to test the values returned by these functions. If an error occurs in a domain or co-domain the variable errno (described by a macro in this subfile) is set to the value EDOM or ERANGE respectively. All mathematical functions called with an incorrect number of arguments return the value 0 and set the errno variable to the EDOM value.

NOTE:
no error message is issued in this case.

18.2 abs

Name abs

Synopsis

```
  #include <math_h>

int abs(int);
  int abs (i)

int i;
```

Description

abs returns the absolute value of its integer operand.

Limitation

The absolute value of i must be less than or equal to 2147483647 (which is 2^{31} -1).

Diagnostic

If the argument is equal to the most negative integer abs returns the value 0 and errno is set to EDOM.

18.3 fabs

Name fabs

Synopsis

```
#include <math_h>

double fabs(double);
double fabs (x)

double x;
```

Description

fabs returns the absolute value of x.

Diagnostic

If the argument type is incorrect fabs returns the value 0 and errno is set to EDOM.

18.4 floor

Name floor

Synopsis

```
#include <math_h>

double floor(double);
double floor (x)

double x;
```

Description

floor returns the largest integer not greater than x.

Limitation

The absolute value of x must be less than or equal to 2147483647 (which is 2^{31} -1).

Diagnostic

If the absolute value of x is greater than 2147483647, floor returns x and sets errno to EDOM. If the argument type is incorrect, floor returns the value 0 and errno is set to EDOM.

18.5 ceil

Name ceil

Synopsis

```
#include <math_h>

double ceil(double);
double ceil (x)

double x;
```

Description

ceil returns the smallest integer not less than x.

Limitation

The absolute value of x must be less than or equal to 2147483647 (which is 2^{31} -1).

Diagnostic

If the absolute value of x is greater than 2147483647, ceil returns x and sets errno to EDOM. If the argument type is incorrect, floor returns the value 0 and errno is set to EDOM.

18.6 fmod

Name fmod

Synopsis

```
#include <math_h>

double fmod(double, double);
double fmod (x,y)

double x,y;
```

Description

fmod returns the remainder of x divided by y.

Limitation

y must not be equal to zero.

Diagnostic

If y is equal to zero, fmod returns the value 0 and errno is set to EDOM. If the argument type is incorrect (for either x or y), fmod returns the value 0 and errno is set to EDOM.

18.7 modf

Name modf

Synopsis

```
#include <math_h>

double modf(double, double *);
double modf (value,iptr)

double value, *iptr;
```

Description

modf returns the fractional part of value and stores the integral part indirectly through iptr.

Limitation

The absolute value of value must be less than or equal to 2,147,483,647 (which is one less than 2 to the power of 31).

Diagnostic

If the absolute value of value is greater than 2,147,483,647, modf returns value 0 and sets errno to EDOM. If the argument type is incorrect or the pointer iptr is invalid, modf returns the value 0 and errno is set to EDOM.

18.8 sin

Name sin

Synopsis

```
#include <math_h>

double sin(double);
double sin (x)

double x;
```

Description

sin returns the sine of x (measured in radians).

Limitation

To avoid loss of precision, the absolute value of x must be less than 10^{15}.

Diagnostic

If the argument type is incorrect sin returns the value 0 and errno is set to EDOM.

18.9 asin

Name asin

Synopsis

```
#include <math_h>

double asin(double);
double asin (x)

double x;
```

Description

asin returns the arc sine of x in the following range

```
-Pi/2 to +Pi/2
```

Limitation

The absolute value of x must be less than or equal to 1.

Diagnostic

If the argument is greater than 1, or if the argument type is incorrect, asin returns the value 0 and errno is set to EDOM.

18.10 sinh

Name sinh

Synopsis

```
#include <math_h>

double sinh(double);
double sinh (x)

double x;
```

Description

sinh returns the hyperbolic sine of x.

Diagnostic

When the correct value overflows, sinh returns a huge value with the appropriate sign and errno is set to ERANGE.

If the argument type is incorrect, sinh returns the value 0 and errno is set to EDOM.

18.11 cos

Name cos

Synopsis

```
#include <math_h>

double cos(double);
double cos (x)

double x;
```

Description

cos returns the cosine of x (measured in radians).

Limitation

To avoid loss of precision, the absolute value of x must be less than 10^{15}.

Diagnostic

If the argument type is incorrect cos returns the value 0 and errno is set to EDOM.

18.12 acos

Name acos

Synopsis

```
#include <math_h>

double acos(double);
double acos (x)

double x;
```

Description

acos returns the arc cosine of x in the range 0 to PI.

Limitation

The absolute value of x must be less than or equal to 1.

Diagnostic

If the argument is greater than 1, or if the argument type is incorrect, acos returns the value 0 and errno is set to EDOM.

18.13 cosh

Name cosh

Synopsis

```
#include <math_h>

double cosh(double);
double cosh (x)

double x;
```

Description

cosh returns the hyperbolic cosine of x.

Diagnostic

If the correct value overflows, cosh returns a huge value with the appropriate sign and errno is set to ERANGE.

If the argument type is incorrect, cosh returns the value 0 and errno is set to EDOM.

18.14 tan

Name tan

Synopsis

```
#include <math_h>

double tan(double);
double tan (x)

double x;
```

Description

tan returns the tangent of x (measured in radians).

Limitation

To avoid loss of precision, the absolute value of x must be less than 10^{15}.

Diagnostic

The value of tan at its singular points is a huge number, and errno is set to ERANGE. If the argument type is incorrect, tan returns the value 0 and errno is set to EDOM.

18.15 atan

Name atan

Synopsis

```
#include <math_h>

double atan(double);
double atan (x)

double x;
```

Description

atan returns the arc tangent of x in the range -PI/2 to +PI/2.

Diagnostic

If the argument type is incorrect atan returns the value 0 and errno is et to EDOM.

18.16 atan2

Name atan2

Synopsis

```
#include <math_h>

double atan2(double, double);
double atan2 (y,x)

double x,y;
```

Description

atan2 returns the arc tangent of y/x in the range -PI/2 to +PI/2.

Limitation

x and y must not both be zero.

Diagnostic

If x and y are both equal to zero atan2 returns the value 0 and sets errno to EDOM. If the argument type is incorrect (for either x or y) atan2 returns the value 0 and errno is set to EDOM.

18.17 tanh

Name tanh

Synopsis

```
#include <math_h>

double tanh(double);
double tanh (x)

double x;
```

Description

tanh returns the hyperbolic tangent of x.

Diagnostic

If the argument type is incorrect tanh returns the value 0 and errno is set to EDOM.

18.18 exp

Name exp

Synopsis

```
#include <math_h>

double exp(double);
double exp (x)

double x;
```

Description

exp returns the exponential value of x.

Limitation

The absolute value of x must be greater than -200 and less than 174.67.

Diagnostic

If the correct value overflows, exp returns a huge value; if the correct value underflows, exp returns a null value. In both cases, errno is set to ERANGE.
If the argument type is incorrect, exp returns the value 0 and errno is set to EDOM.

18.19 log

Name log

Synopsis

```
#include <math_h>

double log(double);
double log (x)

double x;
```

Description

log returns the natural logarithm of x.

Diagnostic

When x is zero or negative, log returns a huge negative value and errno is set to EDOM.

If the argument type is incorrect, log returns the value 0 and errno is set to EDOM.

18.20 log2, frexp

Name log2, frexp

Synopsis

```
#include <math_h>
    double log2 (double x);
#include <math_h>
    double frexp (double value, int *exp);
```

Description

The log2 function computes the base-two logarithm of x. It is not part of ANSI The frexp function returns a double x of the interval [0.5 ; 1 [, and it returns an int *exp, such that the value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.

EXAMPLE:

The following example uses the frexp function.

```
#include <math.h>
#include <stdio.h>
int i1,i2;
double x1, x2;
main () {
x1 = frexp (1.0, &i1);
x2 = frexp (-0.12, &i2);
printf("%f=frexp(1.0,%d); %f=frexp(-0.12,%d)\n",x1,i1,x2,i2);
}
```

❑

The following program prints on stdout:

0.500000=frexp(1.0,1); -0.960000=frexp(-0.12,-3)

18.21 log10

Name log10

Synopsis

```
#include <math_h>

double log10(double);
double log10 (x)

double x;
```

Description

log10 returns the base ten logarithm of x.

Diagnostic

When x is zero or negative, log10 returns a huge negative value and errno is set to EDOM.

If the argument type is incorrect log10 returns the value 0 and errno is set to EDOM.

18.22 pow

Name pow

Synopsis

```
#include <math_h>

double pow(double, double);
double pow (x,y)

double x,y;
```

Description

pow returns x to the power of y. (x^y).

Diagnostic

If the correct value overflows, pow returns a huge value; if the correct value underflows pow returns a null value. In both cases, errno is set to ERANGE.

If the argument type is incorrect pow returns the value 0 and errno is set to EDOM.

18.23 ldexp

Name ldexp

Synopsis

```
#include <math_h>

double ldexp(double, int);
double ldexp (x,exp)

int exp;

double x;
```

Description

ldexp returns x.2 exp (2 to the power of exp multiplied by x).

Limitation

The absolute value of exp must be greater than or equal to -260, and less than 251.

Diagnostic

If the correct value overflows, ldexp returns a huge value with the appropriate sign; if the correct value underflows, ldexp returns a null value. In both cases, errno is set to ERANGE.

18.24 sqrt

Name sqrt

Synopsis

```
#include <math_h>

double sqrt(double);
double sqrt (x)

double x;
```

Description

sqrt returns the positive square root of x.

Diagnostic

When x is negative, sqrt returns zero and errno is set to EDOM. If the argument type is incorrect, sqrt returns the value 0 and errno is set to EDOM.

19. Time and Date

19.1 The <TIME_H> Header Subfile

This standard subfile is used for the time and date functions.

19.2 Time Retrieval

Name clock

Synopsis

```
#include <time_h>

clock_t clock(void);
clock_t clock ( )

time_t time (time_t *timer);
```

Description

The clock function returns the elapsed processor time since the beginning of program execution. This value is the best approximation of the implementation. The value returned must be divided by the value of the macro CLOCKS_PER_SEC to obtain the time in seconds.

Diagnostic: The Clock Function

The returned value is set to -1 if the time is not available or if clock is called with arguments.

Diagnostic: The Time Function

The time function returns the elapsed time in seconds, with the beginning date of January 1, 1970 at 00:00. If the timer is a valid pointer, the returned value is also assigned to the value to which it points. The timer parameter is mandatory. If it is not required, enter a null value to be passed to the function.

The returned value is set to -1 if time and date are not available.

IMPORTANT:

The main entry must be activated first if the clock and time functions are used. If not, the clock or time return -1.

19.3 Time Handling

Name time handling

The struct tm type holds the component of a calendar time, which is called the broken-down time. This structure contains the following members:

int tm_sec	Seconds after the minute Range	0-61
int tm_min	Minutes after the hour Range	0-23
int tm_hour	hours since midnight Range	0-59
int tm_mday	day of the month Range	1-31
int tm_mon	months since January Range	0-11
int tm_year	years since 1900	
int tm_wday	days since Sunday Range	0-6
int tm_yday	days since January 1 Range	0-365
int tm_isdst	Daylight Saving Time flag	

Synopsis

```
#include <TIME_H>

double difftime(time_t, time_t);
double difftime (time_t time1, time_t time2);

time_t mktime (struct tm *timeptr);

char *ctime (const time_t *timer);

struct tm *gmtime (const time_t *timer);

struct tm *localtime (const time_t *timer);
```

Description

For all these functions, the timer pointer (tmrptr) points to the calendar.

The difftime function computes the difference between the two calendar times time1 and time2. It returns this difference expressed in seconds as a double.

The mktime function converts the broken-down time into a calendar time value with the same encoding as that of the values returned by the time function. The values of tm_wday and tm_yday are ignored, and the other values are not restricted to the ranges indicated above. Upon successful completion, the value of tm_wday and tm_yday are set appropriately, and the other values are forced to the ranges indicated above to represent the specified calendar time. If the calendar time cannot be represented, the function returns the value -1.

The ctime function converts the calendar time to local time in the form of a string. It is equivalent to asctime, which is localtime(timer).

The gmtime function converts the calendar time into a broken-down time expressed as Coordinated Universal Time. The timer pointer points to the calendar time. It returns a null pointer if UTC is not available.

The localtime function converts the calendar time into a broken-down time, expressed as local time.

Diagnostic

The returned value is set to -1 if the time and date are not available. If the number of arguments is incorrect time returns -1

19.4 Time Edition

Name asctime

Synopsis

```
#include <TIME_H>

char *asctime (const struct tm *timeptr);

size_t strftime(char *, size_t, char *,
                const struct tm *);
size_t strftime (char *s, size_t maxsize,
       const char *format, const struct tm *timeptr);
```

Description

The asctime function converts the broken-down time into a string in the following form:

```
Tue Jan  8 14:24:47 1991\n\0
```

The timeptr points to the asctime function.

The strftime function places characters into the array to which s points. Format points to the string that controls this. The format string contains ordinary characters and conversion specifiers that consist of a % character followed by a character that determines the behavior of the conversion specification. All ordinary characters are copied unchanged into the array. No more than maxsize characters are placed into the array.

The characters in the following list replace each conversion specifier. The LC_TIME of the current locale determines the characters and also the structure to which the timeptr points determines them.

The replacement characters are as follows:

%a	Abbreviated weekday name of the locale	
%A	Full weekday name of the locale	
%b	Abbreviated month name of the locale	
%B	Full month name of the locale	
%c	Appropriate date and time representation of the locale	
%d	Day of the month as a decimal number	01-31
%H	Hour (24-hour clock) as a decimal number	00-23
%I	Hour (12-hour clock) as a decimal number	01-12
%j	Day of the year as a decimal number	001-366
%m	Month as a decimal number	01-12
%M	Minute as a decimal number	00-59
%p	Locale equivalent of the AM/PM designation associated with a 12-hour clock	
%s	Second as a decimal number	00-61
%U	Week number of the year as a decimal number. The first Sunday as the first day of week 1	00-53
%w	Weekday as a decimal number 0 (Sunday)-6	
%W	Week number of the year as a decimal number. The first Monday as the first day of week 1	00-53
%x	Appropriate date representation of the locale	
%X	Appropriate time representation of the locale	
%y	Year without century as a decimal number	00-99
%Y	Year with century as a decimal number	
%Z	No character	
%%		

The strftime function returns the number of characters placed into the array to which s points. However, this does not include the terminating null character if that total number (including the terminating null character) is not more than maxsize. Otherwise, 0 is returned and the contents of the array are indeterminate.

20. STDARG Functions

20.1 The <STDARG_H> Header Subfile

The starg.h file is a header file. It declares a type va_list that holds information for the macros va_start, va_arg, and va_end. The starg.h file uses several arguments to call functions.

The three macros described in this section access in a portable way all the varying arguments that are unnamed in the called function. However, the called function is declared with at least one fixed parameter. To access variable arguments, the called function declares an object of va_list type. In the following description, the object of the va_list is referred to as ap and the right-most fixed parameter of the called function is referred to as parmN.

If one of these macros is suppressed (by #undef), the execution of the called function leads to an exception or unpredictable behavior.

20.2 va_start Macro

Synopsis

```
#include <stdarg.h>

  va_start (ap,parmN)

   va_list ap;
```

Description

The va_start macro is invoked before any access to the unnamed argument. This macro initializes ap for subsequent use by the va_arg or va_end macros.

If the parameter parmN is declared with the register storage-class, a function type, an array type, or a type not compatible with the promote type usually done in the C language, an exception or an undefined behavior can occur at execution time. For example, float parameters are promoted to double type, and char or short parameters are promoted to int type.
The va_start macro returns no value.

20.3 va_arg Macro

Synopsis

```
#include <stdarg.h>

#include <stdarg.h>

 type va_arg(ap, type)

 va_list ap;
```

Description

- The va_arg macro expands to an expression that has the type and the value of the next argument in the call.
- The ap parameter is the same as the one initialized by the va_start macro.
- Successive invocation of the va_arg macro modifies ap so that the values of successive arguments are returned in turn.
- The type parameter is a type name that must be compatible with the type of the desired argument. If it is not, the returned value is undefined, which also occurs if there is no more following argument.
- The first invocation of va_arg macro returns (only after va_start invocation) the value of the argument after that specified by parmN. Successive invocations return the values of the next parameters in their respective order.

20.4 va_end Macro

Synopsis

```
#include <stdarg.h>

  void va_end(ap)

   va_list ap;
```

Description

The va_end macro modifies the ap parameter, which is initialized by the va_start macro so that it is no longer usable. Using this macro after invocation creates an exception at execution time.

The va_end macro returns no value.

21. Diagnostics

21.1 The <ASSERT_H> Header Subfile

The header file <assert_h> declares one function (assert) and refers to one macro (NDEBUG) which must be defined in the user source for the assert function to have any effect.

Synopsis

```
#include <assert_h>

assert (integer_expression)
```

Description

Assert is used for diagnostics in C programs. The argument is an integer expression to be tested. If, at execution, the expression is false, (which means it returns zero), an error message is sent in the following form:

```
Assertion failed: integer expression, file xyz, line nnn
```

This message is sent to the standard error file stderr and the exit (0) function is called.

If the expression is true, assert returns no value.

22. Signal

22.1 What is a Signal?

A signal is a classification of exceptions that occur during program execution. These exceptions require specific treatment, and a signal is a class of all the exceptions that require the same treatment. Each signal has its own treatment definition.

The C ANSI STANDARD defines six signals. They are as follows:

SIGABRT	Abnormal termination, as initiated by the abort function.
SIGFPE	An erroneous arithmetic operation, such as dividing by zero or an operation resulting in overflow.
SIGILL	Detection of an invalid function image, such as an illegal instruction.
SIGINT	Receipt of an interactive attention signal (break).
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

22.2 Description of a Signal

```
#include <signal_h>

void (*signal (int sig, void (*func) (int))) (int);
```

The signal function determines how to handle the signal number. The signal number is called sig. The definition of this function is as follows:

sig	This is the signal number. It has one of the definitions described below.
func	The standard header file SIGNAL_H can define one of the two following values for this: SIG_DFL: This is the default value used in GCOS 7 system exception handling. SIG_IGN: Ignore the signal.

If the standard header file does not define these values, the func parameter points to the function to call when the sig occurs. Then, when sig is raised, the default handling is reset for sig and the user function is called. This executes the equivalent of signal (sig, SIG_DFL), followed by the execution of the equivalent of (*func)(sig).

The program startup executes the equivalent of signal (sig, SIG_DFL) for all sig values.

The signal function returns the value of func for the previous call to signal for the specified signal sig, or a value of SIG_ERR if sig is not a known signal. In that case errno is set to ENOSUCHSIG. There is only one function at a time linked to a signal, because the previous one is lost and returned by the signal function. The returned function cannot be called if the value is SIG_DFL, SIG_IGN or SIG_ERR

```
#include <signal_h>

int raise (int sig);
```

The raise function sends the signal sig to the executing program. It returns zero if successful, or a nonzero value otherwise. The two signals SIGABRT and SIGTERM can be raised only by this function, as ANSI STANDARD does not require the system to raise all the signals.

22.3 Writing a Signal Handler

A signal handler is the user function to be called when a signal occurs.

Signal handlers are normal C functions that take an argument that is the raised signal. They do not return a value. This function performs the required action and then either returns or calls the abort, exit, or longjmp function. If it does return, the program resumes at the point it was interrupted. Because this function can be called on asynchronous events, the following rules must be respected:

- This function can store a value into an object with static storage duration only if it is of type volatile sig_atomic_t. Using an object with volatile storage implies that the program must be compiled at a level of ANSI or GANSI.
- This function does not call any function in the standard library other than the function signal itself, which allows the function to reset the signal. If a new signal handler is set while still in the current signal handler, it is available only at the end of the current signal handler.

The function may terminate by executing a return statement or by calling the abort, exit or longjmp function. If the function terminates by a return statement, the program resumes at the instruction after the point it was interrupted.

22.4 The Signal Handler Mechanism

The exception mechanism in GCOS 7 calls a signal handler. The function main must be called to initialize this mechanism.

The exceptions in GCOS 7 are divided into classes and types. The GCOS 7 exceptions and the exceptions that the Run-Time package recognizes are mapped as follows:

```
SIGILL   : Class 9 - all types.
           Class 12 - types 3 to 6.

SIGSEGV  : class 6 - types 0 and 2.
           class 10 - type 0.
           class 12 - types 0 to 2.
           class 17 - type 2.

SIGFPE   : class 16 - types 0 to 2.
           class 17 - types 0 and 1.

SIGINT   : Break signal in IOF environment.
```

Only the raise function can emit the two other signals SIGABRT and SIGTERM.

22.5 Limitations of the Signal Handler

GCOS 7 places the following limitations on the exception mechanism:

- The stack-frame can not be multi-segmented or greater than 64K.
- A signal is raised only for the duration of its process.

22.6 Example

The following is an example of signal definition and handling in the C language.

```
#include <STDIO_H>
#include <STDLIB_H>
#include <SIGNAL_H>

static volatile sig_atomic_t res_fpe;

void handler_sigfpe (int sig) {
   if (sig != SIGFPE) abort ();
   res_fpe = 1;
   if (signal (sig, handler_sigfpe) != SIG_DFL)
      abort();
}

main () {
int a;

if (signal (SIGFPE, handler_sigfpe) != SIG_DFL)
   abort ();
res_fpe = 0;
a = a / 0;
if (res_fpe) printf ("Zero divide.\n");
else printf ("SIGFPE not raised on zero divide!\n");
res_fpe = 0;
if (raise (SIGFPE)) abort ();
if (res_fpe) printf ("SIGFPE raised.\n");
else printf ("SIGFPE not raised by the raise "
             "function!\n");

if (signal (SIGFPE, SIG_IGN) != handler_sigfpe)
   abort ();
res_fpe = 0;
a = a / 0;
if (~res_fpe) printf ("Zero divide ignored.\n");
else printf ("SIGFPE not ignored on zero divide!\n");
}
```

The execution that results from this example is as follows:

```
Zero divide.
SIGFPE raised.
Zero divide ignored.
```

23. Reporting Error Conditions

23.1 The <ERRNO_H> Header Subfile

The standard header file ERRNO_H defines the variable errno and several macros relating to the reporting of error conditions. The errno variable is set to 0 at program startup, which is at the main function. The value of errno can be set to nonzero by a library function, but no library function can set it to 0.

23.2 Description

The list below gives the correspondence between the error codes defined in ERRNO_H and their meaning. Note that some of these codes may actually be not returned by any library functions; they are in ERRNO_H only for reasons of compatibility, for future use, or both.

EACCESS	: Access denied
EBADBASE	: Wrong numeric base
EBADDATE	: Wrong date value
EBADF	: Bad file number
EDOM	: Domain error
EFAULT	: Reserved
EFAULTPTR	: Invalid pointer
EFILESIZE	: Wrong file size
EINVAL	: Invalid argument
EMFILE	: Too many open files
ENOBASE	: Wrong pointer
ENODEV	: No such device
ENOENT	: No such file
ENOEXEC	: Exec format error
ENOFSETPOS	: File positionning failure
ENOMEM	: Not enough core

ENOMEMB	: Element number not provided
ENOSIZE	: Size not provided
ENOSR	: Invalid file position
ENOSUCHSIG	: Wrong signal
ERANGE	: Result out of range
EXDEN	: Cross-device link
E2BIG	: Reserved

24. Localization

24.1 What is Localization?

Localization provides a set of facilities that interpret orthographical differences between the languages of the world. These orthographical differences include those of alphabet, collation, formatting numbers and currency, date format, and time format.

The C language uses localization to deal with these differences. Localization operates only at run-time execution, and the source program in the C language remains in English. A called function determines which specific locale for the localization to interpret. The function allows the localization to recognize language differences.

The orthographic differences that localization interprets are as follows:

- Alphabet. The English language uses the 26 letters derived from the Latin alphabet. Some European languages add other characters to this alphabet, while other languages use non-Latin alphabets. Also, different languages do not always use the same upper-case and lower-case for all characters.
- Collation. A machine sort can be successful in both ASCII and EBCDIC for ordering strings. However, some European languages use punctuation codes for alphabetic characters, and their ordering is not alphabetic. For example, in Spanish, "ll" sorts as a single letter following "l".
- Number and Currency Format. Some countries use a period as the decimal point, and a comma to separate units of thousands (groups of three digits). Other countries use just the opposite. In the United States the period is used for the decimal point; some European countries use a comma. Units of thousands are separated by a comma in the United States, while a period is common elsewhere. In printing currency amounts, the currency symbol can precede, follow, or be embedded in the digits.
- Date and Time Format. There are several formats for presenting the date and the time, even within the same country.

24.2 Run Time Package Functions and Localization

This subsection describes the effect that localization has on the Run Time Package (RTP) functions.

Strings

The following functions provide a locale for sorting strings:

```
strxfrm
strcoll
```

Character Classification

The following functions provide a classification for characters:

```
isalnum    isgraph    ispunct    tolower
isalpha    islower    isspace    toupper
iscntrl    isprint    isupper
```

Multibyte Characters

The following functions provide basic transformations on multibyte characters:

```
mblen    mbstowcs
mbtowc   wcstombs
wctomb
```

Decimal Point and isspace Function

The following functions are formatted input and output functions that are affected by the value of the decimal point and the isspace function:

```
scanf
printf
```

Time and Date

The following function provides a formatted output of the time and the date:

```
Strftime
```

24.3 Localization Functions

24.3.1 setlocale

Synopsis

```
#include <LOCALE_H>

char *setlocale (int category, const char *locale);
```

Description

This function selects the portion of the program's locale that the category and locale arguments specify. This function returns a pointer to a string that contains the newly-installed localization, or a NULL pointer if the selection is not possible.

If locale is a NULL pointer, this function returns a pointer to the string that contains the current localization associated with the specified category.

A subsequent call restores that part of the program's locale.

Valid values for category are as follows:

LC_COLLATE	Affects the strcoll and strxfrm functions.
LC_CTYPE	Affects all the character handling and multibyte functions, except for the following: isdigit, isxdigit, toupper, tolower, mblen, mbtowc, wctomb, wcstombs, mbstowcs.
LC_MONETARY	Affects the monetary formatting information that the localeconv function returns.
LC_NUMERIC	Affects the decimal-point character for the formatted input/output functions, the string conversion functions, and the non-monetary formatting information that the localeconv function returns.
LC_CTIME	Affects the strftime function.
LC_ALL	Affects entire locale of the program.

Locale is the name of the localization to be installed. It is a character string of up to 12 characters.

Diagnostics

This function returns a NULL pointer if the selection cannot be honored. The locale of the program is not changed.

24.3.2 localeconv

Synopsis

```
# include <LOCALE_H>

struct lconv *localeconv(void);
```

Description

The localeconv function returns an object that describes how to interpret numeric formats (including monetary) of the current locale. A subsequent call can overwrite the structure to which the returned value points.

Members that are of type char * are pointers to strings, any of which (except decimal_point) can point to "" (a blank) to indicate that the value is either not available in the current locale or is of length zero.

Members that are of type char with a non-negative value indicate that the value is not available in the current locale. Any of these members can be CHAR_MAX.

The members are as follows:

char *decimal_point	The decimal-point character that formats numeric quantities that are non-monetary.
char *thousands_sep	The character that separates groups of digits before the decimal_point character in formatted numeric quantities that are non-monetary.
char *grouping	A string whose elements indicate the size of each group of digits in formatted numeric quantities that are non-monetary.
char *int_curr_symbol	The international currency symbol that applies to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217 Codes for the Representation of Currency and Funds. The fourth character separates the international currency symbol from the amount of the money.

char *currency_symbol	The local currency symbol for the current locale.
char *mon_decimal_point	The decimal point that formats monetary quantities.
char *mon_thousands_sep	The separator for groups of digits before the decimal-point in formatted monetary quantities.
char *mon_grouping	A string whose elements indicate the size of each group of digits in formatted monetary quantities.
char *positive_sign	The string that indicates a formatted monetary quantity whose value is non-negative.
char *negative_sign	The string that indicates a formatted monetary quantity whose value is negative.
char int_frac_digits	The number of decimal places to display after the decimal point in an internationally-formatted monetary quantity.
char frac_digits	The number of decimal points to display after the decimal-point in a formatted monetary quantity.
char p_cs_precedes	Set to 1 when the currency_symbol precedes the values for a formatted monetary quantity that is nonnegative. Otherwise, set to 0.
char p_sep_by_space	Set to 1 when the currency_symbol is separated by a space from the values for a formatted monetary quantity that is nonnegative. Otherwise, set to 0.
char n_cs_precedes	Set to 1 when the currency_symbol precedes the values for a formatted monetary quantity that is negative.
char n_sep_by_space	Set to 1 when the currency_symbol is separated by a space from the values for a formatted monetary quantity that is negative. Otherwise, set to 0.
char p_sign_posn	Indicates the position of the positive_sign for a formatted monetary quantity that is nonnegative.
char n_sign_posn	Indicates the position of the negative_sign for a formatted monetary quantity that is negative.

The elements of grouping and mon_grouping are as follows:

CHAR_MAX	No further grouping is performed.
0	The remainder of the digits use the previous element.
other	The number of digits that comprise the current group. The next element determines the size of the group of digits before the current group.

The values of p_sign_posn and n_sign_posn are as follows:

0-	Parentheses surround the quantity and currency_symbol.
1-	The sign string precedes the quantity and currency_symbol.
2-	The sign string follows the quantity and currency_symbol.
3-	The sign string immediately precedes the currency_symbol.
4-	The sign string immediately follows the currency_symbol.

24.4 Multibyte Functions

The C language provides some basic functions to process multibyte characters. These functions are not available by default; they require the following:

- A localization that includes multibyte characters.
- The command SEGTABi = (SHRLEVEL=2, VSEG=n). Linking requires this command because localization uses a large segment when it loads.
- A call to the setlocale function using the user localization and the category LC_CTYPE or LC_ALL. This call installs the multibyte environment, provided that the multibyte mode is on.
- Three macros, defined in the standard include STDIO_H. These macros allow the multibyte-mode to be modified. The macros are as follows:

```
set_multibyte_mode();
cancel_multibyte_mode();
test_multibyte_mode();
```

Synopsis

```
#include <STDLIB_H>
int mblen (const char *s, size_t n);
int mbtowc (wchar_t *pwc, const char *s, size_t n);
int wctomb (char *s, wchar_t wchar);
size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

Description

mblen	This function returns the number of bytes comprising the multibyte character to which s points, or it returns -1 if *n* or fewer number of bytes do not form a valid multibyte character.
mbtowc	This function converts the multibyte character to which s point into a value of type wchar_t and stores it in the object pointed to by pwc. No more than *n* number of bytes of the array to which s points are examined. This function returns the number of bytes comprising the multibyte character to which s points, or it returns -1 if the *n* or fewer number of bytes do not form a valid multibyte character.

wctomb	This function converts the code whose value is wchar into a multibyte character and store it in the object to which s points. The most number of characters stored is the value of MB_CUR_MAX. It returns the number of bytes that comprise the multibyte character corresponding to the value of wchar, or it returns -1 if the value does not correspond to a valid multibyte character.
mbstowcs	This function converts a sequence of multibyte characters from the array to which s points into a sequence of corresponding codes. It stores not more than *n* number of codes into the array to which pwcs points, and does not examine multibyte characters that follow a null character (which is converted into a code with value zero). This function returns the number of codes stored into the array to which pwcs points. This does not include any existing terminating zero code or (size_t)-1 if an invalid multibyte character is encountered.
wcstombs	This function converts a sequence of codes from the array to which pwcs points into a sequence of multibyte characters. It then stores these multibyte characters into the array to which s points, stopping if a multibyte character exceeds the limit of *n* total bytes or if a null character is stored. It returns the number of bytes stored into the array to which s points. This does not include any existing terminating null character or (size_t)-1 if a code that does not correspond to a valid multibyte character is encountered.

24.5 Default Localization

Valid values for the locale parameter of the setlocale function are as follows:

"C"	The minimal environment
""	The native environment

At program startup, the C locale is default. It is the minimal C environment, as follows:

Collating sequence	EBCDIC collating sequence.
Character conversion type	See the section describing general I/O considerations.
Monetary and Numeric Format	The '.' symbol is the decimal_point value. No other format information is available.

Date and Time Format	The date and time format is:
	`Sun Sep 16, 1973`
	`14:46:03`
	The 12-hour clock format is:
	`AM and PM`

The native environment provides a national localization, which must be specified. The next paragraph describes how to add this localization.

24.6 Introducing New Localization

Many localizations can be introduced. This section describes the format for LC_MONETARY, LC_NUMERIC or LC_TIME category. There is not yet any standard description of a localization, so other localization for the LC_COLLATE and LC_CTYPE categories has yet to be added.

For LC_MONETARY and LC_NUMERIC, the locale must be named with no more than 12 characters. Files must be added in the SYS.C.INCLUDE system file with these names:

```
TABLE_locale_NUMERIC for LC_NUMERIC
TABLE_locale_MONETARY for LC_MONETARY
TABLE_locale_TIME for LC_CTIME
```

For native localization (the "" locale), the tables are named as if the locale were "_". Each category uses a different format for the information in these files.

The format for the information in the LC_MONETARY category is as follows:

```
<int_curr_symbol>
<currency_symbol>
<mon_decimal_point>
<mon_thousand_sep>
<mon_grouping>
<positive_sign>
<negative_sign>
<frac_digits>
<p_cs_precedes>
<p_sep_by_space>
<n_cs_precedes>
<n_sep_by_space>
<p_sign_posn>
<n_sign_posn>
```

The format for the information in the LC_NUMERIC category is as follows:

```
<decimal_point>
<thousands_sep>
<grouping>
```

The format for the LC_CTIME category is as follows:

```
<abbreviated weekday names>
<full weekday names>
<abbreviated month names>
<full month names>
<appropriate date representation>
<appropriate time representation>
PM=<equivalent of pm(post-meridiem) or blank character>
AM=<equivalent of am(ante-meridian) or blank character>
```

A colon separates each weekday and month, and a period follows the last entry.

The date and time representation have the same format as for the strftime argument, except that formats %c, %x and %X are not allowed.
Each piece of information is contained in one and only one record.

LOCALIZATION EXAMPLE 1:

This example shows localization for France. The format is for numbers, money, date, and time. The number amount, money amount, time, and date are as follows:

```
12.456,3456
F  12.456,50
18 Août 1987
16:54:03
```

- For the number format, the resulting file is:

```
TABLE_FRANCE_NUMERIC :
,
.
\3
```

- For the currency format, the file is:

```
TABLE_FRANCE_MONETARY :
FFR
F
,
.
\3

-
2
2
0
1
0
1
1
1
```

- For the date and time format, the resulting file is:

```
TABLE_FRANCE_TIME :
Dim:Lun:Mar:Mer:Jeu:Ven:Sam.
Dimanche:Lundi:Mardi:Mercredi:Jeudi:Vendredi:Samedi.
Jan:Fev:Mars:Avr:Mai:Juin:Juil:Août:Sep:Oct:Nov:Dec.
Janvier:Fevrier:Mars:Avril:Mai:Juin:Juillet:Août:Septembre:
Octobre:Novembre:Decembre.
%d %B %Y
%H:%M:%S
PM=PM
AM=AM
```

❑

LOCALIZATION EXAMPLE 2:

This example describes how to use the new localization made in the previous example. This is after it is stored in the SYS.C.INCLUDE library.

```
#include <stdio_h>
#include <locale_h>
#include <time_h>
#define LG_RES 100
main () {
time_t *today;
struct tm *loc_time;
char fmt_time[LG_RES];

   if (time(today) == -1) {
       printf("Calendar time not available.\n");
       exit (10000);
   }
   loc_time = localtime (today);

   if (strftime(fmt_time, LG_RES,
            "Localization C\n"
            "   Date : %x\n"
            "   Time : %X\n",
            loc_time))
   printf ("%s",fmt_time);
   if (setlocale (LC_TIME,"FRANCE") == NULL) {
      printf("Localization FRANCE doesn't exist"
             " for category LC_TIME\n");
      exit (10000);
   }
   if (strftime(fmt_time, LG_RES,
            "Localization FRANCE\n"
            "   Date : %x\n"
            "   Time : %X\n",
            loc_time))
   printf ("%s",fmt_time);
}
```

❑

25. Standard Definition Header File

25.1 The <STDDEF_H> Header Subfile

The STDDEF_H standard header file works with the definitions of three previously-defined C types, the NULL macro, and the offsetof macro.

25.2 The Previously-defined C Types

The C types that the STDDEF_H header file works with are:

ptrdiff_t	The signed integral type of the result of subtracting two pointers.
size_t	The unsigned integral type of the result of the sizeof operator.
wchar_t	An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales. For more information, see the chapter that describes localization.

25.3 The NULL Macro

The NULL macro expands to the value 0. It uses the offsetof macro.

25.4 The OFFSETOF Macro

The offsetof macro is as follows:

```
offsetof (type, member-designator)
```

This offset value then expands to an integral constant expression that has type size_t, whose value represents the offset measured in bytes. This measurement is from the beginning of its structure to the structure member. The member-designator designates the beginning of the structure, and the type designates the structure member.

The member-designator is as follows:

```
static type t;
```

From this, the following expression evaluates an address constant.

```
&(t.member-designator)
```

If member-designator is a bit-field, the behavior is undefined.

A. File and Volume Syntax

A.1 Syntax of a File Literal

```
file-literal     ::= local-file
file-literal     ::= remote-file

local-file       ::= cataloged-file
local-file       ::= temporary-file
local-file       ::= uncataloged-file

cataloged_file   ::= path-name[/g-suffix][..subfile]
                     [$CAT[i]]

                                          {:md[/md].  ..:dvc  }
temporary-file   ::= path-name[..subfile] [ {$RES              }  ] $TEMPRY
                                          {$VOLSET : [ name6] }

uncataloged-file ::= path-name[..subfile]
                     [:md[/md]...:dvc][suffix1]
uncataloged-file ::= DUMMY
uncataloged-file ::= SYS.OUT
uncataloged-file ::= *:md[/md]...:dvc[suffix2]
uncataloged-file ::= [path-name]:[md]:TN[$UNCAT]

suffix1          ::= $RES
suffix1          ::= suffix2
suffix1          ::= suffix3

suffix2          ::= $MFT
suffix2          ::= $MFTi
suffix2          ::= $MFT+
suffix2          ::= $UNCAT

suffix3          ::= $NONE
suffix3          ::= $NSTD
```

The V7 release does not support the remote-file syntax ($site-name:local-file).

A.2 Syntax of a Volume Literal

```
volume-literal     ::= { md[/md]...:dvc[suffix3] | $VOLSET [:name6] }
```

For more detail about file or volume literal, refer to the *IOF Terminal User's Reference Manual*.

Index

#

_

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X

Technical publication remarks form

Title :	DPS7000/XTA NOVASCALE 7000 C Language User's Guide Languages: C

Reference N° :	47 A2 60UL 06	**Date:**	**February 2005**

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 Dept.
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
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Phone: +33 (0) 2 41 73 72 66
FAX: +33 (0) 2 41 73 70 66
E-Mail: srv.Duplicopy@bull.net

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 60UL 06