FORTRAN 77

User’s Guide

Languages: FORTRAN

REFERENCE
47 A2 16UL 03
DPS7000/XTA
NOVASCALE 7000
FORTRAN 77
User’s Guide

Languages: FORTRAN

July 1990

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

REFERENCE
47 A2 16UL 03
The following copyright notice protects this book under Copyright laws which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

Copyright © Bull SAS 1990
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**

We acknowledge the right of proprietors of trademarks mentioned in this book.

Intel® and Itanium® are registered trademarks of Intel Corporation.

Windows® and Microsoft® software are registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Linux® is a registered trademark of Linus Torvalds.

The information in this document is subject to change without notice. Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.
MANUAL OBJECTIVES

This manual provides information about FORTRAN 77, as it is implemented under the GCOS 7 operating system.

It describes how to compile, link, execute, debug and maintain FORTRAN 77 programs with a maximum of efficiency. This manual complements the FORTRAN 77 Reference Manual which contains a formal specification of the FORTRAN 77 Programming Language.

INTENDED READERS

This manual is intended for all those wishing to program applications in FORTRAN 77 within the GCOS 7 environment. This manual assumes that the reader is familiar with the FORTRAN 77 language and has a general knowledge of GCOS 7.

STRUCTURE OF THE MANUAL

Section 1 gives an overview of the GCOS environment of FORTRAN 77 programs.

Sections 2 and 3 describe the compilation and linkage of FORTRAN 77 programs.

Section 4 describes execution, debugging and maintenance.

Section 5 discusses the representation of data in memory and the effects of the ROUND compilation parameter.

Section 6 discusses calling and called programs, and recursion.

Section 7 discusses segmentation for Virtual Memory Management, and the effects of the PSEGMAX, DSEGMAX and DEBUG parameters.
Section 8 describes various programming techniques for reducing the size and increasing the execution speed of FORTRAN 77 programs.

Section 9 discusses file usage.

Section 10 describes the standard record formats accepted by Data Management.

Section 11 describes the use of unit record files.

Section 12 discusses the difference between GCOS FORTRAN and GCOS FORTRAN 77, and the effects of the LEVEL=GCOS1E compilation parameter.

Section 13 discusses modular programming and protection, and the effects of the SUBCK compilation parameter.

Section 14 discusses optimizing techniques and the effects of the OPTIMIZE=n compilation parameter.

Appendix A gives a list of the FORTRAN 77 intrinsic functions; whether they are implemented by firmware and whether they can be passed directly as arguments to a called procedure.

ASSOCIATED DOCUMENTS

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

- For an overall description of GCOS functions
  System Overview........................................................................................ 47 A2 04UG

- For GCOS7 JCL functions
  JCL Reference Manual................................................................................. 47 A2 11UJ
  JCL User Guide ........................................................................................... 47 A2 12UJ

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

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

- 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 FORTRAN 77 language definition, file naming and intrinsic functions
  
  FORTRAN 77 Reference Manual .................................................................. 47 A2 15UL

- For data management utilities and for extended JCL statements
  
  Data Management Utilities Manual (VBO) ...............................................47 A2 05UF
  Data Management Utilities (FBO)................................................................47 A2 26UF

- For measuring system performance
  
  SBR User's Guide.......................................................................................47 A2 03US

- For diskette file handling
  
  Diskette/Stream Reader Facility User Guide..............................................47 A2 06UU

- For Virtual Memory Management (V3A systems)
  
  System Administrator's Manual ..................................................................47 A2 01US

- For Virtual Memory Management (V3B systems)
  
  GCOS7-V3B System Administrator's Manual ..............................................47 A2 08US

- For Virtual Memory Management (V5 systems)
  
  GCOS7-V5 System Administrator's Manual ................................................47 A2 10US

- Fodebugging purposes
  
  Error Messages and Return Codes .............................................................47 A2 10UU
  Program Checkout Facility User's Guide .....................................................47 A2 15UP

- For file management and access
  
  UFAS-EXTENDED User's Guide.................................................................47 A2 04UF
  GAC-EXTENDED User's Guide ..................................................................47 A2 12UF
SYNTAX NOTATION

The commands use the following syntax:

ITEM

An item in upper case is a name or keyword and is entered literally as shown. The upper case is merely a convention; in practice you can specify the item in upper or lower case.

item

An item in lower case indicates that a user-supplied value is expected.

In most cases it gives the type and maximum length of the value:

- char105 a string of up to 105 alphanumeric characters
- name31 a name of up to 31 characters
- lib78 a library name of up to 78 characters
- file78 a file name of up to 78 characters

In some cases, it gives the format of the value:

- a means a single alphabetic character
- nnn means a 3-digit number
- hh.mm means a time in hours and minutes

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

- device-class
- condition
- any-characters

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

An item enclosed in square brackets is optional.

ITEM

An underlined item is a default value. It is the value assumed if none is specified.

<item>

Angle brackets indicate a single key on the micro computer.

=., $ *, /, \n
Enter these special non-alphabetic characters as shown.

Examples:

(1) WHEN = \{IMMED \| \{dd.mm.yy. \| hh.mm + nnn \{W | D | H | M \} item \}

This means you can specify:

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

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

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

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

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

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

<F10> refers to the F10 key

<shift> refers to the shift key
FORTRAN 77
# Table of Contents

1. **Introduction** ............................................................................................................... 1-1

1.1 **GCOS FUNCTIONS** ............................................................................................... 1-1

1.2 **JOB CONTROL LANGUAGE** .................................................................................. 1-1

1.3 **JOBS AND STEPS** ................................................................................................... 1-2

1.4 **SYSIN AND SYSOUT** ............................................................................................ 1-3

1.5 **GCOS7 COMMAND LANGUAGE** .......................................................................... 1-3

2. **Compilation** ............................................................................................................. 2-1

2.1 **THE FOR77 JCL STATEMENT** ............................................................................. 2-1

2.1.1 The F7C Abbreviation .......................................................................................... 2-2

2.1.2 Where to Place the FOR77 Statement .................................................................. 2-3

2.1.3 Parameters of FOR77 Compilation ...................................................................... 2-4

2.1.3.1 SOURCE, INFILE, INLIB and INLIBn Parameters .............................................. 2-4

2.1.3.2 CULIB Parameter ............................................................................................. 2-5

2.1.3.3 DEBUG and NDEBUG Parameters ..................................................................... 2-6

2.1.3.4 DEBUGMD and NDEBUGMD Parameters ......................................................... 2-7

2.1.3.5 DSEGMAX Parameter ...................................................................................... 2-8

2.1.3.6 PSEGMAX Parameter ...................................................................................... 2-9

2.1.3.7 OBJ and NOBJ Parameters .............................................................................. 2-9

2.1.3.8 LEVEL Parameter ............................................................................................ 2-9

2.1.3.9 LIST and NLIST Parameters ............................................................................ 2-10

2.1.3.10 XREF and NXREF Parameters ..................................................................... 2-10

2.1.3.11 SILENT and NSILENT Parameters .................................................................. 2-11

2.1.3.12 MAP and NMAP Parameters ........................................................................ 2-11

2.1.3.13 OBSERV and NOBSERV Parameters ............................................................. 2-11

2.1.3.14 WARN and NWARN Parameters .................................................................... 2-12

2.1.3.15 PRTFILE Parameter ...................................................................................... 2-12
| 2.1.3.16 | PRTLIB Parameter | 2-13 |
| 2.1.3.17 | SIZEOPT Parameter | 2-13 |
| 2.1.3.18 | SUBCK and NSUBCK Parameters | 2-13 |
| 2.1.3.19 | ROUND and NROUND Parameters | 2-14 |
| 2.1.3.20 | INIT Parameter | 2-14 |
| 2.1.3.21 | ADP and NADP Parameters | 2-15 |
| 2.1.3.22 | CODE Parameter | 2-15 |
| 2.1.3.23 | PI and NPI Parameters | 2-15 |
| 2.1.3.24 | OPTIMIZE Parameter | 2-16 |
| 2.1.3.25 | PACKAGE Parameter | 2-16 |
| 2.2 | JCL STATUS | 2-17 |
| 2.3 | SERIAL COMPILATION | 2-18 |
| 2.4 | INTERACTIVE COMPILATION | 2-18 |
| 2.5 | BATCH COMPILATION: FORTRAN SOURCE FORMAT | 2-19 |
| 2.5.1 | FORTRAN Source on Library Member (or a Sequential File) | 2-19 |
| 2.5.2 | FORTRAN Source on Cards | 2-20 |
| 2.5.3 | Input Enclosure in a Library Member | 2-20 |
| 2.6 | SUMMARY OF COMPILATION | 2-22 |
| 3. | Linking | 3-1 |
| 3.1 | SEGMENT NUMBERS | 3-1 |
| 3.2 | LINKER JCL STATEMENT | 3-2 |
| 3.2.1 | Load-Module-Name Parameter | 3-3 |
| 3.2.2 | INLIB Parameter | 3-4 |
| 3.2.3 | OUTLIB Parameter | 3-5 |
| 3.2.4 | COMMAND and COMFILE Parameters | 3-5 |
| 3.2.5 | ENTRY Parameter | 3-6 |
| 3.2.6 | PRTFILE Parameter | 3-6 |
| 3.2.7 | PRTLIB Parameter | 3-6 |
| 3.2.8 | STEPOPT Parameter | 3-7 |
| 3.3 | SERIAL LINKAGE | 3-8 |
| 3.4 | INTERACTIVE OPERATION | 3-9 |
| 3.5 | LINKER COMMANDS | 3-10 |
Table of Contents

3.5.1 ENTRY Command ................................................................. 3-10
3.5.2 INCLUDE Command ............................................................. 3-10
3.5.3 STACK3 Command ............................................................... 3-11
3.5.4 SEGTAB1 Command ............................................................. 3-11
3.5.5 FILE Command ................................................................. 3-12

3.6 PRINTER OUTPUT ................................................................. 3-12
3.6.1 Segment List ................................................................. 3-13
3.6.2 Linkage Report ............................................................... 3-14
3.6.3 Error Messages .............................................................. 3-14

4. Execution and Debugging ....................................................... 4-1
4.1 STEP EXECUTION .............................................................. 4-1
4.2 EXECUTION IN INTERACTIVE MODE ..................................... 4-3
4.3 EXECUTION IN BATCH MODE ............................................... 4-5
4.4 OPTIONS STRING ............................................................. 4-6
4.5 JCL STATEMENTS USING PERMANENT LIBRARIES .............. 4-7
4.6 BATCH OR INTERACTIVE DEBUGGING .................................. 4-8
4.7 EXCEPTION CONTROL IN INTERACTIVE MODE ..................... 4-9
4.8 DEBUGGING AND MAINTENANCE WITH PERMANENT SOURCE CODE .... 4-11

5. Representation and Manipulation of Data by the Hardware ........ 5-1
5.1 DATA REPRESENTATION ................................................... 5-1
5.1.1 Format of Information in Main Storage ................................ 5-1
5.1.2 Short Integer Data ......................................................... 5-2
5.1.3 Integer Data ............................................................... 5-2
5.1.4 Real Data ................................................................. 5-2
5.1.5 Double Precision Data .................................................. 5-3
5.1.6 Quadruple Precision Data ............................................. 5-4
5.1.7 Character Data ............................................................ 5-6
5.1.8 Logical Data .............................................................. 5-6
5.1.9 Short Logical Data ......................................................... 5-6
## 5.1.10 Complex Data ........................................................................................................ 5-7
## 5.1.11 Complex Double Precision Data ........................................................................ 5-7

## 5.2 FLOATING POINT OPERATIONS ........................................................................ 5-8
### 5.2.1 Scientific Registers .......................................................................................... 5-8
### 5.2.2 Characteristics of Floating Point Operations .................................................... 5-9
### 5.2.3 Locations of Operands and Results .................................................................... 5-9
### 5.2.4 Guard Digit ....................................................................................................... 5-9
### 5.2.5 Normalization .................................................................................................... 5-11
### 5.2.6 Underflow ......................................................................................................... 5-11
### 5.2.7 Overflow ........................................................................................................... 5-11

## 6. Calling and Called Programs and Recursion Phenomena .................................. 6-1
### 6.1 LINKAGES SUPPORTED ....................................................................................... 6-1
### 6.2 LINKAGES NOT SUPPORTED ............................................................................... 6-2
### 6.3 CORRESPONDANCE BETWEEN FORTRAN, COBOL, AND GPL DATA TYPES ........................................ 6-3
### 6.4 FORTRAN 77, COBOL, AND GPL STORAGE DATA .................................... 6-4
### 6.5 PASSING ARGUMENTS FOR DPS 7 FORTRAN 77 ........................................ 6-6
### 6.6 FORTRAN 77, COBOL, AND GPL ARGUMENT CORRESPONDENCE ............ 6-8
### 6.7 RETURNING FUNCTION VALUES ...................................................................... 6-11
### 6.8 FORTRAN 77 AND GPL FUNCTIONS .............................................................. 6-12
### 6.9 EXCHANGE OF FILES BETWEEN FORTRAN 77 AND COBOL .................... 6-14
#### 6.9.1 Files Preconnected to System Units ............................................................... 6-14
#### 6.9.2 Other Shareable Files .................................................................................... 6-15
### 6.10 EXCHANGE OF FILES BETWEEN FORTRAN 77 AND GPL ..................... 6-17
### 6.11 JCL COMMUNICATION ..................................................................................... 6-17
### 6.12 RECURSIVITY OF FORTRAN 77 PROCEDURES .......................................... 6-18
# Table of Contents

6.13 REENTRANCE CONDITIONS OF A FORTRAN 77 PROCEDURE ..................................6-19

7. Segmentation and Addressing .................................................................7-1

7.1 GENERATED SEGMENTS ........................................................................7-2

7.1.1 PSEGMAX, DSEGMAX, and DEBUG Parameters are Not Used ............7-2
7.1.2 PSEGMAX or DSEGMAX Parameters are Specified ..............................7-3
7.1.3 DEBUG Parameter is Specified ..........................................................7-3
7.1.4 Evaluating the Number of Segments ..................................................7-3
7.1.5 Guideline for Segmentation ..............................................................7-4

7.2 DATA ADDRESSING ................................................................................7-6

7.2.1 Alignment ...........................................................................................7-6
7.2.2 Indexation ............................................................................................7-6
7.2.3 Allocation Performed by the Compiler ................................................7-7

8. Efficiency Techniques ..............................................................................8-1

8.1 GENERAL GUIDELINES ..........................................................................8-2

8.2 PARAMETERS OF THE FOR77 JCL STATEMENT .......................................8-4

8.3 SPECIFICATION STATEMENTS ............................................................8-5

8.3.1 Data Type ............................................................................................8-5
8.3.2 Constant Values ..................................................................................8-5
8.3.3 Arrays ..................................................................................................8-5
8.3.4 Alignment of Data ..............................................................................8-6
8.3.5 COMMON Blocks, DSEGMAX Parameter ...........................................8-6

8.4 EXECUTABLE STATEMENTS ....................................................................8-7

8.4.1 Label .....................................................................................................8-7
8.4.2 Array References ...............................................................................8-7
8.4.3 Intrinsic Functions .............................................................................8-8
8.4.4 CALL ....................................................................................................8-8

8.5 INPUT/OUTPUT .......................................................................................8-9

8.5.1 Fast READ/WRITE ............................................................................8-9
8.5.2 OPEN Statement ................................................................................8-9
8.5.3 Characteristics of READ/WRITE Statements ......................................8-10
9. **Files** ................................................................................................................................. 9-1

9.1 **FILE NAMES** ................................................................................................................... 9-1

9.1.1 FORTRAN Unit Name ....................................................................................................... 9-1

9.1.2 Internal File Name ............................................................................................................. 9-1

9.1.3 External file name .............................................................................................................. 9-2

9.2 **FILE NAMING FACILITIES** ............................................................................................ 9-3

9.2.1 Printing .............................................................................................................................. 9-3

9.2.2 Unmixed FORTRAN Units ............................................................................................... 9-3

9.2.3 Cataloged File at OPEN Time ........................................................................................... 9-4

9.2.4 Cataloged File at ASSIGN Time ....................................................................................... 9-4

9.2.5 Mixed FORTRAN and COBOL ....................................................................................... 9-5

9.3 **FILE USE IN JOR** ........................................................................................................... 9-6

9.3.1 Types of IFN ....................................................................................................................... 9-6

9.3.1.1 Numeric ......................................................................................................................... 9-6

9.3.1.2 Unit Record Device IFN ............................................................................................... 9-6

9.3.1.3 Numeric Preceded by a Semicolon .............................................................................. 9-6

9.3.1.4 Eight Star IFN ............................................................................................................... 9-6

9.3.1.5 Others ......................................................................................................................... 9-6

9.4 **DATA MANAGEMENT OVERRIDING RULES** ................................................................. 9-7

9.4.1 Specification and Applicability of File Characteristics ...................................................... 9-7

9.4.2 Summary of Overriding Rules .......................................................................................... 9-7

9.4.3 Checks at OPEN Time ....................................................................................................... 9-8

9.4.3.1 Record Length .............................................................................................................. 9-8

9.4.3.2 Permitted File Organizations ....................................................................................... 9-8

9.5 **OPTIONAL FILES** ........................................................................................................... 9-9

9.5.1 DUMMY Parameter ........................................................................................................... 9-9

9.5.2 OPTIONAL Parameter ....................................................................................................... 9-9

9.5.3 I/O Restrictions ................................................................................................................ 9-9

9.6 **CLOSE WITH STATUS='DELETE'** .................................................................................. 9-10

9.7 **THE POOL JCL STATEMENT** ......................................................................................... 9-10

9.8 **MULTIVOLUME FILES** .................................................................................................. 9-10

9.9 **MULTIPLE FILE TAPE VOLUMES** ................................................................................ 9-11
Table of Contents

9.10 FILE CONCATENATION ................................................................. 9-11

9.11 FILE ACCESS SYSTEMS .............................................................. 9-12
9.11.1 UFAS .......................................................................................... 9-12
9.11.2 BFAS .......................................................................................... 9-12
9.11.3 Specifying the Access System .................................................. 9-12

9.12 CREATING FILES ................................................................. 9-13
9.12.1 PREALLOC Statement ............................................................... 9-13
9.12.2 ALLOCATE Statement ............................................................... 9-13

9.13 TERMINAL READ/WRITE ...................................................... 9-13

9.14 ERROR HANDLING ............................................................... 9-14

9.15 RETURN CODE ........................................................................... 9-14

10. Standard Record Formats ............................................................ 10-1

10.1 THE FOUR STANDARD RECORD FORMATS .......................... 10-1
10.1.1 System Standard Format (SSF) .................................................. 10-1
10.1.2 Standard Access Record Format (SARF) ................................. 10-1
10.1.3 American Standards Association Format (ASA) ..................... 10-1
10.1.4 Device Oriented Format (DOF) .................................................. 10-2

10.2 FORTRAN 77 MANIPULATIONS ON SSF FILES .................. 10-3
10.2.1 SSF Header Format ................................................................. 10-3
10.2.1.1 Record Type ........................................................................... 10-3
10.2.1.2 Header Type ........................................................................... 10-3
10.2.1.3 Truncation Value .................................................................... 10-3
10.2.1.4 Line Number ........................................................................... 10-3
10.2.1.5 Form Control ........................................................................... 10-3

10.2.2 Effects of Header Type ............................................................. 10-4
10.2.3 Creation and Updating by FORTRAN 77 ............................... 10-4
10.2.4 SSF and the Utilities ................................................................. 10-4
10.2.5 Reading SSF Files ................................................................. 10-5
10.2.6 Writing SSF Files ................................................................. 10-6

10.3 FORTRAN 77 MANIPULATIONS ON SARF FILES .............. 10-7
10.3.1 SARF and the Utilities ............................................................. 10-7
10.3.2 Reading SARF Files in FORTRAN 77 Programs ................. 10-8
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3.3</td>
<td>Writing SARF Files in FORTRAN 77 Programs</td>
<td>10-8</td>
</tr>
<tr>
<td>10.4</td>
<td>SSF AND SARF WITH THE OUTPUT WRITER</td>
<td>10-9</td>
</tr>
<tr>
<td>11.</td>
<td>Using Unit Record Files</td>
<td>11-1</td>
</tr>
<tr>
<td>11.1</td>
<td>PRINTING</td>
<td>11-1</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Using SYSOUT Files for Printing</td>
<td>11-1</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Printing Directly</td>
<td>11-3</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Form Control</td>
<td>11-3</td>
</tr>
<tr>
<td>11.2</td>
<td>CARDS</td>
<td>11-4</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Reading Cards Directly</td>
<td>11-4</td>
</tr>
<tr>
<td>11.2.2</td>
<td>Punching Cards</td>
<td>11-4</td>
</tr>
<tr>
<td>11.2.2.1</td>
<td>Using Sysout Files for Cards</td>
<td>11-5</td>
</tr>
<tr>
<td>11.2.2.2</td>
<td>Punching Cards Directly</td>
<td>11-5</td>
</tr>
<tr>
<td>11.3</td>
<td>PAPER TAPE</td>
<td>11-6</td>
</tr>
<tr>
<td>11.4</td>
<td>DISKETTES</td>
<td>11-7</td>
</tr>
<tr>
<td>11.5</td>
<td>PAUSE AND STOP LITERAL</td>
<td>11-7</td>
</tr>
</tbody>
</table>
12. Differences Between GCOS FORTRAN and GCOS FORTRAN 77

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 DIFFERENCES REMOVED BY THE PARAMETER &quot;LEVEL=GCOS1E&quot;</td>
<td>12-1</td>
</tr>
<tr>
<td>12.1.1 Differences on Reading End of File</td>
<td>12-1</td>
</tr>
<tr>
<td>12.1.2 Differences on Split Records</td>
<td>12-2</td>
</tr>
<tr>
<td>12.1.3 Differences on Argument Passing</td>
<td>12-2</td>
</tr>
<tr>
<td>12.1.4 Differences on Local Variables</td>
<td>12-2</td>
</tr>
<tr>
<td>12.2 DIFFERENCES NOT REMOVED BY THE PARAMETER &quot;LEVEL=GCOS1E&quot;</td>
<td>12-3</td>
</tr>
<tr>
<td>12.2.1 Differences on ENDFILE Statement</td>
<td>12-3</td>
</tr>
<tr>
<td>12.2.2 Differences on Secondary Entry Points</td>
<td>12-3</td>
</tr>
<tr>
<td>12.2.3 Differences in Service Routine Names</td>
<td>12-3</td>
</tr>
<tr>
<td>12.2.4 Differences in JCL ASSIGN</td>
<td>12-4</td>
</tr>
<tr>
<td>12.2.5 Blank Characters in Formatted Read Statement</td>
<td>12-4</td>
</tr>
<tr>
<td>12.2.5.1 Record Length</td>
<td>12-4</td>
</tr>
<tr>
<td>12.2.5.2 Blank Spaces Ignored</td>
<td>12-4</td>
</tr>
<tr>
<td>12.2.6 Exceptions on Execution</td>
<td>12-5</td>
</tr>
<tr>
<td>12.2.6.1 R3STACKOV</td>
<td>12-5</td>
</tr>
<tr>
<td>12.2.6.2 WRITE RIGHT Violation</td>
<td>12-5</td>
</tr>
</tbody>
</table>

13. Modular Programming and Protection

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 MODULAR PROGRAMMING</td>
<td>13-1</td>
</tr>
<tr>
<td>13.1.1 Encapsulation of Procedures into Modules</td>
<td>13-1</td>
</tr>
<tr>
<td>13.1.2 Independence Between Logical Modularity and Physical Support</td>
<td>13-4</td>
</tr>
<tr>
<td>13.1.3 Packaging Program Units after Indepandant Compilations</td>
<td>13-6</td>
</tr>
<tr>
<td>13.1.4 Static Testing of an Executable Program</td>
<td>13-8</td>
</tr>
<tr>
<td>13.1.4.1 Cross Reference Table of a Package Unit</td>
<td>13-8</td>
</tr>
<tr>
<td>13.1.4.2 Cross Reference Table of a Large Executable Program</td>
<td>13-8</td>
</tr>
<tr>
<td>13.1.4.3 Check of Procedure Calls</td>
<td>13-8</td>
</tr>
<tr>
<td>13.1.5 Packaging to Reduce the Execution Time and Number of Segments</td>
<td>13-9</td>
</tr>
<tr>
<td>13.2 PROTECTION</td>
<td>13-10</td>
</tr>
<tr>
<td>13.2.1 What is Protection</td>
<td>13-10</td>
</tr>
<tr>
<td>13.2.2 Requirements for Protection</td>
<td>13-10</td>
</tr>
<tr>
<td>13.2.3 Protection with DPS 7 FORTRAN 77</td>
<td>13-11</td>
</tr>
<tr>
<td>13.2.3.1 Early Error Signaling</td>
<td>13-11</td>
</tr>
<tr>
<td>13.2.3.2 Exception Mechanism</td>
<td>13-11</td>
</tr>
<tr>
<td>13.2.3.3 Run Time Package</td>
<td>13-12</td>
</tr>
<tr>
<td>13.2.3.4 SUBCK Parameter of the FOR77 JCL Statement</td>
<td>13-12</td>
</tr>
</tbody>
</table>
Illustrations

Figures

2-1 FOR77 JCL Statement Format ................................................................. 2-2
2-2 Action of FORTRAN Statement .............................................................. 2-22
3-1 LINKER JCL Statement Format ............................................................... 3-2
5-1 Hardware Representation of Real Number .............................................. 5-3
5-2 Hardware Representation of Double Precision Number ......................... 5-4
5-3 Hardware Representation of Quadruple Precision Number .................... 5-5
5-4 Hardware Representation of Floating Point Numbers .......................... 5-8
5-5 Format of 15 Digit Intermediate Result .............................................. 5-10
5-6 Format of 29 Digit Intermediate Result (con't) .................................... 5-10

Tables

2-1 Severity Value Set by the Compiler ......................................................... 2-17
6-1 Correspondence Between Data Types .................................................. 6-3
6-2 Storage Data ......................................................................................... 6-4
9-1 Applicability of File Characteristics ..................................................... 9-7
9-2 Permitted File Organizations ................................................................. 9-8
1. Introduction

The GCOS operating system provides an environment for FORTRAN 77; effective use of this environment is necessary for the optimization of FORTRAN 77 programs.

This section describes the principal features of GCOS that affect the preparation and execution of FORTRAN programs.

1.1 GCOS FUNCTIONS

Some of the tasks carried out by GCOS are as follows:

- Scheduling of jobs according to their priority
- Allocation of memory for programs and data areas
- Execution of programs and organization of data
- Input and output, including device allocation
- Maintenance of libraries
- Detection of errors

More information about GCOS is given in the System Overview manual.

1.2 JOB CONTROL LANGUAGE

Communication with the operating system is obtained by means of a Job Control Language (JCL) that describes input, output, and layout of files.

See the JCL Reference Manual for the syntax of JCL statements.
1.3 JOBS AND STEPS

A job is a unit of work for a user of the computer. It is identified by the JCL statement $JOB at the beginning and $ENDJOB at the end; these statements define what is known as a job enclosure.

The syntax of $JOB is found in the JCL Reference Manual.

A job must consist of at least one step, which represents the loading and execution of a program. A step can be identified by an extended JCL statement such as FOR77 or LINKER, or it can begin with STEP and end with ENDSTEP. These last two statements define a step enclosure. The JCL Reference Manual gives a full description of the statements STEP and ENDSTEP.

Most jobs will include some input, either of a source program to be compiled, of data cards to be read by the program, or both. Input cards are placed in an input enclosure, defined by $INPUT at the beginning and $ENDINPUT at the end. An input enclosure can occur anywhere within the job enclosure, but not inside the step enclosure. It can contain several program units (for example a main program and subroutines); a separate compilation is performed for each. If the input is FORTRAN source code, $INPUT may contain the parameter TYPE = DATASSF or TYPE = DATA (default value). Several input enclosures for data can be used by a FORTRAN step.
1.4SYSIN AND SYSOUT

The standard system input file is known as SYSIN. Records read into SYSIN will be deleted at the end of a job.

The standard system output file is called SYSOUT. Its contents are usually printed at the end of the job, using the system Output Writer, and are then deleted.

Section 11 describes how to set up a permanent input or output file if records are to be saved, and how to produce card punch output.

1.5GCOS7 COMMAND LANGUAGE

You can also use the FORTRAN 77 compiler from an interactive terminal using GCOS Command Language (GCL) commands. For details see the IOF Terminal User's Reference Manual (FOR77 command). Most of the examples given in the present manual are JCL-based.
FORTRAN 77
This section describes the use of the FORTRAN 77 compiler. The necessary JCL and output of the compiler are described in detail.

2. Compilation

2.1 THE FOR77 JCL STATEMENT

The extended JCL statement FOR77 executes the FORTRAN 77 compiler. The compiler generates a compile unit and listing. The compile unit can be stored in a temporary or permanent library. The linking and execution of the program must be requested by the user in subsequent job steps. The listing can also be stored in a temporary or permanent library or file.

Figure 2-1 shows the format of the FOR77 statement. Note that the parameters which are underlined are the default values assumed by the compiler when no alternative is chosen. For example, if the OBSERV parameter is not specified in the FOR77 statement, the NOBSERV parameter will be assumed by the compiler: Default values are therefore redundant in the FOR77 statement.
2.1.1 The F7C Abbreviation

The abbreviation F7C can be used instead of FOR77. Note that this abbreviation is used to prefix error messages in the Job Occurrence Report.

FOR77
F7C

INFILE = (sequential – input – file – description)

SOURCE = \( \begin{cases} \text{input – enclosure – name} \\ \text{member – name} \\ \text{(member – name[member – name]...)} \\ \text{(star – name[star – name]...)} \end{cases} \)

CULIB = \{ \text{output – library – description} \}

NDEBUG \ NDEBUGMD \ NROUND \ NSUBCK \ OBJ

DSEGMAX = \text{nnnnK} \ PSEGMAX = \text{nnnnK} \ INIT = \text{hh} \ CODE \ OBJ

LEVEL = \text{GCOS7} \ [\text{GCOSTE}] \ [\text{LIST}] \ [\text{NMAP}] \ [\text{NOBSERV}] \ [\text{OBJECT}] \ OPTIMIZE = \text{0} \ 1 \ 2 \ 3 \ 4

PACKAGE = \{ \text{NO} \ \text{OPEN} \ \text{CLOSE} \}

PRTFILE = (print – file – description)

PRTLIB = \{ \text{TEMP} \ \text{NXREF} \ \text{PI} \ \text{NPI} \}

SIZEOPT = (size – parameters)

STEPOPT = (step – parameters)

\( \begin{cases} \text{WARN} \\ \text{SILENT} \\ \text{NADP} \ \text{ADP} \end{cases} \)

\( \begin{cases} \text{XREF} \\ \text{PI} \ \text{NPI} \end{cases} \)

Figure 2-1. FOR77 JCL Statement Format
2.1.2 Where to Place the FOR77 Statement

As the FOR77 statement is an extended JCL statement it must be placed outside a step enclosure and not within.

The following example illustrates the use of this statement.

```
$JOB...
LIBALLOC CU, (CU.LIB, SIZE=5), MEMBERS=50;
FOR77 SOURCE = *PROG1,CULIB=CU.LIB;
$INPUT PROG1;
    PROGRAM PROG1
    INTEGER I
    ;
$ENDINPUT;
$ENDJOB;
```

The LIBALLOC CU statement is used to create a library CU.LIB with a size of 5 cylinders. Normally the library already exists and this utility need not be used. The compiler reads the source program from the input enclosure PROG1 (via SYSIN) and stores the compile unit in the compile unit library CU.LIB.
2.1.3 Parameters of FOR77 Compilation

The following sub-sections describe the parameters which may be used in the FOR77 statement. Note that the following symbolic names used in Figure 2-1 refer to standard parameter groups and are described in the JCL Reference Manual.

sequential-input-file-description
input-library-description
output-library-description
print-file-description
print-library-description

See the JCL Reference Manual for all information concerning these standard parameter groups.

2.1.3.1 SOURCE, INFILE, INLIB and INLIBn Parameters

These parameters specify the name and location of the program or programs to be compiled. A series of programs can be compiled during a single execution of the compiler. See "Serial Compilation" later in this section.

Either the SOURCE or INFILE parameter must be specified in a FOR77 statement. All of the other parameters are optional. SOURCE and INFILE can 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:

FOR77 SOURCE = *input-enclosure-name;

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

If the source program is held in a library, the name of the library and the member are both specified in the FOR77 statement as follows:

FOR77 SOURCE = member-name
    INLIB = (input-library-description) ;

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

LIB SL INLIB1 = (input-library-description)
    [ INLIB2 = (input-library-description)
      [ INLIB3 = (input-library-description)]];
FOR77 SOURCE = member-name ;
The LIB JCL statement defines a "search path" for the compiler. The compiler searches 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 is compiled and any other of the same name is 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 FOR77 JCL statement. 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)
;
FOR77 SOURCE = member name

INLIB1
INLIB2
INLIB3
```

The three methods of specifying a member name and library, described above, can 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.

**Example:**

```
FOR77 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:

```
FOR77 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 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)
;
FOR77 SOURCE = member name

INLIB1
INLIB2
INLIB3
```

Note that if the library to be used is specified in the FOR77 statement (i.e. no library search is carried out), then 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 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.

2.1.3.2 CULIB Parameter

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

If a library is specified, it must have been previously allocated by, for example, the LIBALLOC utility, 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 is 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 symbolic name of the PROGRAM, SUBROUTINE, FUNCTION or BLOCKDATA statement. If these are absent the member name is FORTRANMAIN for a main program or FORTRANBLK0, FORTRANBLK1, etc., for BLOCKDATA subprograms.

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 LIB CU JCL statement is required to define the search path.

2.1.3.3 DEBUG and NDEBUG Parameters

The DEBUG parameter requests the compiler to build a table of all the symbols and labels of the program unit, indicating their type and structure together with the generated segment addresses. This table is stored in the compile unit. After linking, the program can be executed under control of the Program Checkout Facility. See Section 4.

NDEBUG is the default parameter assumed if DEBUG is not specified. If DEBUG is not specified, the Program Checkout Facility can be used only with effective addresses.
2.1.3.4 DEBUGMD and NDEBUGMD Parameters

DEBUGMD causes compilation of FORTRAN lines containing asterisks in both column 1 and column 2 of the card. Such statements are used to insert test values or print out intermediate results. When DEBUGMD is omitted the lines will be treated as comments. See Sub-section 4-8.

NDEBUGMD causes the compiler to ignore lines containing asterisks in both column 1 and 2 of the card and to treat these as comments. This is the default parameter in the absence of DEBUGMD.

2.1.3.5 DSEGMAX Parameter

This parameter defines the maximum size of segments for local data (refer to Section VII on segment generation), in units of 1024 bytes.

If DSEGMAX is not used, only one internal static segment is produced with a size large enough to contain the corresponding data and any alignment gaps.

If DSEGMAX is used, data is stored on as many segments as necessary in order that the segment size does not exceed the DSEGMAX value. However, for an array, a block of EQUIVALENCEd data is not split if it is larger than the DSEGMAX value. In these cases, a warning is issued and the DSEGMAX value doubled. This process is repeated until an adequate DSEGMAX value is reached.

DSEGMAX has no influence on a COMMON block, nor on all COMMON blocks of a packaged unit not declared in an EXTCOMMON statement. A segment is allocated for all COMMON blocks.

DSEGMAX also provides the limit for the size of supplementary segments produced when the DEBUG parameter is specified. However, there is the overriding restriction of a maximum segment size of 64K bytes imposed by the limits of the Program Checkout Facility.

DSEGMAX can be used to improve Virtual Memory Management performance in the case of very large program units. The DSEGMAX option will cause the compiler to generate supplementary instructions. These instructions load registers with segment addresses each time that data of these segments are accessed. This results in a slower execution speed. It is therefore helpful to use DSEGMAX only when the data segments exceed the size of the tracks on the system disk. It is not recommended practice to use the package feature to improve execution speed and DSEGMAX to reduce segment size, as DSEGMAX slows down the execution speed. It is better to reduce the number of subroutines contained in the package.

For the DPS7, the maximum size of a data segment is 4096 K. If a segment larger than 4096 K is produced by the compiler it will be rejected by the linker. If the compiler and linker produce a segment which exceeds the size of available physical memory, then it is rejected by the loader.
2.1.3.6  PSEGMAX Parameter

The PSEGMAX parameter defines the maximum size of code segments to be produced during compilation, in units of 1024 bytes. If PSEGMAX is not specified, the linkage section, constant section and code are produced in the one single segment, this being as large as necessary. If PSEGMAX is specified, the linkage and constant section are in one segment while the code is stored in as many segments as necessary, so that the segment sizes do not exceed, by too much, the PSEGMAX value.

In the case of very large program units, PSEGMAX may be used to improve the performance of Virtual Memory Management. Note, however, that the PSEGMAX option will result in supplementary instructions being generated by the compiler, which will result in a slower execution speed.

These instructions are mainly indirect branches each time there is a transfer from one segment to another. It is suggested that PSEGMAX be used only when the code segments exceed the track size of the system disk.

In order to avoid significant performance degradation, the compiler does not cut the source code at any point. If the program unit is a PACKAGE unit, the change for another code segment is made only after an END statement following the point where the PSEGMAX value has been exceeded. If the program unit is not a PACKAGE unit, the change of code segment is made only after a statement which is followed by a DO statement and where the DO loop satisfies the following conditions:

- The DO loop is not contained in any other DO loop (cuts are never made within DO loops).
- The value of PSEGMAX for the current segment has been exceeded at the beginning of the DO loop.

There are two ways of using PSEGMAX to reduce the number of segments:

- Set PSEGMAX to a very high value such as 4000000. In this case the linkage and constant section are produced in a separate segment to the code and no supplementary code is generated. In doing this, the execution speed will be kept to a maximum, except for the overhead of Virtual Memory Management.

- Set PSEGMAX to a value less than the code size. In this case the code is split into several segments and extra branch instructions are generated for transfer between segments. In general, a suitable value for PSEGMAX is to set it to a little less than the track size of the system disk, so that the sizes of the code segments do not exceed this track size. You may obtain the sizes of the segments by specifying the MAP parameter and looking at the line location map.
### 2.1.3.7 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 compilation time is greatly shortened. The decrease in compilation time may be by a ratio of from 6 to 12, depending on the other compilation parameters. 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 to list a program compilation without performing a program execution.

If NOBJ is specified the following parameters have no effect: MAP (NMAP is assumed), PSEGMAX, DSEGMAX.

### 2.1.3.8 LEVEL Parameter

The LEVEL parameter is used to specify how the source program should be interpreted.

When LEVEL = GCOS1E, the interpretation is close to the one used by the GCOS64 1E release of the FORTRAN compiler.

When LEVEL = SIRIS8, the interpretation is close to that of the SIRIS8 FORTRAN compiler.

When LEVEL = GCOS7, the interpretation is that of the ANSI 77 standard with extensions and precisions, provided by DPS 7 FORTRAN 77. GCOS7 is the reference value, and its effect is as specified in the DPS 7 FORTRAN 77 Reference Manual.

The only differences between these LEVEL values are as follows:

**LEVEL = GCOS1E:**
- All variables and arrays are treated as if they were specified in a SAVE statement.
- All variables and arrays are allocated in segments in the order of their declaration, without gaps for alignment (except when specified by EQUIVALENCE statements).
- Each file, if opened, is treated as if opened by an OPEN statement with the SPANNING = YES specifier. If spanning records are written, they are the same as those produced by GCOS1E FORTRAN. If spanning records are read, those produced by both DPS7 FORTRAN and GCOS1E FORTRAN are accepted.
- The READ statements recognize the record produced by the GCOS1E FORTRAN when executing an ENDFILE statement.
- The ROUND parameter is assumed.
- The execution speed is less than the one achieved with LEVEL = GCOS7 parameter.
LEVEL = SIRIS8:

- All variables and arrays are treated as if they were specified in a SAVE statement.

- All variables and arrays are allocated in segments, in the order of their declarations, without gaps for alignment.

- The NROUND parameter is assumed.

- The comma is considered as a separator in formatted input records, except for format A.

- Internal READ/WRITE can go beyond the last record (as for SIRIS8 ENCODE/DECODE).

- All files are assumed to be opened by an OPEN statement with the PAD = YES specifier.

- Format specifiers without length specification are supported, with the following restrictions:
  - in input, the allowed separators are "," and " "(blank).
  - in output, if an edit descriptor G.d leads to a F.d edition, then four blanks are added at the end of the edition instead of only one blank, as is the case for SIRIS8 FORTRAN

- Adjustable formats (with = instead of N as SIRIS8), except that a variable format may not contain a - = P edit descriptor.

- The execution speed is less than that achieved by using the LEVEL = GCOS7 parameter.

2.1.3.9 LIST and NLIST Parameters

NLIST specifies that the source program listing is not produced. However, lines associated with the production of error messages are produced. LIST specifies that the complete program listing is produced and is the default option.

2.1.3.10 XREF and NXREF Parameters

The XREF parameter produces a cross-reference listing in alphabetic order. NXREF means that no such cross-reference listing is required and is the default parameter.
2.1.3.11 SILENT and NSILENT Parameters

The SILENT and NSILENT parameters have an effect when running in an interactive environment under IOF.

The SILENT parameter specifies that nothing is printed at the terminal, except the first and last FORTRAN 77 banners, the source identification, and the summary. The errors with the corresponding source lines are printed only in the listing.

The NSILENT parameter is the default value and specifies that errors are printed at the terminal with the corresponding source lines. Only the first 20 erroneous source lines are printed at the terminal.

2.1.3.12 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.

2.1.3.13 OBSERV and NOBSERV Parameters

The OBSERV parameter specifies that all observation diagnostic messages be included in the program listing. The messages begin with one asterisk. If errors of this type are detected, the JCL status value is set to SEV1 at the end of compilation, provided that no errors of a higher severity are detected. (See "JCL Status", below).

The NOBSERV parameter suppresses the printing of all observation diagnostic messages in the program listing and is the default parameter if OBSERV is not specified. However, the number of such messages is printed in the compilation summary page and in the Job Occurrence Report.

Note that observation diagnostic messages correspond to features that conform to the DPS 7 FORTRAN 77 Reference Manual but that may result from error or omission in programming and that may usefully be pointed out during compilation.

A typical example is the implicit conversion in an expression, resulting from the mistyping of a symbol, leading to an implicit type different from that of the expected symbol.
2.1.3.14 WARN and NWARN Parameters

The WARN parameter produces all warning diagnostic messages in the program listing. These messages begin with two asterisks. If errors of this type are detected by the compiler and no serious or fatal errors are found, the JCL status value is set to SEV2 at the end of the compilation. See "JCL Status" below. WARN is the default parameter if NWARN is not specified.

The NWARN parameter suppresses all warning diagnostic messages in the program listing. However, if errors of this type are detected by the compiler, their total is printed in the compilation summary and the Job Occurrence Report.

Warning diagnostic messages correspond to features that do not conform to the DPS 7 FORTRAN 77 Reference Manual, but for which the compiler has performed some repairing. Note that warning messages can be important because the feature is not the one usually expected, or because some of these messages will become fatal in the next release.

2.1.3.15 PRTFILE Parameter

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

```
$JOB...
FOR77
SOURCE = *FORSOURCE,
CULIB = CU.LIB,
MAP,
XREF,
PRTFILE = (COBFILE, DEVCLASS = MT/T9/D1600, MEDIA = ATAPE);
```

In this case, only the Job Occurrence Report will be printed at the end of job execution. (Note that the Job Occurrence Report is unaffected by the PRTFILE parameter.)

If the PRTFILE parameter is used, the compiler adds the program 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).

When serial compilation occurs, and PRTFILE parameter is used, all listings are stored in a single file.
2.1.3.16 PRTLIB Parameter

This parameter is similar to PRTFILE except that the listing is stored in a member of the library specified in the PRTLIB parameter. If several programs are compiled in series when the PRTLIB parameter is used, the listing for each program is stored in a separate library member. This is given the symbolic name specified in the PROGRAM, SUBROUTINE, FUNCTION or BLOCKDATA statement of the source program suffixed by " L", or the symbolic name FORTRANMAIN suffixed by " L" in the case of a main program without a PROGRAM statement. It replaces any member of the same name (For BLOCKDATA, FORTRANMAIN will be replaced by FORTRANBLK0, FORTRANBLK1...).

2.1.3.17 SIZEOPT Parameter

The SIZEOPT parameter group is used to specify one or more of the parameters included in the SIZE JCL statement (see the JCL Reference Manual).

2.1.3.18 SUBCK and NSUBCK Parameters

The SUBCK parameter causes the compiler to perform or generate extra checks. If the check is possible at compilation, it is performed and can lead to a compiler message. If the check is possible only at execution, it is generated by the compiler and leads to a possible run time message. The checks performed are the following:

- Check that an array subscript is within the array bounds for the corresponding dimension. This check is not made for the last dimension of an assumed size array.

- Check that a subscript expression is within the corresponding character variable or array element. This check is made even if it is an assumed size character variable or array element.

- Check that the label value of the integer variable of an assigned GOTO is one of the labels on the list, if any.

- Check, but only at compile time, that the subscript values specified by a DATA statement are within the bounds of the corresponding array. Note that the SUBCK parameter results in extra code being generated by the compiler, which can slow down execution speed. The NSUBCK parameter causes no checking and is the default option in the absence of the SUBCK parameter.
2.1.3.19 ROUND and NROUND Parameters

The NROUND parameter specifies that all floating point operations on real and double precision data should be performed with truncation of the extra digits of the hardware intermediate results (Refer to Section 5). The ROUND parameter specifies that all floating point operations on real and double precision data should be performed with use of the hardware rounding of intermediate results. In simple cases, rounding is more accurate than truncation. But in complex cases, where the least significant digits result from approximations of previous computations, rounding propagates noise rather than accuracy. This is particularly sensitive on convergence algorithms, where truncation allows monotonic increasing, whereas rounding may produce oscillation or divergence. Execution speed is slightly faster with the NROUND parameter, compared with the ROUND parameter. The NROUND parameter is the default in the absence of the ROUND parameter (if LEVEL=GCOS7 or SIRIS8).

2.1.3.20 INIT Parameter

The INIT parameter is used to preinitialize certain variables in a FORTRAN 77 program. It sets each byte of the variable to a value specified by two hexadecimal numbers (INIT=hh). The variables are affected as follows:

- For variables that are not initialized by a DATA instruction but appear in a SAVE instruction, all the bytes are set to hh.
- For variables that are not initialized by a DATA instruction but appear in the COMMON statement of a BLOCK DATA section, all the bytes are set to hh.
- For variables that are only partially initialized by a DATA instruction, all the remaining (uninitialized) bytes of the variable are set to hh.

Example:

The parameter INIT = "OF" is included in the compilation statement of the program shown in the box below Result:is preinitialized to the hexadecimal value "OF0FOFOF"

```fortran
PROGRAM P
REAL |
SAVE |
. |
. |
END
```
2.1.3.21 ADP and NADP Parameters

The ADP (Automatic Double Precision) parameter specifies that every real (or complex) variable, array or external function must be treated as if it had been declared in a DOUBLE PRECISION (COMPLEX DOUBLE PRECISION) statement. Moreover, the specific names of intrinsic functions that return a real (complex) result are replaced by the specific names of intrinsic functions that return a double precision (complex double precision) result.

NADP (Non Automatic Double Precision) is the default value.

2.1.3.22 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 DPS7000 and 64/DPS
- Class C: DPS7/X0,X17,X27
- Class D: DPS7/1XX7.

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

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

When the FORTRAN compiler runs under GCOS7-V3A, the default value is OBJA.

When the FORTRAN compiler runs under GCOS7-V3B, the default value is OBJCD.

A program compiled with CODE=OBJA and executed on a class D computer may lead to a loss of precision on floating point results and will be rejected by the system.

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 is used to get information on the compatibility class of a compiled unit. It must be interpreted as follows:

<table>
<thead>
<tr>
<th>CU class</th>
<th>A-C compatible</th>
<th>C-D compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>O or none</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

2.1.3.23 PI and NPI Parameters

The PI parameter specifies that the rule of integrity of parentheses must be applied (see Reference Manual, section 6.6.3 ). NPI is the default value.
2.1.3.24 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 14.

OPTIMIZE=4  Global optimization. To reduce execution time, some loops are unrolled. For more information, see the optimization description in section 14.

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 can increase the amount of generated code.

2.1.3.25 PACKAGE Parameter

This parameter specifies that the source file requires automatic packaging. Automatic packaging can create either an "open package" or a "closed package". The default is no automatic packaging.

For more information on automatic packaging, see section 19 of the Fortran 77 Reference Manual.
2.2 JCL STATUS

At the end of the compilation, subsequent job processing can be determined by testing, with the JUMP JCL statement, a severity value set by the compiler or by the system. Severity values are printed in the Job Occurrence Report. The possible values are shown in Table 2-1.

<table>
<thead>
<tr>
<th>Severity Value</th>
<th>Status</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEV0</td>
<td>0</td>
<td></td>
<td>No error.</td>
</tr>
<tr>
<td>SEV1</td>
<td>100</td>
<td>*</td>
<td>Observation: correct feature, but may be due to programming error.</td>
</tr>
<tr>
<td>SEV2</td>
<td>1000</td>
<td>*</td>
<td>Warning: non conformity with the Reference Manual, but repaired by the compiler.</td>
</tr>
<tr>
<td>SEV3</td>
<td>10000</td>
<td>***</td>
<td>Fatal error: non conformity with the Reference Manual, does not allow production of the compile unit, but does not stop analysis of the program.</td>
</tr>
<tr>
<td>SEV4</td>
<td>20000</td>
<td>****</td>
<td>Untraceable error: non conformity with the Reference Manual in structure of the program, or limit of the compiler reached, or system error, or unsuccessful check in the compiler. Analysis of the program is stopped. A source listing is generally given.</td>
</tr>
<tr>
<td>SEV5</td>
<td>40000</td>
<td></td>
<td>Compilation killed by the operator or the user (TJ command).</td>
</tr>
<tr>
<td>SEV6</td>
<td>60000</td>
<td></td>
<td>Abort requested by system (exception in the compiler) or system crash.</td>
</tr>
</tbody>
</table>

The following example shows how the severity value can be tested to decide whether to link a program which has just been compiled:

```
$JOB
FOR77
SOURCE = *COMPST
JUMP ABNOR, SEV,GT,2;
LINKER COMPST,COMPFILE=*LKF77;
ABNOR:.. ..
```
2.3 SERIAL COMPILATION

The compiler can compile a series of programs during a single execution. Two levels of "seriality" are available.

- At a lower level of seriality there can be several program units or several package units in a single member, or in a file. Each program unit or package unit will be compiled in its turn.

- The upper level may be at the level of the SOURCE parameter of the FOR77 JCL statement. This parameter may specify more than one member. In that case, all program units or package units contained in the first specified member are compiled, then all those contained in the second specified member, and so on.

In the case of a package unit spanning several program units, all of the units will be compiled. The order in which the members are specified in the SOURCE parameter must be consistent with their content. The member containing the begin of the package unit must be specified first and the member containing the end of the package unit must be specified last.

2.4 INTERACTIVE COMPILATION

The compiler can be executed from an IOF terminal. The normal FOR77 JCL statement can be used for this purpose. Serial compilation can be specified in the normal way. The compiler executes as in batch mode with the following exceptions:

The printer output is stored in the TEMP source library rather than in a standard SYSOUT subfile, if neither the PRTFILE nor the PRTLIB parameter is used. If the user wishes the printer output to be actually printed, a WRITER JCL statement must be used. Before it is printed, the output can be examined using the SCAN or LIBMAINT utility.

Diagnostic messages are output on the IOF terminal as well as in the program listing. If a given message is output more than once for a particular compilation, the explanatory text part of the message is not repeated on the terminal for the second and subsequent occurrences of the message. The compiler will suppress the output of error messages when 20 lines with diagnostics have been output on the terminal. All messages sent to the Job Occurrence Report by the compiler are also sent to the terminal, including the compilation summary.

If the Break key on the terminal is pressed, the current compilation or series of compilations is terminated immediately.

NOTE: An input enclosure cannot be entered on a terminal, so the source program must be held in a normal library member or in a sequential file. The format of the source lines must be as described in the following section, for library members.
2.5 BATCH COMPILATION: FORTRAN SOURCE FORMAT

The format of the source of a FORTRAN executable program vary according to the following cases.

2.5.1 FORTRAN Source on Library Member (or a Sequential File)

In this case each program unit or the whole executable program is on one library member with type FOR, or DATA.

The compiler assumes that each source line has the following format:

- col. 1 to 5 : statement label
- col. 6 : continuation
- col. 7 to 25 : FORTRAN statement.

This format is especially suited for a member created and updated interactively by the Text Editor EDIT under LIBMAINT, as lines may be less or more than 72 characters by the effect of substitute commands.

Note that the ANSI 77 standard allows only lines of 72 characters.

If you want to restrict your program to this format, you may use a record length specification in the Z command of the Text Editor.

For example, if you want to write your buffer into a member SB1, use:

```
Z(FOR,72) SB1
```

Note that if you do a Z without a type specification, for example:

```
Z SB1
```

the type DATA is assumed. But it is better to specify the type FOR at creation:

```
Z(FOR) SB1
```

This allows you a better administration of your library.

Similarly you are advised to create the library member for the JCL with the type JCL. For example:

```
Z(JCL) JC1
```
2.5.2 FORTRAN Source on Cards

The compiler assumes that each source line of an input enclosure has the following format:

- col. 1 to 5: statement label
- col. 6: continuation
- col. 7 to 72: FORTRAN statement
- col. 73 to 80: ignored, may be used for card numbering.

The following is an example of the JCL for the compilation of FORTRAN source on cards.

```plaintext
$JOB SPECIAL,USER = X,REPEAT;
  FOR77 SOURCE = *INPENC,OBSERV;
  LINKER FORTRANMAIN;
$STEP FORTRANMAIN,TEMP;
$ENDSTEP;
$INPUT INPENC,TYPE = DATASSF;
  I = 1
  STOP
  END
$ENDINPUT;
$ENDJOB;
```

2.5.3 Input Enclosure in a Library Member

If an input enclosure has been copied from cards, and stored in a library member, possibly with the JCL, care must be taken that the FORTRAN source of this input enclosure remains in card format, as specified above.

This format is sometimes inadvertently changed when the stored input enclosure is updated using LIBMAINT. If this occurs, the compiler will take the following action:

If a source line contains less than 72 characters, the line will be padded with blanks on the right up to 72 characters. If the line contains more than 72 characters, only the first 72 characters will be processed by the compiler.

This may result in errors, mainly for character constant or hollerith constants continued on the following card.
Compilation

To avoid this, create the library member, from the cards, by use of the LIBMAINT MOVE command, in the following manner:

MOVE COMFILE: subfile-name, TYPE = FOR ;

If TYPE = FOR is not specified when the library member is created, the format can be corrected in the following way. Assuming that the member is stored with TYPE = DATASSF, one could use the following LIBMAINT command:

MOVE subfile-name, TYPE = FOR, REPLACE, FORMAT=(0,0,1,72) ;

This sequence of commands causes columns 72 to 80 of each line to be suppressed. Therefore, card numbers that are interpreted as FORTRAN statements or blanks that are taken to be part of constant strings (when such strings are longer than two source lines) are suppressed.
2.6 SUMMARY OF COMPILATION

Figure 2-2. Action of FORTRAN Statement
3. Linking

LINKER is a utility that builds an executable load module from a set of compile units. These compile units can result from the compilation of programs written in different source languages. LINKER resolves all references between compile units and sets up links to FORTRAN 77 run-time package procedures and system procedures, which are resolved at run-time.

The information concerning LINKER given below is confined to the following topics:

- LINKER JCL;
- serial linkage;
- interactive use of LINKER;
- LINKER commands of interest to the FORTRAN 77 programmer;
- listings of interest to the FORTRAN 77 programmer.

For a more detailed description of the LINKER listings and commands see the LINKER manual.

3.1 SEGMENT NUMBERS

The system recognizes two forms of segment number during compilation, linking, program loading, and execution. To avoid confusion in this and later sections, these forms of segment number are explained below:

- Internal Segment Number. This is the segment number generated by the FOR77 compiler to identify the segments within a compile unit. It is this number which appears to the left of the colon in the data map, cross-reference, and procedure map listings produced by the compiler. Internal segment numbers are also included in the segment lists produced by the compiler and by LINKER.

- LINKER Segment Number. This is the segment number 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.
3.2 LINKER JCL STATEMENT

The LINKER utility is called by the extended JCL statement LINKER. Figure 3-1 shows the format of the LINKER statement.

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

Figure 3-1. 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 PROG1 LM
           ENTRY=PROG1
           OUTLIB=LM.LIB;
$ENDJOB;
```

The LIB CU JCL statement 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 PROG1 (specified in ENTRY=PROG1). 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 PROG1 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 3-1 refer to standard parameter groups that are described in the JCL Reference Manual:
3.2.1 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.

It is not possible to use the same name in this way when several source programs and compile units are to be linked into a single load module. However, it is advisable to adopt a systematic convention for program naming. For example:

- Load module INV comprises compile units INV-A, INV-B and INV-C which were compiled from source programs INV-A, INV-B and INV-C respectively.

- Load module UPDATE comprises compile units MAIN-UPDATE and ADMIN-UPDATE which were compiled from source programs MAIN-UPDATE and ADMIN-UPDATE respectively.

An asterisk (*) can be specified instead of output-module-name. This indicates that a series of load modules are to be linked during a single execution of LINKER. See "Serial Linkage", below.
3.2.2 INLIB Parameter

This parameter is used to modify the search path used by LINKER. The input library specified in this parameter is 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 is:

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 is:

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 is:

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 is:

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 is ignored if the INLIB parameter is also used.
3.2.3 OUTLIB Parameter

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

If a library is specified, it must be 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 is 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. When 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 a "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.

3.2.4 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, FILE, INCLUDE, STACK3, SEGTAB1, EXTR, FETCH, GATE, LIST, LINKTPE, MSEGAT, PLACE, REPLACE, TASK and VACSEG. The first five of these are described briefly in Section 3.5 and are of special interest to the FORTRAN 77 programmer. For a full description of all commands, see the LINKER manual. Commands must be separated by one or more spaces or by a comma and zero or more spaces. The final command must 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. See "Serial Linkage", below.
3.2.5 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 is 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.

Entry-name must be the value written in the PROGRAM, SUBROUTINE or FUNCTION statement.

When the ENTRY parameter is used the COMMAND and COMFILE parameters must be omitted.

3.2.6 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 is not printed automatically. Printing can be requested later by using a WRITER JCL statement. Only the Job Occurrence Report is printed at the end of job execution.

When serial linkage is requested (see "Serial Linkage" below) and the PRTFILE parameter is used, all listings are stored in a single file.

3.2.7 PRTLIB Parameter

This parameter is similar to PRTFILE except that the listing is 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 is stored in a separate library member. Each library member is given a name comprising the load-module-name suffixed by "_K". It replaces any member of the same name.
3.2.8 STEPOPT Parameter

The STEPOPT parameter can be used to specify one or more of the parameters included in the STEP JCL statement (see the JCL Reference Manual). However, the following cannot be included in the STEPOPT parameter for LINKER:

- load-module-name
- TEMP, SYS, or input-library-description
- the ALL option of the DUMP parameter
- the OPTIONS parameter
3.3 SERIAL LINKAGE

LINKER links a series of load modules during a single execution. In order to do this, an asterisk (*) is specified in the LINKER statement instead of load-module-name. The only other parameter that is permitted in such a LINKER statement is either COMMAND or COMFILE. In this case one of them is mandatory.

The COMMAND or COMFILE parameter is used to name each load module to be linked and to specify the commands applicable to each load module. For each load module to be linked, the load-module-name must be specified, followed by any associated commands. Each group of load-module-name and commands, including the last, must be followed by a semi-colon (;).

Example:

```
LINKER *
  COMMAND= LOAD-MODULE-1, ENTRY=ALPHA;
  LOAD-MODULE-2;
  LOAD-MODULE-3, ENTRY=BETA;'

LINKER *
  COMFILE=*CMD;
$INPUT CMD;
  LOAD-MODULE-1, ENTRY=ALPHA;
  LOAD-MODULE-2;
  LOAD-MODULE-3, ENTRY=BETA;
$ENDINPUT;
```

Note that the asterisk (*) specified in the LINKER statement does not have the same meaning as that specified in the FOR77 statement.
3.4 INTERACTIVE OPERATION

LINKER can be executed from an IOF terminal. The normal LINKER JCL statement is used. LINKER operates as in batch mode, except that the LINKER listing is not directed to the SYSOUT file. The listing, unless otherwise specified, is stored in the TEMP source library and must be printed via the WRITER JCL statement. The print subfile is stored in TEMP with the name lmn_K, where lmn is the load module name. The following example illustrates the use of LINKER at an IOF terminal.

S:   LK MYPROG;
     >>>14:31 LINKER 60:20
     WORKING ON:MYPROG
     LKOO.(60.20) SUMMARY FOR MYPROG   NO ERROR DETECTED
     OUTPUT MODULE PRODUCED
     <<<14:33
S:   WRITER (TEMP, SUBFILE=MYPROG_K);
3.5 LINKER COMMANDS

3.5.1 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 is used instead.

3.5.2 INCLUDE Command

The format of the INCLUDE command is as follows:

\[
\text{INCLUDE} = \begin{cases} \text{INLIB} \\ \text{INLIB1} \\ \text{INLIB2} \\ \text{INLIB3} \\ \text{INLIB4} \end{cases}
\]

The INCLUDE command incorporates compile units referred to in the FORTRAN 77 CALL statement. This form of the statement does not specify a program name at compilation time, so the LINKER cannot automatically incorporate the required compile unit into the load module. This is done by the programmer by using the INCLUDE command, which names all the compile units which may possibly be named in the data item referenced by CALL.

If INLIB is specified in the INCLUDE command, the contents of the library specified in the INLIB parameter of the LINKER JCL statement are included. If INLIBn is specified in the INCLUDE command, the contents of the first, second, third, or fourth library specified in the preceding INLIB CU JCL statement are included, depending upon the value of n.

There can be several INCLUDE commands for one load module. A list of included compile units are printed by LINKER.
3.5.3 STACK3 Command

The format of the STACK3 command, as used for an executable FORTRAN 77 program, is as follows:

\[
\text{STACK3} = \begin{cases} \text{INITSIZE} = m[K] \\ \text{MAXSIZE} = n[K] \\ \text{PAGING} = \begin{cases} \text{YES} \\ \text{NO} \end{cases} \end{cases}
\]

**INITSIZE** specifies the size of the initial page of the stack (i.e. storage for non SAVE variables). The units are taken as bytes unless the suffix K is present, in which case the units are 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:

\[
\text{INITSIZE}=2K, \quad \text{MAXSIZE}=16K, \quad \text{PAGING}=\text{NO}.
\]

3.5.4 SEGTAB1 Command

The format of the SEGTAB1 command, as used for an executable FORTRAN 77 program, is as follows:

\[
\text{SEGTAB1} = (\text{VSEG}=1)b
\]

The command is required only when an executable program reads or writes a file containing split records, when the maximum whole length of the records exceeds 64K bytes.
### 3.5.5 FILE Command

The format of the FILE command, as used for an executable FORTRAN 77 program, is as follows:

\[
\text{FILE} = \begin{cases} 
\text{FILEORG} = \begin{cases} 
\text{SEQ} \\
\text{DIRECT} \\
\text{RELATIVE}
\end{cases} \\
\text{NBUF} = \begin{cases} 
1 \\
2 \\
\vdots
\end{cases} \\
\text{NUMBER} = \begin{cases} 
n \\
1
\end{cases}
\end{cases}
\]

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

### 3.6 PRINTER 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 LINKER JCL statement 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 produced only 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.
3.6.1 Segment List

The segment list, contains an entry for each segment in the load module (including global segments but excluding code 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 segmenting a FORTRAN 77 program (DSEGMAX option) and when estimating working set requirements for program execution.

The headings and information in the segment list that are of use to the FORTRAN 77 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 FORTRAN summary page. Segments that are generated directly from user source code have a name of the form cun.isn where cun is the compile unit name and isn is the internal segment number.

TYPE This indicates that the segment contains code (C..), data (.D.) or linkage information (.L). Combinations of these types are also possible.

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.
3.6.2 Linkage Report

The first line of the linkage report contains either "ERRORS DETECTED" or "NO ERRORS DETECTED". If errors are not detected, the linkage report ends immediately after printing the line "OUTPUT MODULE PRODUCED ON LIBRARY library-name". However, if errors are detected, a summary of errors is 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 module is produced and the linkage report ends 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.

3.6.3 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" can have a value of 2, 3 or 4. (Severity 1 corresponds to a WARNING). Severity 4 is fatal and no load module is output. The total number of error messages of each severity is given in the linkage report.
4. Execution and Debugging

This section describes how executable programs can be executed in batch and interactive mode.

The debugging facilities of DPS7 FORTRAN 77 are also described.

4.1 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, (MYLIBRARY,DVC=MS/M452,MD=K104)

, CPTIME = 10000
, LINES = 20000
, DEBUG = (ALIBRARY,DVC=MS/M452,MD=K104, SUBFILE=DEBPROG)
, OPTIONS = 'CASE1'
, REPEAT;
The only mandatory parameters are MYPROG and its library description. The 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 100 minutes.

**LINES** limits the number of records written on the SYSOUT file, i.e. 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 the intrinsic functions ITEST and KTEST.

**REPEAT** specifies that the step is to be restarted after a system crash.
4.2 EXECUTION IN INTERACTIVE MODE

A typical sequence in an interactive session is as follows:

1. Starting from the system level (prompt S:) enter LIBMAINT using the LM JCL statement to drop down to the command level (prompt C:)

2. Use the EDIT command to enter into the editor and write or edit an executable FORTRAN program

3. Save the program source using the editor command Z

4. Exit from the editor and LIBMAINT using the "/" command to return to system level

5. Compile the program using the FOR77 (or F7C) JCL statement

6. Link it using LINKER

7. Execute the program using the STEP and ENDSTEP JCL statements.

An example of such a sequence is given below where a program P is created as a member of the library Z2.SLLIB and executed. Except for the underlined prompts, the system responses are not shown.
Example 4.1:

S: LMN SL, LIB=(Z2.SLLIB, DVC=MS/M452, MD=K104);
C: EDIT;
R: A
I: PROGRAM P
I: WRITE(*,*)'OK'
I: STOP
I: END
I: [F
R: Z(FOR) P
R: Q
C: QUIT;
S: F7C SOURCE=P, INLIB=(Z2.SLLIB, DVC=MS/M452, MD=K104);
S: LINKER P;
S: STEP P, TEMP;
S: ENDDATE;

On execution of the program P, "OK" appears on the terminal screen. This is produced by the WRITE statement in the FORTRAN program. Data making up the compile unit produced by the compiler and the load module produced by the linker is stored in temporary libraries and is deleted when the terminal is logged off. In order to keep it, permanent libraries must be specified.


4.3 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 4.2:

S:   LMN SL,LIB=(Z2.SLLIB,DVC=MS/M452,MD=K104);
C:   EDIT;
R:   A
I:   $JOB JCLP ,USER=RADOUAN,REPEAT;
I:   FOR77 SOURCE=P,INLIB=(Z2.SLLIB,DVC=MS/M452,MD=K104);
I:   LINKER P;
I:   STEP P,TEMP,CPTIME=5000,LINES=1000;
I:   ENDSTEP;
I:   $ENDJOB;
I:   [F
R:   Z(JCL) JCLP
R:   Q
C:   QUIT;
S:   SJ JCLP:Z2.SLLIB:K104:MS/M452

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 forces termination of the program if it is caught in an endless loop.
4.4 OPTIONS STRING

DPS 7 FORTRAN 77 does not use the OPTIONS parameter for file preconnection, as opposed to release 1E FORTRAN. The character string specified in the OPTIONS parameter is reserved to the user. Its length can be obtained by the ITEST intrinsic function and its value by the KTEST intrinsic function.

Example 4.3:

Source program:

```fortran
PROGRAM P

character*255 KTEST,STRING
integer LENGTH
LENGTH=ITEST('OPTIONS')
STRING=KTEST('OPTIONS')
IF(STRING(1:LENGTH).EQ.'CASE1')THEN
  ...
ELSEIF(STRING(1:LENGTH).EQ.'CASE2')THEN
  ...
ENDIF

character*255 KTEST,STRING
integer LENGTH
LENGTH=ITEST('OPTIONS')
STRING=KTEST('OPTIONS')
IF(STRING(1:LENGTH).EQ.'CASE1')THEN
  ...
ELSEIF(STRING(1:LENGTH).EQ.'CASE2')THEN
  ...
ENDIF
```

Note that if the syntax of the options string is complex, the intrinsic functions IPAST, IVERIF, KTRANS, and ITRT are useful to analyze it.

...
4.5 JCL STATEMENTS USING PERMANENT LIBRARIES

In the following example, permanent libraries are used for the source, compile unit, and load module.

`FOR77  SOURCE=P
       ,INLIB=(TEST_70.SLLIB,DVC=MS/M452,MD=K104)
       ,CULIB=(FORTR_70.CULIB,DVC=MS/M452,MD=K104);`

`LINKER  P
       ,OUTLIB=(FORTR_70.LMLIB,DVC=MS/M452,MD=K104);`

`STEPP (FORTR_70.LMLIB,DVC=MS/M452,MD=K104)
       ,OPTIONS='CASE1';`
4.6 BATCH OR INTERACTIVE DEBUGGING

The Program Checkout Facility is a system resident program that you can use to debug your executable program. It is especially useful in interactive mode, as you can start execution, stop at any line, modify variables, and restart execution, without the need to recompile the program.

You can designate places in your program by the name of subroutines and their source lines, and you can designate symbols by their source names (as opposed to their physical addresses).

The principal 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 source line, if a condition is verified.
- **TRACE** to print the names of subroutines, or the labels by which execution passes them.

A complete description is given in the *Program Checkout Facility User Guide*. This system program has the same syntax of commands to debug FORTRAN, COBOL, and GPL.
4.7 **EXCEPTION CONTROL IN INTERACTIVE MODE**

If an exception occurs when a program is being executed under the Program Checkout Facility, control is given to the Exception Handler.

Typical exceptions are:

- **access out of segment bounds:**
  when addressing an array with a wrong subscript expression, and if the SUBCK parameter was not specified on the FOR77 JCL statement

- **fault data descriptor:**
  when calling a procedure not linked with the calling program

- **floating point data underflow,** if there is a computation of a real, double precision, or quadruple precision type, whose theoretical value is less than the minimum value that can be represented in the DPS 7.
  This exception is, by default, masked, but can be unmasked (with zero result), by calling the LSETS intrinsic function.

**Example 4.4:**

```
LOGICAL = LSETS('UNDERFLOW','MASK')
```

- **floating point data overflow,** if there is a computation of a real, double precision or quadruple precision type, whose theoretical value is greater than the nominal value that can be represented in the DPS 7.

**Example 4.5:**

This example shows how to take control from an overflow exception and how to correct and continue processing, in interactive mode.

Source program with lines numbered by the compiler:

```
1    PROGRAM Z2OVERF
2    CALL S1
3    STOP
4    END

1    SUBROUTINE S1
2    REAL A,B,C
3    A = 1.E70
4    B = 1.E70
5    C = A*B
6    RETURN
7    END
```

The interactive session, starting from the prompt S: is show below.
Only prompts and useful system messages are shown in this example. They are underlined.

```
S: F7C SOURCE=Z2OVERF, INLIB=(Z2.SLLIB, DVC=MS/M452, MD=K104), DEBUG;
S: LINKER Z2OVERF;
S: STEP Z2OVERF, TEMP, DEBUG;
S: ENDS;
  *DEFAULT APPY: Z2OVERF

... PCF AT PG INIT
D: GO;
*** EXCEPTION 10-00: FLOATING POINT DATA OVERFLOW

... PCF AT LINE: 0.0 ILN: 5 IN S1
D: DUMP A, B;

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.1000000E+71</td>
</tr>
<tr>
<td>B</td>
<td>0.1000000E+71</td>
</tr>
</tbody>
</table>

D: CHANGE B = 1.E0;

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.1000000E+01</td>
</tr>
</tbody>
</table>

D: GOTO ILN 5;
```

This leads to a correct termination of the executable program, with reexecution of the statement C = A*B.

Note that Internal Line Numbers (ILN) are used in this example.

Note also that both the compilation and execution JCL must specify the DEBUG parameter.
4.8 DEBUGGING AND MAINTENANCE WITH PERMANENT SOURCE CODE

If you have a large executable program, debugging with the Program Checkout Facility as explained in the previous paragraphs may lead you to introduce the same debugging commands several times. Furthermore, if you are in a maintenance stage, you can trace your executable program without the need to study each module in order to insert trace commands.

A classical method is to introduce debugging code when writing your programs. This code is in normal FORTRAN and, typically, performs the following type of action:

- print module names and dummy arguments on entry
- check that these arguments are in the proper range
- print the main files or tables of the program

To avoid loss of execution speed you can set certain flags by using the OPTIONS string to indicate that you are in debugging mode. The debugging code is executed only if these flags are set.

Furthermore, you can avoid the loss in memory space of this debugging code for the current version, but go back to a debug version, without source code modification. This is achieved by setting the characters ** in columns 1 and 2 of the debugging code lines. If you specify the DEBUGMD parameter of the FOR77 JCL statement, those lines are considered as if there were no asterisks. If you specify the NDEBUGMD parameter, those lines are considered as comments (which is standard ANSI77 FORTRAN). Note that this conditional compilation is not exclusive from the conditional execution with flags, as several flags allow the amount of traces to be tuned.

Except if you want strict presentation, the use of list directed formatting and of character constants is especially suited for traces.
Example 4.6:

Source program:

```fortran
PROGRAM MAINTAIN
    ... decoding of the options string
    ... obtained by the ITEST and KTEST functions
    CALL S1(A, B)
    ...
END
SUBROUTINE S1(X, Y)
c this common contains the flag values
c derived from the options string
**    COMMON /DEBUG/IFLAG1, IFLAG2
    ...
**    IF (IFLAG1 .EQ. 1) THEN
c IFLAG1 gives the minimum trace made of
c the procedure names, and parameters names and values
**    WRITE(*,*)'SUBROUTINE S1 X=',X,' Y=',Y
**    IF (IFLAG2 .EQ. 1) THEN
    c IFLAG2 gives heavy tracing by calling the subroutine
    c SDEBUG which prints all the COMMON elements
    c for the COMMON blocks of the executable program.
c As SDEBUG is independently compiled, it is
    c brought into memory by Virtual Memory Management,
c only if IFLAG2 is set.
**    CALL SDEBUG
**    ENDIF
**    ENDIF
    ...
END
```

JCL statements for debugging mode:

```
F77 SOURCE=MAINTAIN,DEBUGMD,INLIB=...
LINKER MAINTAIN(...) ;OPTIONS='FLAG1,FLAG2';
ENDSTEP;
```
5. Representation and Manipulation of Data by the Hardware

This section describes hardware representation of FORTRAN data after the FORTRAN compiler translation, and the hardware interpretation of FORTRAN operations. In particular, details are given concerning real, double precision, and quadruple precision data with information on rounding, truncation, normalization, overflow, and underflow.

5.1 DATA REPRESENTATION

5.1.1 Format of Information in Main Storage

The basic element of information in main store that is handled by machine instructions is the byte (eight bits). A group of two consecutive bytes forms a halfword. Four consecutive bytes form a word. An address defines the location of a byte in main storage. The location of a group of bytes (e.g., halfword, word) is defined by the address of the leftmost byte. Consecutive bytes from left to right are defined by consecutive increasing addresses. A group of bytes is called half-word, word, or doubleword aligned, if its address is a multiple of two, four, or eight, respectively.

The bits forming a byte are defined from left to right and are numbered zero through seven. Byte format is represented as follows:

```
 0 1 2 3 4 5 6 7
```
5.1.2 Short Integer Data

a. **Size** - Two consecutive bytes (halfword)
b. Signed and in two's complement form (the sign is defined by the leftmost bit and is equal to one if negative).
c. Integer with the assumed radix point to the right of the least significant bit.
d. **Storage location** - Specified by the address of the leftmost byte.

5.1.3 Integer Data

a. **Size** - Four consecutive bytes (word)
b. Signed and in two's complement form (the sign is defined by the leftmost bit and is equal to one if negative).
c. Integer with the assumed radix point to the right of the least significant bit.
d. **Storage location** - Specified by the address of the leftmost byte.

5.1.4 Real Data

a. **Value** - The value of a real number is defined by the following equation:

\[ V = (-1)^S \times 16^E \times .M \]

Where

\[ E = C - 64 \]

S is the sign, E is the exponent, C is the characteristic, and M is the mantissa of the real number.

The value zero is represented by a real number with the mantissa equal to zero. The value of true zero is represented either by a real number with all 32 bits equal to zero, or by a real number with the left hand bit set to one and the 31 right hand bits equal to zero.
b. **Format** - A real number occupies four consecutive bytes.

The format is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>C</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 5-1. Hardware Representation of Real Number**

c. **Sign S** - The sign S of a real number is contained in bit 0.

\[
S = 0 \text{ positive sign}\\
S = 1 \text{ negative sign}
\]

d. **Characteristic C, Exponent E**. The characteristic C of a real number is contained in bits 1 through 7. Its range is \(0 < C < 127\).

The exponent E is the power to which 16 is raised in calculating the value of the real number. The exponent E is equal to \(C - 64\).

e. **Mantissa M** - The mantissa M is the hexadecimal number contained in bits 8 through 31. It consists of 6 hexadecimal digits. The radix point is to the left of the high order digit position.

\[
M = \sum_{i=1}^{6} M_i \times 16^{-i}
\]

f. **Storage location** - Specified by the address of the leftmost byte.

### 5.1.5 Double Precision Data

a. **Value** - The value of a double precision number is defined by the following equation:

\[
V = (-1)^S \times 16^E \times .M
\]

Where

\[
E = C - 64
\]

S is the sign, E is the exponent, C is the characteristic, and M is the mantissa of the double precision number.

The value zero is represented by a double precision number with the mantissa equal to zero. A value of true zero is represented by a double precision number with 63 right bits equal to zero.
b. **Format** - A double precision number occupies eight consecutive bytes. The format is as follows:

```
<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>7 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>
```

**Figure 5-2. Hardware Representation of Double Precision Number**

c. **Sign S** - The sign S of a double precision number is contained in bit 0.

\[ S = \begin{cases} 
0 & \text{positive sign} \\
1 & \text{negative sign} 
\end{cases} \]

d. **Characteristic C, Exponent E** - The characteristic C of a double precision number is contained in bits 1 through 7. Its range is 0 < C < 127.

The exponent E is the power to which 16 is raised in calculating the value of the floating point number. The exponent E is equal to C - 64.

e. **Mantissa M** - The mantissa M is the hexadecimal number contained in bits 8 through 63. It consists of 14 hexadecimal digits. The radix point is to the left of the high order digit position.

\[ M = \sum_{i=1}^{14} M_i \times 16^{-i} \]

f. **Storage location** - Specified by the address of the leftmost byte.

### 5.1.6 Quadruple Precision Data

a. **Value** - The value of a quadruple precision number is defined by the following equation:

\[ V = (-1)^S \times 16^E \times .M \]

Where

\[ E = C - 64 \]

S is the high-order sign, E is the exponent, C is the high order characteristic, and M is the mantissa of the quadruple precision number.
The value zero is represented by a quadruple precision number with the mantissa equal to zero. A value of true zero is represented by a quadruple precision number with 127 right bits equal to zero.

The low-order sign and the low-order characteristic do not take part in determining the value of the quadruple precision number.

b. Format - A quadruple precision datum occupies sixteen consecutive bytes. The format is as follows:

```
HIGH-ORDER
SIGN
```

```
S
```

```
HIGH-ORDER
CHARACTERISTIC
```

```
MANTISSA
```

```
0 1 7 8 31
```

```
MANTISSA
```

```
32 63
```

```
LOW-ORDER
SIGN
```

```
S
```

```
LOW-ORDER
CHARACTERISTIC
```

```
MANTISSA
```

```
64 65 71 72 95
```

```
MANTISSA
```

```
```

Figure 5-3. Hardware Representation of Quadruple Precision Number

c. Sign S - The sign S of a quadruple precision number is equal to the high-order sign contained in bit 0.

S = 0 positive sign
S = 1 negative sign

The low-order sign is contained in bit 64. Although the high-order and low-order sign may be set equal by floating point instructions, they are not required to be equal before instruction execution.
d. **Characteristic C, Exponent E** - The characteristic C of a quadruple precision number is equal to the high-order characteristic which is contained in bits 1 through 7. Its range is \(0 < C < 127\).

The exponent E is the power to which 16 is raised in calculating the value of the floating point number. The exponent E is equal to \(C - 64\).

The low-order characteristic is contained in bits 65 through 71. The low-order characteristic may be set 14 less than the high-order characteristic by floating point instructions, but it is not required to have this value before instruction execution.

e. **Mantissa M** - The mantissa M is the hexadecimal number contained in bits 8 through 63 and bits 72 through 127. These two fields are concatenated and treated as a single value in floating point instructions. The mantissa consists of 28 hexadecimal digits. The radix point is to the left of the high order digit position.

f. **Storage location** - Specified by the address of the leftmost byte.

### 5.1.7 Character Data

a. Character data is contained in consecutive bytes with one character per byte.

b. **Storage location** - The storage location of a character number is specified by the address of the byte.

c. **Character representation** - Characters are represented internally as shown in Appendix A of the DPS7 FORTRAN 77 Reference Manual.

### 5.1.8 Logical Data

a. **Size** - Four consecutive bytes (word)

b. The value false is '0000 0000'Z
   The value true is '0000 0001'Z

c. **Storage location** - Specified by the address of the leftmost byte.

### 5.1.9 Short Logical Data

a. **Size** - One byte

b. The value false is '00'Z
   The value true is '01'Z
5.1.10 Complex Data

A complex number is made of two real data in consecutive bytes.

5.1.11 Complex Double Precision Data

A complex double precision number is made of two double precision data in consecutive bytes.
5.2 FLOATING POINT OPERATIONS

In most applications, knowledge of the machine instructions and registers is never needed. However, in some computations which are sensitive to rounding, the programmer may find the following information useful. In what follows, floating point numbers mean real, double precision or quadruple precision values.

5.2.1 Scientific Registers

Scientific registers are used in the manipulation of floating point numbers. There are four scientific registers. The size of a scientific register is eight bytes. The scientific registers are specified by addresses ranging from zero through three.

A real value is stored in the byte positions zero through seven of a scientific register. A real value stored in a scientific register has the same representation as a double precision value in storage (refer to the paragraph above on double precision datum). Quadruple precision values can be stored in scientific registers.

The representation of a floating point number in a scientific register is as follows:

```
    S   C   MANTISSA
  0  1  7  8  |   |   |   |

    MANTISSA
  32 |   |   |   |

  32  |   |   |   |

Figure 5-4. Hardware Representation of Floating Point Numbers
```
5.2.2 Characteristics of Floating Point Operations

When a real value is loaded into a scientific register, it must first be converted to a double precision value. This is done by extending the real value number to the right with eight ZERO digits. When the content of a scientific register is stored as a real value, it is rounded or truncated to six hexadecimal digits.

If the FORTRAN program is compiled with the ROUND option, a store instruction is generated with a ROUND bit, leading to rounding at execution.

If the FORTRAN program is compiled with the NROUND option, a store instruction is generated without a ROUND bit. This leads to a truncation on a DPS7/X0 or DPS7/X17-X27 model, or to a rounding on a 64/DPS, DPS7/X5 or DPS 7000 model.

5.2.3 Locations of Operands and Results

When a floating point operand or result is in storage, an address is derived from the address syllable (AS) field in the instruction, according to the storage access mechanism. This is the address of the leftmost byte of the floating point number. The result of a floating point operation can be in a scientific register. The compiler produces generated code that maintains the results in a register as long as possible. If the program is not compiled with the DEBUGMD option, the result can be kept in a register for a sequence corresponding to more than one FORTRAN 77 statement.

5.2.4 Guard Digit

The floating point operations (addition, subtraction, multiplication, division) on floating point values are performed with one more hexadecimal digit than needed by the scientific registers. This extra digit is called the guard digit. The formats of these intermediate results are shown in Figures 5-5 and 5-6.

Depending on the value of the ROUND bit in the machine instruction, rounding or truncation occurs. If the FORTRAN program is compiled with the ROUND option, rounding occurs. If the FORTRAN program is compiled with the NROUND option, truncation occurs (on any model).
Figure 5-5. Format of 15 Digit Intermediate Result

Figure 5-6. Format of 29 Digit Intermediate Result (con’t)
5.2.5 **Normalization**

A floating point number is normalized when the leftmost hexadecimal digit of the mantissa is non-zero.

Five machine instructions (Store Integer from Scientific Register to General Register, Divide Scientific Register by Storage, Divide Scientific Register into Scientific Register, Compare Storage with Scientific Register, and Compare Scientific Register with Scientific Register) require the operands to be normalized or to be zero in value. (The operand used as the divisor instruction can not be zero. If it is zero, a floating point divide exception occurs. If the operands are not normalized, an illegal floating point data exception occurs.) The remaining machine instructions do not require their operands to be normalized. In the add, subtract and multiply instructions, however, significance can be lost if each operand is neither normalized nor true zero.

The add, subtract, multiply, divide machine instructions perform normalization on intermediate results. Normalization consists of shifting the mantissa portion of the floating point number left one hexadecimal digit at a time until the leftmost digit is non-zero. The characteristic is reduced by the number of shifts, and zeros are inserted in the vacated low order digit positions.

The add, subtract, and multiply machine instructions operating on quadruple precision values perform normalization and also produce a result with the low-order exponent made 14 less than the high order exponent.

5.2.6 **Underflow**

If the result of an addition, subtraction, multiplication, or division produces a result that although non-zero, is too small to be represented, an underflow occurs. A Characteristic Underflow Exception occurs if the intrinsic function LSETS has previously been called, with the following arguments: LSETS (‘UNDERFLOW’, ‘UNMASK’). A zero result without any exception is given if LSETS has previously been called with the following arguments:LSETS (‘UNDERFLOW’, ‘MASK’);

In the absence of a call to LSETS by the executable program, LSETS has the following default arguments: LSETS (‘UNDERFLOW’, ‘MASK’).

5.2.7 **Overflow**

If the result of an addition, subtraction, multiplication, or division is too large to be represented, an overflow occurs. This leads to a Characteristic Overflow Exception. (Refer to Section 4 for recovery of exceptions).
FORTRAN 77
6. Calling and Called Programs and Recursion Phenomena

This section describes how FORTRAN 77 can communicate with other languages.

Since an application can be divided up into independently compiled programs, it may be simpler to write some of these programs in other languages. Each program is compiled by its respective compiler to produce compile units. Provided that certain compatibility constraints are satisfied, these compile units can be successfully linked by the LINKER to form a single executable load module.

The load module may thus be an executable program (DPS7 FORTRAN 77 Reference Manual 2.4.2) made up of a FORTRAN 77 main program together with FORTRAN 77 procedures and procedures written in other languages.

6.1 LINKAGES SUPPORTED

An application can be made by linking FORTRAN 77, GPL, and COBOL 74 procedures or programs. The main program can be FORTRAN 77, GPL, or COBOL 74. The resulting linked application can be a load module, submitted for execution as a JCL step. Conditions under which the FORTRAN 77, COBOL, and GPL procedure or programs can share data or files are specified in this chapter.

No other environments, languages or conditions are supported, even though they may be made to work in simple cases.
6.2 LINKAGES NOT SUPPORTED

Linkages not specified in the previous paragraph are not supported. The following cases are typical examples of non-supported cases, but not an exhaustive list:

- mixing FORTRAN 77 and GCOS1E FORTRAN: this results in a LINKER fatal error
- mixing FORTRAN 77 and APL
- executing FORTRAN 77 under TDS.

If it is desired to run FORTRAN 77 in a TDS supported environment, the following suggestions should be helpful.

- do not use COMMON statements
- do not use input/output statements (including ENCODE/DECODE or internal READ/WRITE)
- do not call intrinsic functions implemented as mathematical procedures. It may happen that execution is correct in certain cases, but that an abort occurs at run time in cases leading to a run time error message.
- do not use exponentiation with non-integer exponents: this leads to a call of the Mathematical Library, and eventually to a run time error message in case of overflow.
- do not use character arguments
- do not use a SAVE statement
- do not compile with the SUBCK parameter of the JCL FOR77 statement.
### 6.3 CORRESPONDANCE BETWEEN FORTRAN, COBOL, AND GPL DATA TYPES

**Table 6-1. Correspondence Between Data Types**

<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>COBOL</th>
<th>GPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER*2</td>
<td>COMP-1</td>
<td>FIXED BIN (15)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COMP-2</td>
<td>FIXED BIN (31)</td>
</tr>
<tr>
<td>REAL</td>
<td>COMP-9</td>
<td>/</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>COMP-10</td>
<td>/</td>
</tr>
<tr>
<td>CHARACTER*Cst</td>
<td>PIC X (Cst)</td>
<td>CHAR(Cst)</td>
</tr>
<tr>
<td>CHARACTER*(*)</td>
<td></td>
<td>CHAR(*)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHAR(Var)</td>
</tr>
</tbody>
</table>

where 'Cst' means constant expression, and 'Var' means variable expression.
6.4 FORTRAN 77, COBOL, AND GPL STORAGE DATA

Table 6-2. Storage Data

<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>COBOL</th>
<th>GPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETER</td>
<td>% REPLACE</td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>STATIC</td>
<td></td>
</tr>
<tr>
<td>Not SAVE</td>
<td>AUTOMATIC</td>
<td></td>
</tr>
<tr>
<td>COMMON /C/</td>
<td>EXTERNAL</td>
<td>EXTERNAL WITHIN C</td>
</tr>
</tbody>
</table>

Example of external data exchange between FORTRAN, GPL, and COBOL:

SUBROUTINE FORTRAN
COMMON/C/C1,C2
CHARACTER*100 C1,C2

CALLGPL
CALLCOB
END

GPL: PROC;
DCL 1 C EXTERNAL WITHIN C,
   2 C1 CHARACTER (100),
   2 C2 CHARACTER (100);

END;

COBOL: IDENTIFICATION DIVISION
PROGRAM-ID. COB.

DATA DIVISION
WORKING-STORAGE SECTION.

01 C EXTERNAL.
   02 C1 PIC X(100).
   02 C2 PIC X(100).

EXIT PROGRAM.
The LINKER JCL statement must specify: PLACE=(C,C),MSEGAT= (GLOBSEG,C,...) because FORTRAN 77 allocates one common named C into a segment of name C, whereas COBOL-74 allocates externals into the blank segment. The blank common of FORTRAN 77 is of no use as it is allocated into the segment named *CB. (The reason being that the blank common may have different lengths in different program units, and so must be treated as a special segment).
6.5 PASSING ARGUMENTS FOR DPS 7 FORTRAN 77

From the programming language point of view, arguments are passed by reference or by value. Because several programs written in several programming languages can be linked, it is necessary to know how the argument passing is implemented by each compiler. This is detailed below.

For the DPS 7 FORTRAN 77, arguments are passed by address or by descriptor.

1. **Address**: the address of the argument is passed in the hardware stack.

   This is always the case if the FORTRAN 77 program is compiled with the LEVEL = GCOS1E parameter in the FOR77 JCL statement. But in this case, assumed size characters are rejected by the compiler. Furthermore, linking programs (compiled with or without LEVEL = GCOS1E) can lead to an execution abort if they exchange character arguments. If the FORTRAN 77 program is compiled with the default parameter LEVEL = GCOS7, then the address of the argument is passed in the following cases:

   - arithmetic, logical or Hollerith constants: the address of the constant is passed. This may result in an exception if the calling procedure attempts to modify its corresponding dummy argument (passing by value, checked).
   - arithmetic or logical variable, array, array element or one of them enclosed in (ignored) parentheses. The address of the datum is passed (passing by reference).
   - arithmetic or logical expression not reduced to one of the preceding cases. The value of the expression is moved into an area of the hardware stack, whose address is passed (passing by value, unchecked).

2. **Descriptor**: the address of a descriptor is passed in the hardware stack. The descriptor is made of two words:

   - first word: address of the argument (GPL POINTER)
   - second word: length of the argument (GPL FIXED BIN(31))

   If the length is known at compile time, the descriptor is allocated like a SAVE datum, so that the length is statically initialized. If the length is not known at compile time, the descriptor is allocated in the hardware stack.

   A descriptor is passed if the argument is of type character. (The reason is that the called procedure may declare an assumed size length for the corresponding dummy argument.)

   The address in the descriptor is the following:

   a) For a character constant: the address of the constant. This may result in an exception if the called procedure attempts to modify its corresponding dummy argument (passing by value, checked).

   b) For a variable, array element, array substring or one of them enclosed in (ignored) parentheses, the address of the datum is passed (passing by reference).

   c) For a character expression, not reduced to one of the previous cases, the value of the expression is moved into an area of the hardware stack, whose address is passed (passing by value, unchecked).
The following example illustrates the mode of passing: by value or by reference, as a consequence of the implementation rules. Note that the EVALUATE intrinsic function evaluates its arguments and returns their value. So it may be used to force passing by value.

```
PROGRAM P
INTEGER I, IT(10)
CHARACTER C, CT(10)
* Passing the called procedure by reference or descriptor
* may modify its argument.
CALL S1 (I)
CALL S2 (IT(I))
CALL S3 (IT)
CALL S4 (C)
CALL S5 (CT(I))
CALL S6 (C(1:1))
* passing by value (the parentheses are ignored)
CALL S1 ((I))
CALL S4 ((C))
* passing by value, but memory protection
* does not allow modification by the callee.
CALL S1 (I)
CALL S4 ('A')
* passing by value: modification by the callee
* has no effect. (EVALUATE is an intrinsic function
* returning the value of its argument).
CALL S1(I+1)
CALL S4(C/C)
CALL S1(EVALUATE(I))
CALL S4(EVALUATE(C))
STOP
END
```
6.6 FORTRAN 77, COBOL, AND GPL ARGUMENT CORRESPONDENCE

There is no difficulty in exchanging integers (as shown by examples 6-1 and 6-2) but there is a problem exchanging characters between FORTRAN 77 and COBOL 74. This is because arguments are passed by descriptor in FORTRAN 77 but by address in COBOL 74.

There are three solutions to this problem:

(1) Use a GPL intermediate procedure, as shown in examples 6-1 and 6-2.

(2) Compile the FORTRAN 77 program with the LEVEL=GCOS1E parameter, as shown in example 6-3.

(3) Use the REFCHAR intrinsic function, as indicated in the FORTRAN 77 Reference Manual, Table 15-1.

Example 6-1. FORTRAN 77 calls GPL and COBOL:

* FORTRAN MUST NOT PASS CHARACTER TO COBOL
SUBROUTINE FORTRAN
INTEGER I
CHARACTER *100 C
REAL A
...
CALL COBI (I,A)
CALL GPL (C)
END
...
PROGRAM-ID. COBI.
...
WORKING-STORAGE SECTION.
...
LINKAGE SECTION.
77 I COMP-2.
77 A COMP-9.
PROCEDURE DIVISION USING I,A.
...
EXIT PROGRAM.
/* this GPL procedure transforms a descriptor */
/* into an address */
GPL: PROC(C);
   DCL C CHAR(*);
   DCL I FIXED BIN(31);
   DCL COBC ENTRY (CHAR(500),FIXED BIN(31));
   I = MEASURE(C);
   CALL COBC(C,I);
   RETURN;
END;
Calling and Called Programs and Recursion Phenomena

Example 6-2. FORTRAN is called by COBOL and GPL:

```fortran
PROGRAM-ID. COB.
...
WORKING-STORAGE SECTION.
...
PROCEDURE DIVISION.
...
MOVE 100 TO I.
CALL "FORI" USING I.
CALL "GPL" USING C,I.
EXIT PROGRAM.
SUBROUTINE FORI(I)
INTEGER I
...
RETURN
END
GPL:PROC(C,I);
DCL C CHAR (I);
DCL FORC ENTRY(CHAR(*));
CALL FORC (C);
RETURN;
END;
SUBROUTINE FORC(C)
CHARACTER*(*)C
I = LEN(C)
RETURN
END
```
Example 6-3. FORTRAN 77 exchanges characters with COBOL and GPL:

*     FORTRAN 77 IS COMPILED WITH LEVEL = GCOS1E
PROGRAM FORTRAN1
CHARACTER *100 C1
CALL COB (C1)
CALL GPL (C1)
END
...
IDENTIFICATION DIVISION.
PROGRAM-ID. COB.
DATA DIVISION.
WORKING-STORAGE SECTION.
...
77 C2 PIC X(100).
...
LINKAGE SECTION.
77 C1 PIC X(100).
PROCEDURE DIVISION USING C1.
...
CALL FORTRAN2 USING C2.
...
*     FORTRAN 77 IS COMPILED WITH LEVEL = GCOS1E
SUBROUTINE FORTRAN2(C2)
CHARACTER *100 C2
RETURN;
END;
GPL :   PROC(C1);
D CL C CHAR(100)
D CL FORTRAN2 ENTRY CHAR(100);
C ALL FORTRAN2(C2)
R ETURN;
E ND;
6.7 RETURNING FUNCTION VALUES

- INTEGER function returns its value in general register G7 (32 bits)
- INTEGER *2 function returns its value in general register G7
- REAL function returns its value in scientific register SO (64 bits), as any REAL expression evaluated in register.
- DOUBLE PRECISION function returns its value in scientific register SO (64 bits)
- QUADRUPLE PRECISION function returns its value in scientific registers SO, S1 (64 bits each)
- COMPLEX function returns its value in scientific registers SO, S1 (64 bits each)
- COMPLEX DOUBLE PRECISION function returns its value in scientific registers SO, S1 (64 bits each)
- LOGICAL function returns its value in general register G7
- LOGICAL *2 function returns its value in general register G7
- CHARACTER function is called with one more argument which is a descriptor of the character value, as for a usual character argument. This supplementary argument is the first (before the first actual argument). The descriptor is passed by passing its address in the hardware stack. The length in the descriptor is the length specified by the calling procedure.

- The convention of returning the function value as the first argument is also used by GPL but there is no CHAR(*) function in GPL. So a FORTRAN 77 character function must be called by GPL, by a CALL statement with one more first argument.
6.8 FORTRAN 77 AND GPL FUNCTIONS

Example 6-4. GPL calls FORTRAN 77 (integer case):

\[ \text{P: PROC;} \]
\[ \quad \% \text{ REPLACE IBOUND BY 1000;} \]
\[ \quad \text{DCL IFUN ENTRY ((IBOUND) FIXED BIN(31), FIXED BIN(31))} \]
\[ \quad \text{RETURNS (FIXED BIN(31));} \]
\[ \quad \text{DCL ITAB(IBOUND) FIXED BIN(31);} \]
\[ \quad \text{DCL I FIXED BIN(31);} \]
\[ \quad \text{I = IFUN(ITAB, IBOUND);} \]
\[ \quad \text{RETURN;} \]
\[ \quad \text{END;} \]
\[ \times \quad \text{FORTRAN 77} \]
\[ \quad \text{INTEGER FUNCTION IFUN(ITAB, IBOUND,} \]
\[ \quad \quad \text{INTEGER ITAB (IBOUND)} \]
\[ \quad \quad \text{INTEGER IBOUND} \]
\[ \quad \text{IFUN = ITAB(IBOUND)} \]
\[ \quad \text{RETURN} \]
\[ \quad \text{END} \]

Example 6-5. FORTRAN 77 calls GPL (exactly the reverse of the above case):

\[ \text{SUBROUTINE P} \]
\[ \quad \text{EXTERNAL IFUN} \]
\[ \quad \text{INTEGER IFUN} \]
\[ \quad \text{INTEGER IBOUND} \]
\[ \quad \text{PARAMETER (IBOUND = 1000)} \]
\[ \quad \text{INTEGER ITAB(IBOUND)} \]
\[ \quad \text{INTEGER I} \]
\[ \quad \text{I = IFUN(ITAB, IBOUND)} \]
\[ \quad \text{RETURN} \]
\[ \quad \text{END} \]

\[ /* \text{GPL} */ \]
\[ \text{IFUN: PROC (ITAB, IBOUND) RETURNS (FIXED BIN)} \]
\[ \quad \text{DCL ITAB (IBOUND) FIXED BIN(31);} \]
\[ \quad \text{DCL IBOUND FIXED BIN(31);} \]
\[ \quad \text{RETURN (ITAB(IBOUND));} \]
\[ \quad \text{END;} \]
Example 6-6. GPL calls FORTRAN 77 (characters):

```plaintext
P: PROC;
   % REPLACE IBOUND BY 1000;
   % REPLACE LG BY 1000;
   DCL CFUN ENTRY ((IBOUND)CHAR(*), FIXED BIN(31))
      RETURNS (CHAR(LG));
   DCL CTAB (IBOUND) CHAR (LG);
   DCL C CHAR (LG);
   C = CFUN(CTAB, IBOUND);
   RETURN;
END;

*FORTRAN 77
   CHARACTER (*) FUNCTION CFUN(CTAB, IBOUND)
   CHARACTER (*) CTAB(IBOUND)
   INTEGER IBOUND
   CFUN = CTAB(IBOUND)
   RETURN
END
```

Example 6-7. FORTRAN77 calls GPL:

```plaintext
SUBROUTINE P
   PARAMETER (IBOUND = 1000)
   PARAMETER (LG = 1000)
   EXTERNAL CFUN
   CHARACTER *(LG) CFUN
   CHARACTER *(LG) C
   C = CFUN (CTAB, IBOUND)
   RETURN
END

/* GPL */
CFUN: PROC(CTAB, IBOUND) RETURNS CHAR(*),
   DCL CTAB(IBOUND) CHAR(*);
   DCL IBOUND FIXED BIN(31);
   RETURN(CTAB(IBOUND));
END;
```
6.9 EXCHANGE OF FILES BETWEEN FORTRAN 77 AND COBOL

6.9.1 Files Preconnected to System Units

The FORTRAN statements READ (*,...) and WRITE (*,...) use the same files, respectively as the COBOL statements ACCEPT and DISPLAY. In batch mode, the system input file must be assigned with the ifn H_RD, the system output file can be assigned with the ifn H_PR. The records of these files can be read or written indifferently by a FORTRAN or a COBOL subprogram, provided that the first READ or the first WRITE has been executed by a FORTRAN subprogram.

Example:

FORTRAN:               COBOL:
...                  ...                  
CHARACTER *72 A         PROGRAM_ID. COBSUB.  
...                  ...                  
READ (*,8 1)A           01 A PIC X(72)       
...                  ...                  
WRITE (*,82)A           PROCEDURE DIVISION.   
81 FORMAT(A72)           ...                  
82 FORMAT (1X,A80)       ACCEPT A.               
DO 10 i=1,3       DISPLAY A.                 
CALL COBSUB       PET. EXIT PROGRAM.         
REAP (*,81)A                        
WRITE (*,82)A                        
10 CONTINUE                  
STOP                             

In this example, the odd records are read and written by FORTRAN, the even records are read and written by COBOL.
6.9.2 Other Shareable Files

They must be connected by a FORTRAN 77 statement OPEN (u, FILE=IFN=HCOBFORi) where i = 1, 2, 3, 4 or 5, and declared in the COBOL program by a SELECT clause with an EXTERNAL attribute and the same internal file name HCOBFORi:

SELECT EXTERNAL fname ASSIGN TO HCOBFORi.

They must be assigned by a JCL statement such as

$ASSIGN HCOBFORi efn...;

Two cases may be distinguished:

a) The file need not be accessed by COBOL (or FORTRAN) while it is accessed by FORTRAN (COBOL). There is no "mixing" of records. In this case, the file can be opened, treated, and closed by FORTRAN (COBOL), then opened, treated, and closed by COBOL (FORTRAN), and so on.

Example:

COBOL:

....
    PROGRAM ID_COBMAIN.
    ....
    SELECT F1 EXTERNAL ASSIGN TO HCOBFOR1.
    SELECT F2 EXTERNAL ASSIGN TO HCOBFOR2.
    ....
    OPEN INPUT F1 OUTPUT F2.
    PERFORM F1-TO-F3 3 TIMES.
    CLOSE F1 F2.
    CALL "FORSUB".
    STOP RUN.
    F1-TO-F2.
    READ F1 AT END STOP RUN.
    WRITE F2-RECORD FROM F1-RECORD.

FORTRAN:

SUBROUTINE FORSUB
    CHARACTER*50 C
    OPEN (1,FILE= IFN=HCOBFOR1 ,FORM= UNFORMAT)
    OPEN (2,FILE= IFN=HCOBFOR2 ,FORM= UNFORMAT)
    DO 10 i=1,3
    READ(2) C
    WRITE(1) C
    10    CONTINUE
    RETURN
END

JCL:

STEP COBMAIN...;
ASG HCOBFOR1 EFN1...;
ASG HCOBFOR2 EFN2...;
b) The file needs to be accessed simultaneously by COBOL and FORTRAN; there is "mixing" of records. In this case, the following rules must be observed.

- The file must be opened by a FORTRAN OPEN (u, FILE = IFN=HCOBFORi) statement.
- The file must be used by COBOL with the same processing mode as the last time it was used by FORTRAN.

This rule is obeyed if:

- a COBOL sequential READ statement is executed after a FORTRAN sequential READ statement
- a COBOL sequential WRITE statement is executed after a FORTRAN sequential WRITE statement
- a COBOL relative READ/WRITE statement is executed after a FORTRAN direct READ/WRITE statement.
- If the file must be explicitly closed, it must be closed by a FORTRAN CLOSE (u) statement.

Example:

FORTRAN 77:

```
PROGRAM FORMAIN
  CHARACTER*50 C
  OPEN (1,FILE = IFN=HCOBFOR1 , FORM= UNFORMAT)
  OPEN (2,FILE = IFN=HCOBFOR2 , FORM= UNFORMAT)
  DO 10 i = 1,100
     READ (1) C
     READ (2) C
     CALL COBSUB
  10  CONTINUE
  STOP
  END
```

COBOL:

```
... PROGRAM_ID. COBSUB.
...
  SELECT F1 EXTERNAL ASSIGN TO HCOBFOR1.
  SELECT F2 EXTERNAL ASSIGN TO HCOBFOR2.
...
  READ F1 AT END STOP RUN.
  WRITE F2_RECORD FROM F1=RECORD.
  RET. EXIT PROGRAM.
```

JCL:

```
STEP FORMAIN
  ASG HCOBFOR1 EFN1...;
  ASG HCOBFOR2 EFN2...;
  EST;
```
6.10 EXCHANGE OF FILES BETWEEN FORTRAN 77 AND GPL

The rules of file exchange between FORTRAN and GPL are the same as those given for file exchange between FORTRAN and COBOL.

The following file descriptions must be used:

$H_FD H_RD, SYSIN;
$H_FD H_PR, SYSOUT, RECSIZE = 264, BLOCKSZ = 264, DATAFORM=SSF,
   NUMBUF = 2
$H_FD HCOBFOR1, FILEFORM = UFAS, RECSIZE =2032, BLOCKSZ = 2036,
   FILEORG = SE, ADDFORM = TTROP, DATAFORM = SARF, RECFORM = V,
   NUMBUF = 2;
$H_FD HCOBFOR2 ...

The ifn H_CP, referred to by the PUNCH FORTRAN statement, can also be used by means of the file description:

$H_FD H_CP, SYSOUT, DATAFORM = SARF

The other ifn's cannot be shared with those used by FORTRAN, because their file descriptions are dynamically created in the FORTRAN Run Time Package, by means of the primitive

$H_CRFD o_ptr, i_ifn,...;

6.11 JCL COMMUNICATION

The intrinsic functions IFRTWT, IFRSWT, KTEST and ITEST can be used. They are documented in the FORTRAN 77 Reference Manual.
6.12 RECURSIVITY OF FORTRAN 77 PROCEDURES

A function (FORTRAN 77 Reference Manual 15.2) or subroutine (FORTRAN 77 Reference Manual 15.5.3) must not reference itself directly or indirectly. If someone tries to do this, direct recursivity is rejected by the compiler. Indirect recursivity is not diagnosed either inside a package unit or as an external call.

In the case of indirect recursivity, recursivity occurs only if reentrance conditions are met. These conditions are indicated in the following paragraph.

Note that for a very simple case, if the procedure is independently compiled, recursivity occurs; a stack frame of the hardware stack is allocated at each activation.

If all procedures called are in the same package unit, then recursivity will not occur; the same stack frame is used at each activation.

Example of indirect recursivity for two procedures IFACT1,IFACT2:

```
PROGRAM P
WRITE(*,*) IFACT1(6)
STOP
END
FUNCTION IFACT1(I)
IF(I.EQ.1) THEN
    IFACT1=I*IFACT2(I-1)
ENDIF
RETURN
END
* THIS COPY OF IFACT1 IS NECESSARY BECAUSE DIRECT
* RECURSIVITY IS REJECTED
FUNCTION IFACT2(I)
IF (I.EQ.1) THEN
    IFACT2=2
ELSE
    IFACT2=I*IFACT1(I-1)
ENDIF
RETURN
END
```

In the previous example, if the program units are encapsulated into a package unit:

```
PACKAGE P
PROGRAM P
... FUNCTION IFACT1(I)
... FUNCTION IFACT2(I)
... ENDPACKAGE
```

then the second activations of IFACT1 and IFACT2 force the record return address in place of the first ones. The program enters an endless loop until an exception occurs.
6.13 REENTRANCE CONDITIONS OF A FORTRAN 77 PROCEDURE

DPS 7 FORTRAN 77 is reentrant except if:

- there is a SAVE statement (this leads to static allocation of the specified data)
- there is a NAMELIST statement (same reason)
- there is a call of a procedure with character arguments (same reason) except if the arguments are assumed to be dummy arguments of the calling procedure, and if they are variables. In this case, the address of the received descriptor is copied in the hardware stack without creation of a new discriptor
- the procedure is a character function (same reason)
- there is a FORMAT statement (same reason)
- there is an INPUT/OUTPUT statement (this leads to static allocation of a descriptor).
7. Segmentation and Addressing

Segmentation is the process of physically dividing a program into segments which can be located in main memory or on disk independently of each other during load module execution. The process of moving program segments between main memory and disk (swapping) is not the responsibility of the user program. It is handled by the system component "Virtual Memory Management". For a description of Virtual Memory Management see the *System Administrator's Manual*.

However, the programmer can influence the way in which the executable program is divided into segments. Good program segmentation achieves the following:

- The number of times segments are swapped between main memory and backing store is reduced. This has three benefits: first, the reduced rate of I/O minimizes queuing for the disk drives; second, the waiting time of the program for segments to be swapped is reduced as far as possible; third, the CPU time consumed by Virtual Memory Management is minimized.

- Execution of a load module can begin when only part of the load module is in main memory. This has two benefits; first, large load modules do not have to wait for a large amount of memory to become available at one time; second, large load modules that would be too large to fit into the available memory in one piece can be executed in segments.

The remainder of this section provides guidelines for efficient program segmentation.

**NOTE:** For an explanation of FORTRAN 77 segment number, internal segment number, and LINKER segment number, see Section 3, Linking.
7.1 GENERATED SEGMENTS

The number of segments of a compile unit generated by the compilation of a program unit or a package unit is determined as follows.

7.1.1 PSEGMAX, DSEGMAX, and DEBUG Parameters are Not Used

a) Each program unit not in a package unit leads to:
   - 1 segment per COMMON block.
   - 1 segment for local data specified in a SAVE statement, or initialized by a DATA statement, or for compiler generated data if there are I/O statements or adjustable arrays. If none of these features is present, the segment is not generated.
   - 1 segment for the code (executable statements).

b) Each package unit leads to:
   - 1 segment per COMMON block specified in the EXTCOMMON statement
   - 1 segment for local data specified in a SAVE statement, or initialized by a DATA statement, or for compiler generated DATA if there are I/O statements or adjustable arrays. If none of these features are present, the segment is not generated.
   - 1 segment for COMMON blocks not specified in the EXTCOMMON statement.
   - 1 segment for the code (executable statements).

c) For a program unit not in a package unit or for a package unit

Local data not specified in a SAVE statement nor initialized by a DATA statement is located in a stack frame of the hardware stack. This stack frame will be allocated at execution, on entry into the corresponding program unit or package unit, and deallocated on exit from the program unit or package unit. Notice that for a package unit, all this local data for all the program units of the package unit is allocated together at entry into the package unit, and deallocated together at exit from the package unit, but not at exit from a program unit of the package unit.

If scalar data not specified in a SAVE statement is larger than 10 Kbytes, or if the size of 10K is reached, additional local scalar data is allocated with data specified in SAVE statement, and an observation diagnostic message is issued. If array data not specified in a SAVE statement is larger than 8 Kbytes, or if the size of 8K is reached for all data arrays, additional local array data is allocated with data specified in the SAVE statement, and an observation diagnostic message is issued.
7.1.2 PSEGMAX or DSEGMAX Parameters are Specified

As many segments as necessary are generated to bring the total to the value specified in the parameter.

Notice that DSEGMAX has no effect on the stack frame.

7.1.3 DEBUG Parameter is Specified

In this case extra segments are generated.

There are as many segments as necessary so that their size does not exceed 64 Kbytes, or does not exceed the DSEGMAX value if DSEGMAX is used and is less than 64.

7.1.4 Evaluating the Number of Segments

The number of user segments of a load module, resulting from the linkage of several compile units is determined as follows:

• add up the number of code segments of each compile unit,

• add up the number of data segments of each compile unit, if these segments result from local data or COMMON blocks of a package not declared in the EXTCOMMON statement,

• add up the number of data segments resulting from COMMON blocks of program units not in a package, or COMMON blocks of a package declared in the EXTCOMMON statement, remembering that all COMMON blocks with the name, even if the name is blank, lead to only one segment in the load module.

In addition to these segments with STN = 8, 4 segments of STN = 8 are added for the I/O routines.

And finally, a few segments are used for the system, as indicated in the linkage map.
7.1.5 Guideline for Segmentation

Segmentation is a logical division of memory and offers a natural correspondence between FORTRAN objects and physical memory. It also allows an efficient protection at a logical level, preventing erroneous branching or data addressing during debugging.

In the case of a debugged application, you can tune its performance by considering the following suggestions.

1) Impact of the size of segments on Virtual Memory Management

- Performance is improved if all segments in the system are of approximately the same size.
- Large segments tend to be used (i.e., referenced) more often than small ones. For this reason Virtual Memory Management usually allows them to remain in memory for a longer time than small segments. On the other hand, once a large segment has been swapped out of memory considerable rearrangement of memory contents might be necessary in order to provide a large enough area of memory into which it will be swapped back.
- Conversely, smaller segments tend to remain in memory for a shorter time.

In the absence of specific information, a suitable size for segment has a value, a little less or equal to, the size of the track on the system disk.

2) Case of an executable program producing small segments

An executable program is often made of a lot of small subroutines and a small main program. This is good practice of structured programming, but with standard FORTRAN may lead to great amount of small user segments (STN = 8) in the process.

The number of segments of STN = 8 in a process is limited to 256. So it may happen that the linkage may not be possible if there are too many subroutines.

Notice also that if the subroutines are small in code or data, this results in small segments, and is not good for the Virtual Memory Management.

Guideline:

a) The package feature is incorporated to improve the situation. The source code of subroutines and possibly the main program can be grouped into one or several packages. This will considerably reduce the number of segments of the process.

b) If, after using the package feature, the number of segments exceeds 256 but is less than 512, the segments in excess of 256 may be given STN = 9 by a LINKER command. These segments are shareable for the process group which has no consequence in FORTRAN, as you do not use several processes in the process group. Process groups with several processes are reserved for special use, such as TDS.
3) **Case of large arrays**

The DPS 7 Virtual Memory under GCOS 7 provides 512 short segments of up to 64 Kbytes and 12 large segments of up to 4096 Kbytes, for an executable program in a process group.

If there are large arrays in the executable program, the compiler can group them together with small data. This results in a segment too large for the physical memory or for efficient Virtual Memory Management.

**Guideline:**

a) By specifying each array alone in a COMMON statement, itself specified in an EXTCOMMON statement for a package unit, a segment is reserved for each array.

b) If all large arrays are of similar size, the largest of these sizes can be specified as the value of the DSEGMAX parameter. As a result, a segment is reserved for each array.
7.2 DATA ADDRESSING

The data access speed of the CPU when executing a program depends on the relative address of the data in its segment.

7.2.1 Alignment

A byte is a unit of 8 bits, a half word is 2 bytes, a word is 4 bytes, a double word is 8 bytes, and a quadruple word is 16 bytes.

a) For a model without cache memory:

DPS7/X5 or DPS 7000; data whose size is 2 or 4 bytes must be positioned on half word or word boundaries respectively, to obtain the maximum CPU speed. The degradation in speed, if all data is not optimally aligned is generally between 1 and 20 per cent.

b) For a model with cache memory:

DPS7/X0 or DPS7/X17-X27; data whose size is 2, 4, 8 or 16 bytes must be positioned on half word, word, double, or quadruple word boundaries respectively, to obtain the maximum CPU speed. The degradation in speed if all data is not so positioned is generally between 1 and 50 per cent.

7.2.2 Indexation

a) An array located at more than 4 Kbytes from the beginning of its segment can not be accessed directly with the base register at the beginning of the segment and the index value. Additional instructions are generated by the compiler to load a nearer base register or to add the array offset in the index.

b) When the computation of the array subscript contains a constant value known at compile time, the compiler generates instructions with an offset that is not the one of the array in the segment, but with a "Virtual Offset", as in the following example:

```fortran
REAL T (10)
A = T (I-1)
```

The offset of T is t
The virtual offset is t - 4
The physical index is 4*I

This is not possible if t<4, that is if T is allocated at the beginning of its segment, as the hardware does not accept an address part to be outside of its segment. This is useful for segment protection. So in this case the compiler generates extra instructions to compute I-1 in the index register.
7.2.3 Allocation Performed by the Compiler

To cater for the previous constraints, the compiler, linker, and loader proceed as follows.

A segment is allocated at a quadruple word boundary.

A stack frame is allocated at a word boundary.

Local data is allocated in a stack frame of the hardware stack, or in a segment, depending on SAVE and DATA statements.

Local data in a stack frame, or local data in a segment are allocated as follows:

a) Arrays are allocated at the beginning of the memory in order of increasing size, so that most of them are offset less than 4 K. The first array has an offset of a few hundred bytes, so that its "virtual offset" is generally inside the segment.

Each array is located at a boundary proper to the size of its elements, with suitable gaps between arrays. Array elements are always adjacent. Character arrays are located on a byte boundary (no alignment).

After the arrays, variables are grouped by blocks of variables of the same size. Each block is located on a boundary proper to its variables. Character variables are allocated without alignment after the previous blocks.

Compiler-generated data may be inserted between user data.

Compiler-generated format encoding is treated as character variables. Input/Output descriptors generated by the compiler are word aligned.

The few hundred bytes before arrays are used also for scalar data, properly aligned.

All sets of EQUIVALENCEd data are treated as unique occurrences in order to respect the equivalence constraints. This may result in inappropriate alignment of individual data items.

b) COMMON blocks are allocated at a quadruple word boundary. Common elements are allocated, within this storage space, in the order in which they have been specified, without any gaps between them, and respecting equivalences.
8. Efficiency Techniques

The following techniques are recommended to obtain efficient FORTRAN object programs. Consideration is given to data manipulation and data description techniques. See Section VII, Segmentation, for guidelines on efficient segmentation. See the UFAS User's Guide, UFAS-Extended User's Guide or BFAS User's Guide for guidelines on the efficient use of files.

Most of the suggestions are designed to reduce the execution time. Reduction of memory needed is achieved by the use of the PSEGMAX and DSEGMAX parameters of the FOR77 JCL statement, generally at the expense of execution time.
8.1 GENERAL GUIDELINES

If you measure the CPU time spent in the various parts of an executable program (refer to the SBR User’s Guide), you will observe that, in general, most of the CPU time is spent in one small kernel, or a few small kernels of this executable program. Consequently, to improve the execution speed of an executable program made of several program units linked together, it is generally sufficient to apply the efficiency techniques to these small kernels.

This has two advantages:

1) The program execution is improved with a modest programming effort.

2) The whole executable program remains easy to modify and understand, if it was initially programmed logically.

The small kernels may become difficult to modify and understand due to optimization of the programming. But this is acceptable if it is well documented as the critical part of the algorithm.

In the absence of direct measuring techniques, you can usually locate the critical kernels by the following rules:

a) The critical kernel(s) is(are) in the more nested DO loops of the executable program.

b) The more nested DO loops may be found fairly precisely by considering only DO statements, CALL statements and function calls (loops by IF-GOTO can generally be neglected).
The following is an example where all statements except DO, CALL and nested sequences of statements, are not shown:

<table>
<thead>
<tr>
<th>Executable program</th>
<th>Nesting level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM MAIN</td>
<td></td>
</tr>
<tr>
<td>DO 10</td>
<td></td>
</tr>
<tr>
<td>CALL S1</td>
<td>1</td>
</tr>
<tr>
<td>10 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>CALL S2</td>
<td>0</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE S1</td>
<td>1</td>
</tr>
<tr>
<td>CALL S3</td>
<td>1</td>
</tr>
<tr>
<td>DO 20</td>
<td></td>
</tr>
<tr>
<td>CALL S4</td>
<td>2</td>
</tr>
<tr>
<td>DO 20</td>
<td></td>
</tr>
<tr>
<td>CALL S5</td>
<td>3</td>
</tr>
<tr>
<td>20 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE S5</td>
<td>3</td>
</tr>
<tr>
<td>DO 30</td>
<td></td>
</tr>
<tr>
<td>sequence 5</td>
<td>4</td>
</tr>
<tr>
<td>30 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE S2</td>
<td>0</td>
</tr>
<tr>
<td>DO 40</td>
<td></td>
</tr>
<tr>
<td>sequence 2</td>
<td>1</td>
</tr>
<tr>
<td>40 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE S3</td>
<td>1</td>
</tr>
<tr>
<td>DO 10</td>
<td></td>
</tr>
<tr>
<td>DO 10</td>
<td></td>
</tr>
<tr>
<td>CALL S4</td>
<td>3</td>
</tr>
<tr>
<td>10 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE S4</td>
<td>max (3,2) = 3</td>
</tr>
<tr>
<td>sequence 4</td>
<td>3</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

Note that if a subroutine such as S4 is called in several places, its nesting level is the maximum of those of the calls.

The kernel is sequence 5 as it is inside four nested DO loops.

So at least the executable statements of sequence 5 must be optimized, together with the specification statements of the symbols that they use.
8.2 PARAMETERS OF THE FOR77 JCL STATEMENT

DSEGMAX generates supplementary machine instructions to access data in segments.

PSEGMAX generates supplementary machine instructions to branch from one segment to another. Conversely, they may reduce the time spent in Virtual Memory Management, in the case of program units or package units leading to large segments. A generally good choice is to use DSEGMAX or PSEGMAX only if the generated segments are larger than the size of the track on the system disk.

LEVEL = GCOS1E and LEVEL = SIRIS8 result in slower programs than if they are not specified.

SUBCK results in slower programs than NSUBCK.

ROUND results in slower programs than NROUND.
8.3 SPECIFICATION STATEMENTS

8.3.1 Data Type

LOGICAL *1 results in faster programs than LOGICAL.

REAL results in faster programs than DOUBLE PRECISION

CHARACTER *n with n less than or equal to 256 results in faster programs than with n greater than 256.

8.3.2 Constant Values.

PARAMETER statements result in faster programs than DATA statements. For example if CST is not to be modified in the program unit, use PARAMETER (CST = 1) instead of DATA CST/1/.

8.3.3 Arrays

If an array name is a dummy argument, a zero lower bound results in faster programs than a non-zero bound:

Example:

SUBROUTINE S(T)
DIMENSION T (0:999)
instead of
SUBROUTINE S(T)
DIMENSION T (999)

If an array name is the first element of a COMMON block, it results in faster programs if it has a zero lower bound.

Example:

COMMON /C/T (0:999),...
or
COMMON /C/...,T(999),...
instead of
COMMON /C/T(999),...
8.3.4 Alignment of Data

DATA of 1, 2, 4, 8 and 16 bytes result in faster programs if they are allocated respectively on 1, 2, 4, 8 and 16 bytes segment offset. This is done by the compiler except for data specified in EQUIVALENCE or COMMON statements.

A COMMON is allocated on a 16 bytes memory boundary. The compiler allocates its elements in the order of their specifications, except for EQUIVALENCE statements, without any gap. So you may specify COMMON elements in an order, or with fillers, such that they are on a proper memory boundary.

8.3.5 COMMON Blocks, DSEGMAX Parameter

If a program unit or package unit leads to at most two data segments in addition to the hardware stack, the compiler fixes base registers to access their data. This avoids the extra load of base registers, and results in faster programs.

For details about segment generation by the compiler, refer to Section 7: Segmentation and Addressing. The conditions that are sufficient to have at most two segments of data are:

a) No use of the DSEGMAX parameter of the FOR77 JCL statement, and:

b) For a program unit not encapsulated into a package unit:
   have none, or only one COMMON block.

c) For a package unit:
   have no EXTCOMMON statement, or only one COMMON block specified in an EXTCOMMON statement.
8.4 EXECUTABLE STATEMENTS

8.4.1 Label

Optimization is reduced by a label in columns 1 - 5. Therefore sequences of programs without labels result in faster execution than if there were labels.

The only exception, where the use of labels is faster, is for computed GOTO statements.

*Example:*

GOTO (10, 20, 30, 40) I
...
is faster than
IF (I.EQ.1) THEN
ELSE IF (I.EQ.2) THEN
...
ELSE IF (I.EQ.3) THEN
...
ELSE IF (I.EQ.4) THEN
...
ELSE
...
ENDIF

8.4.2 Array References

For an array with several dimensions, faster programs result on a DPS7/X0 or DPS7/X17-X27 (with a cache memory), if the elements are successively accessed in the order of the array storage sequence.

*Example:*

DIMENSION T (100, 100)
DO 10 I=1, 100
DO 20 J=1, 100
... T (J,I)...

and not:

... T (I,J)...

Efficiency Techniques
8.4.3 Intrinsic Functions

The table in Appendix A gives information about the implementation of intrinsic functions. Increased program speed corresponds to the following order of implementation:

- call to a mathematical package
- in-line generation
- firmware
- character and bit manipulations.

Firmware intrinsic functions for character or bit manipulation result in especially fast programs.

For example CHAR, IVERIF, IBSET.

8.4.4 CALL

A call to a subroutine or function of the same PACKAGE unit results in faster programs than a CALL to a subroutine or function compiled separately.

Inside a PACKAGE unit, a call to a subroutine or function which appears in the lines before the call results in faster programs than if the subroutine or function called appears in the lines after the call.
8.5 INPUT/OUTPUT

8.5.1 Fast READ/WRITE

READ/WRITE statements involve fast run-time routines, resulting in faster programs (in CPU time) if the following conditions are satisfied:

- If it is a sequential READ (or WRITE), the previous input/output statement on the same unit is a sequential READ (or WRITE) with the same characteristics: formatted/unformatted.
- The unit value is less than 200.
- There is not a BYTE = specifier.
- The file is not opened with a SPANNING = 'YES' specifier. (No split records.)
- The program unit is not compiled with one of the following parameters of the FOR77 JCL statement:
  - SUBCK
  - LEVEL=GCOS1E
  - LEVEL=SIRIS8
- The READ/WRITE does not result in an error condition, or end of file condition (IOSTAT=0).
- The file connected to the unit is not a dummy file, nor the console of the interactive user, nor the operator console.

8.5.2 OPEN Statement

The OPEN statement, or the first READ/WRITE for a preconnected file takes far more CPU time than a current READ/WRITE. A great number of READ/WRITEs must be executed so that the connection time becomes negligible, for instance 10,000 fast READ/WRITEs.

The CPU time added on the first READ/WRITE on a preconnected file is less than the time of an OPEN statement. To reread a file it is faster to perform a REWIND than a CLOSE followed by an OPEN.
8.5.3 Characteristics of READ/WRITE Statements

In addition to the conditions of paragraph 1 for fast READ/WRITE you should be aware that the CPU time needed for a READ/WRITE depends on the following factors.

- Unformatted records have a faster READ/WRITE time than formatted records.
- An input/output list without DO implied list is faster than with DO implied list.
- An input/output list with one element is faster than with several elements.
- A non-variable format is faster than a variable format.
- A format identifier that is the label of a format statement is faster than a character array name or character expression.
9. Files

This section contains an overview of file handling by the DPS 7 FORTRAN 77.

9.1 FILE NAMES

Each file can have three different names: a unit name, an internal file name, and an external name.

9.1.1 FORTRAN Unit Name

This is specified in the input/output statements by the "UNIT=specifier" parameter. It is generally an integer value, accessible by all the FORTRAN program units of the executable program (i.e. the step).

9.1.2 Internal File Name

The internal file name (or IFN) is specified in the OPEN or INQUIRE statements by the "FILE=specifier". The specified character value must begin with "IFN=". The IFN specified in this character string is accessible by all the procedures of the step, whether or not they are FORTRAN procedures. It is also accessible by certain JCL statements, such as "ASSIGN".
9.1.3 External file name

The external file name (or EFN) is specified in the OPEN or INQUIRE FORTRAN statements. The EFN specified in this character string makes the file known to the system. It is recorded in the file label and possibly in a catalog. It can be accessed, via several languages, from any job having the access rights. In particular, it can be accessed from JCL statements like ASSIGN.

Examples of the use of these names are given in the FORTRAN 77 Reference Manual in Chapter 12, with both the FORTRAN 77 program and the JCL.
9.2 FILE NAMING FACILITIES

DPS 7 FORTRAN 77 allows several possibilities for naming files, depending on the environment of the FORTRAN 77 program.

9.2.1 Printing

Use PRINT or WRITE with UNIT=*  

*Example:*

```fortran
PROGRAM P
WRITE (*,*)'THIS TEXT'
END
```

JCL:  
```bash
$STEP P;
$ENDSTEP;
```

9.2.2 Unmixed FORTRAN Units

For a step made up of FORTRAN units only, preconnection is made by ASSIGN statements in the JCL. The OPEN statement is not necessary, because the IFN has the same character representation as the FORTRAN unit name. For instance, if the unit name is "10" and the IFN is the character string "10" in the ASSIGN statement, both file references will be assigned to the same EFN.

*Example:*

```fortran
PROGRAM P
WRITE(UNIT=10,*)'THIS TEXT'
END
```

JCL:  
```bash
$STEP P;
$ASSIGN 10,MYEFN,DVC=MS/M452,MD=K104;
$ENDSTEP;
```
9.2.3 Cataloged File at OPEN Time

In the case of a step made up of only FORTRAN program units, where connection to a cataloged file is made by the OPEN, the IFN is disregarded and the EFN is used. For instance OPEN(UNIT=10,FILE='MYEFN') where MYEFN is the cataloged EFN.

Example:

```fortran
PROGRAM P
OPEN(UNIT=10,FILE='MYEFN')
WRITE(UNIT=10,*)'THIS TEXT'
END
```

JCL:  $STEP P;
      $ENDSTEP;

9.2.4 Cataloged File at ASSIGN Time

This case is the same as the previous one except that parameterization is required at JCL level by means of an ASSIGN statement.

Example:

```fortran
PROGRAM P
OPEN(UNIT=10)
WRITE(UNIT=10,*)'THIS TEXT'
END
```

JCL:  $STEP P;
      ASSIGN 10,MYEFN,FILESTAT=CAT;
      $ENDSTEP;

Note that specifying an IFN different from a unit name is in general useless in DPS 7 FORTRAN 77. An exception to this rule is when there are constraints resulting from a language other than FORTRAN, as in the example in the next paragraph.
9.2.5 **Mixed FORTRAN and COBOL**

In this case the step is made up of mixed FORTRAN and COBOL program units. FORTRAN accesses the file via the unit name, which is connected to an IFN by an OPEN statement. COBOL accesses the file via a FILENAME, connected to an IFN by a SELECT clause.

**Example:**

FORTRAN: `PROGRAM P
OPEN(UNIT=10,FILE='IFN=HCOBFOR1')
WRITE(UNIT=10,*)'THIS TEXT'
CALL COBOL
END

COBOL: `SELECT FILENAME ASSIGN TO HCOBFOR1.

JCL: `$STEP P;
ASSIGN HCOBFOR1,MYEFN,FILESTAT=CAT;
$ENDSTEP;`
9.3 FILE USE IN JOR

The information provided by the Job Occurrence Report about file use is given through the internal file name (IFN).

9.3.1 Types of IFN

9.3.1.1 Numeric

In the case of a numeric IFN (ex: 23) the file is preconnected by the JCL ASSIGN statement.

Example:

JCL: ASSIGN 23 EFN;
FORTRAN: WRITE(23,99) A

9.3.1.2 Unit Record Device IFN

In this case the IFN has the form H_RD,H_PR,H_CP etc. See FORTRAN 77 Reference Manual Chapter 12.

9.3.1.3 Numeric Preceded by a Semicolon

In this case the file has been connected to the IFN (ex::24) by a FORTRAN OPEN statement.

OPEN (UNIT=24,FILE='EFN')

9.3.1.4 Eight Star IFN

In this case (ex: ********) the file has been referenced by a FORTRAN INQUIRE statement.

9.3.1.5 Others

In other cases the file has been connected by a FORTRAN OPEN statement.

OPEN (UNIT=31,FILE='IFN=HCOBFOR1')
OPEN (UNIT=32,FILE='IFN=MYOWNIFN')
9.4 DATA MANAGEMENT OVERRIDING RULES

The source program defines some of the basic file characteristics, using the OPEN statement. However, a number of file parameters (such as file organization, blocksize and device class) can be specified at different places on the system and at various stages in file creation. Table 9-1 shows the applicability of parameters specified at certain points.

9.4.1 Specification and Applicability of File Characteristics

Table 9-1. Applicability of File Characteristics

<table>
<thead>
<tr>
<th>WHERE SPECIFIED</th>
<th>WHERE APPLICABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>At system generation</td>
<td>Throughout the job</td>
</tr>
<tr>
<td>In source program</td>
<td>Compilation time</td>
</tr>
<tr>
<td>In JCL (ASSIGN and DEFINE)</td>
<td>Job translation time</td>
</tr>
<tr>
<td>In file label</td>
<td>File OPEN time</td>
</tr>
</tbody>
</table>

Data Management overriding rules define the final values to be used for file processing in case of doubt; for example if a parameter has been assigned several conflicting values (or none at all). Data Management sends a message to the user: FATAL if a choice cannot be made, WARNING if it overrides some other potential parameter value.

9.4.2 Summary of Overriding Rules

- Basic file parameters that are specified in the FORTRAN program override (and/or complement) any system values that apply by default.

- File parameters specified by the JCL override those specified in the FORTRAN program.

- File parameters specified in the file label (if the file exists at OPEN time override those specified in the JCL).

9.4.3 Checks at OPEN Time

Data Management performs additional checks on a file when it is opened by a FORTRAN program.

9.4.3.1 Record Length

The record length must be the same as that declared in the OPEN statement in the FORTRAN program.

9.4.3.2 Permitted File Organizations

Table 9-2. Permitted File Organizations

<table>
<thead>
<tr>
<th>ACCESS MODE IN FORTRAN PROGRAM</th>
<th>UFAS</th>
<th>TYPE OF FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td>SEQUENTIAL</td>
<td></td>
<td>Allowed.</td>
</tr>
</tbody>
</table>

For FB the records can be read and cannot be created in this mode.
9.5 OPTIONAL FILES

A file is said to be optional in the following cases:

- Its name, given in the OPEN statement is a character string whose value is 'DUMMY'.
- It is assigned by a JCL ASSIGN statement which includes a DUMMY or OPTIONAL parameter.

The DUMMY and OPTIONAL parameters are used in the following way:

9.5.1 DUMMY Parameter

An ASSIGN statement with a DUMMY parameter specifies that the file is absent and all references to it should be ignored.

ASSIGN IFN,DUMMY;

9.5.2 OPTIONAL Parameter

An ASSIGN statement with an OPTIONAL statement requests operator intervention.

ASSIGN IFN,EFN,DEVCLASS...,MEDIA,OPTIONAL;

At step execution the system searches the volume named in MEDIA for this file. If the volume is mounted and the file exists, it is processed normally. If the volume is absent a MOUNT request is sent to the operator. If the operator refuses the MOUNT request (CRMS...) or if the file does not exist on the volume mounted, then the file is considered as DUMMY.

9.5.3 I/O Restrictions

Dummy files can be used on output only. Using a READ statement causes an error condition. A WRITE statement is allowed but it does not write any record.
9.6 CLOSE WITH STATUS='DELETE'

If STATUS='DELETE' is used to close a file the file is deallocated. Sometimes this may not be possible: for example, with a tape file.

9.7 THE POOL JCL STATEMENT

If a program accesses several files that are on different volumes, the normal procedure is to allocate at step initiation one device per volume. However, only one device may be required for several files, even on different volumes, providing that these files are never accessed at the same time. For example:

```
FORTRAN77:
  READ (10) A
  ...  
  CLOSE (10)
  READ (11) B
  ...  
  CLOSE (11)
```

In this example, processing on the file preconnected to unit 10 is completely finished before processing begins on that preconnected to unit 11, so it is possible to use the same device for both files even if they are on different volumes. This is done by using the POOL JCL statement in conjunction with the POOL parameter in the ASSIGN statement.

```
POOL 1*MS/M452;
ASSIGN 10,MAX.Z,POOL,FIRST...
ASSIGN 11,EMY.I,POOL,NEXT...
```

Thus, a device pool of one MSU452 will be reserved for these files. More than one device can be requested in the POOL statement to facilitate operation; for example, where three or more volumes are processed, two devices can be requested so that, while one device is being used by the program the other is loaded by the operator with the next volume to be processed.

9.8 MULTIVOLUME FILES

If a file is too big to fit on a single disk or tape volume, it can be stored on more than one volume. Such a file is called a multivolume file.

The following paragraphs are applicable to sequential multivolume files only. Each part of a multivolume file is known as a "physical unit". For disk volumes containing other files also, only part of the disk will be occupied by the "physical unit". Space is allocated by the PREALLOC utility (see Data Management Utilities Manual). Only one ASSIGN statement is needed, the volume names listed in the MEDIA parameter or recorded in the files' catalog entry. Volume boundaries are invisible to a FORTRAN 77 program; the system switches automatically to the next volume. For more information on multivolume files see the JCL User Guide.
9.9 MULTIPLE FILE TAPE VOLUMES

Several files on the same tape may be accessed by a FORTRAN 77 program, though only one may be open at the same time. There must be an ASSIGN statement for each file. There may also be a POOL statement to allocate a single device for the magnetic tape. The file sequence number (FSN) parameter must be used in each ASSIGN statement to specify the position numbers of the files. The DEVCLASS and MEDIA parameters must be the same for all files.

Example:

FORTRAN77:

```
WRITE (10) A
CLOSE (10)
WRITE (11) B
CLOSE (11)
WRITE (12) C
CLOSE (12)
```

JCL:

```
POOL MT/T9;
ASSIGN 10,E,FILEA,DEVCLASS=MT/T9,MEDIA=TAPEA,FSN=1,POOL,FIRST;
ASSIGN 11,E,FILEB,DEVCLASS=MT/T9,MEDIA=TAPEA,FSN=2,POOL,NEXT;
ASSIGN 12,E,FILEC,DEVCLASS=MT/T9,MEDIA=TAPEA,FSN=3,POOL,NEXT;
```

9.10 FILE CONCATENATION

File concatenation provides a means of accessing several UFAS or BFAS sequential tape files as if they were one logical sequential file. This process should be distinguished from the concepts of multivolume files and multiple file tape volumes that have been discussed above. File concatenation is used with cassette files.

File concatenation is specified by a sequential string of ASSIGN statements, omitting the IFN on all but the first. A POOL statement should be used to allocate one or two devices to the concatenated files.

```
POOL 2*MT/T9;
ASSIGN,IFN,MY.FILE1,DEVCLASS=MT/T9,MEDIA=A1,POOL,FIRST;
ASSIGN,MY.FILE2,DEVCLASS=MT/T9,MEDIA=A2,POOL,NEXT;
ASSIGN,MY.FILE3,DEVCLASS=MT/T9,MEDIA=A3,POOL,NEXT;
```

Here the three tape files MY.FILE1, MY.FILE2 and MY. are regarded as a single sequential file. Concatenated files must have the same device class and device attributes; and the same RECFORM, BLKSIZE and RECSIZE parameters. File concatenation can be used for a multivolume file or a multi-logical unit file.
9.11 FILE ACCESS SYSTEMS

The following file access systems are available:

UFAS - Universal File Access System
BFAS - Basic File Access System (only sequential)

UFAS and BFAS can be used interchangeably for magnetic tape. Some notes on these two access systems are given below. For further information see:

BFAS User Guide

9.11.1 UFAS

UFAS is the primary file access system. It offers a wide range of facilities and considerable device independence.

9.11.2 BFAS

BFAS, as an alternative to UFAS, has less device independence but better performance for certain types of application. Only sequential BFAS files are supported.

9.11.3 Specifying the Access System

The access system to be used can be specified in the OPEN statement by the FILEFORM parameter. The following values may be used:

FILEFORM='UFAS'
FILEFORM='BFAS'

The FILEFORM parameter is significant only if the file does not already exist (STATUS='NEW').
9.12 CREATING FILES

Files can be created by the FORTRAN 77 OPEN statement or the JCL utilities PREALLOC and ALLOCATE.

9.12.1 PREALLOC Statement

PREALLOC is a JCL utility which prepares space on a disk for a permanent UFAS or BFAS file. See the Data Management Utilities User's Guide for how to calculate the number of tracks or cylinders required.

$JOB JPB, USER=FORUSE, PROJECT=PUBS;
PREALLOC FOR.DATA, DEVCLASS=MS/452,
GLOBAL=((MEDIA=XZY123, SIZE=8),
UFAS=(RELATIVE=(RECSIZE=80, CISIZE=1600)),
FILESTAT=CAT;
$ENDJOB;

9.12.2 ALLOCATE Statement

The ALLOCATE statement prepares space on a disk for UFAS and BFAS permanent uncataloged or temporary files (all organizations except indexed sequential). The ALLOCATE statement must not be used for UFAS files with direct access.

9.13 TERMINAL READ/WRITE

When a program is executed from an interactive terminal the statements READ (*,...) and WRITE (*,...) refer to that terminal. So do READ/WRITE statements specifying a unit connected by means of an OPEN(u,FILE="SPECIAL=IOF") statement. If a READ from a terminal is expected, the prompt I: is printed. Immediately after this prompt, on the same line, data can be entered, followed by a carriage return. Entry of $EOS causes an 'end of file' condition. If you press the "break" key before carriage return, you are asked:

REREAD, STOP OR CONTINUE

If you answer REREAD, a new prompt I: is printed. If you answer STOP, the program execution terminates. If you answer CONTINUE, the program execution continues. If no terminal READ is expected, the interactive session can be terminated by using the "break" key. This causes the prompt "???" to be output. You can stop the program by typing "QUIT", after which the prompt "S:" is output. If you now type "carriage return" the session continues as if the "break" key had not been pressed.
9.14 ERROR HANDLING

If a system procedure returns an abnormal return code to the FORTRAN 77 program as the result of an I/O operation, the step is abnormally terminated unless an "IOSTAT=ERR" or "ERR=" is present in the control list of the input/output statement. You can diagnose the error using the value returned in the "IOSTAT=value" specifier. (These values are listed in Appendix D of the FORTRAN 77 Reference Manual).

9.15 RETURN CODE

The information provided by the "IOSTAT=" specifier is normally sufficient to diagnose most I/O errors. However, the full return code generated by Data Management can be obtained and analysed by the FORTRAN 77 program by means of the intrinsic functions KTEST and ITEST with the argument 'SYSTRIC'.
10. Standard Record Formats

10.1 THE FOUR STANDARD RECORD FORMATS

Data Management recognizes four standard record formats. These formats can be used in magnetic tape or disk file. They are:

10.1.1 System Standard Format (SSF)

In this format each record is made up of an 8-byte header followed by normal data. This header makes the file or subfile device-independent; a file or subfile in SSF can be routed from the disk or tape to any kind of I/O device. This format provides the Stream Reader, compilers, LIBMAINT, and the Output Writer with a standard method of handling their input/output data. This is the format normally used in data subfiles passed between FORTRAN 77 programs.

10.1.2 Standard Access Record Format (SARF)

In this format each record is composed exclusively of normal data without any special heading information. This is the format normally used in data files passed between FORTRAN 77 programs.

10.1.3 American Standards Association Format (ASA)

In this format each record consists of a 1-byte header followed by normal data. The header contains a subset of the information in an SSF header. However, ASA files are not device-independent; they may contain only data to be printed. The main use of ASA files is to assure compatibility with other systems; they should not normally be used for print files processed entirely within the system. The programmer is responsible for the contents of the first character of the record, which contains the skip information.
10.1.4 **Device Oriented Format (DOF)**

In this format each record consists of an 8-byte header followed by normal data. The header contains device oriented control information in the form used by the various unit record devices. This format is used only by "Program Mode" PM100 and PM200 systems.

ASA and DOF are rarely used by the FORTRAN 77 programmer and is not discussed further in this section, which is henceforth devoted to SSF and SARF.
10.2 FORTRAN 77 MANIPULATIONS ON SSF FILES

10.2.1 SSF Header Format

The SSF header takes up eight bytes and consists of the following components:

10.2.1.1 Record Type

This indicates if it is a control record or a normal data record. Control records are added to the file by the system or the Report Writer (if used) to control the handling of the file and the production of page headings.

10.2.1.2 Header Type

If the record is a control record this field specifies the type of control record.

10.2.1.3 Truncation Value

This specifies the number of space characters that have been truncated at the rightmost end of the record.

10.2.1.4 Line Number

This contains the sequence number of the record within the file. It may be derived from the data cards used when the file is read into the system. It may also be generated by the RENUMBER command (LIBMAINT) or the NUMBER option of the MOVE command (LIBMAINT).

10.2.1.5 Form Control

This specifies the paper movement required when printing the record.
10.2.2 Effects of Header Type

If the first record in an SSF file is a control record with header type 101 then the file is handled by the system as if all the records in the file were in SSF format, and the language type specified at file creation is indicated in the header. If the first record of a file does not have this format then the file is handled as if all its records are in SARF format.

10.2.3 Creation and Updating by FORTRAN 77

A file created by FORTRAN 77 is in SSF format if it is the subfile of a library or a print file; it is in SARF format in all other cases. If an SSF or SARF file is updated by FORTRAN 77 its record format remains unchanged.

10.2.4 SSF and the Utilities

This paragraph describes the relationship between SSF and the following: the Stream Reader, LIBMAINT, the FORTRAN 77 compiler, FORTRAN 77 programs, and the Output Writer.

An SSF file can be created from cards contained in an input enclosure. If TYPE=FOR or TYPE=DATASSF is specified in the $INPUT statement the Stream Reader creates a temporary subfile in the SYS.IN system file. This is known as a standard SYSIN subfile and exists only for the duration of a job. Cards from the input stream are read into this subfile as a series of SSF records. Subsequently, this subfile can be read by any job, step or utility. For example, it can be moved to a user library by LIBMAINT.

Example 10-1:

$JOB MYJOB,USER=ME;
LIBALLOC SL,(SSF.LIB,SIZE=2),MEMBERS=13;
LIBMAINT SL,LIB=SSF.LIB,COMFILE=*SSFENC;
$INPUT SSFENC,TYPE=DATASSF;
MOVE COMFILE:SSFMEMB,TYPE=FOR;
...
//EOD
$ENDINPUT;
$ENDJOB;

In this example LIBALLOC creates a resident source library called SSF.LIB with a size of two cylinders. Then LIBMAINT creates a new subfile of SSF.LIB, called SSFMEMB, and transfers card images from the input enclosure to the subfile SSFMEMB ("MOVE" command).
Standard Record Formats

A FORTRAN 77 user program can read data from an SSF library member or from the input enclosure (in the standard SYSIN subfile). The LIBMAINT commands "EDIT" and "UPDATE" can alter the contents of SSF library members. These commands can use line numbers held in the SSF headers.

If the SSF library member contains a FORTRAN 77 program it can be processed by the FORTRAN 77 compiler. If the SSF library member contains JCL it can be used in an INVOKE, EXECUTE or RUN JCL statement. (See the JCL User's Guide).

10.2.5 Reading SSF Files

Any file input to a FORTRAN 77 program may be in SSF or SARF format. The presence of a type 101 control record specifies the SSF format. Data management checks for the existence of this record and processes the file accordingly.

When an SSF file is read by a FORTRAN 77 program the header will be stripped from the first record before it is passed to the program. Also, no control records will be passed to the program. If the SSF header contains a non-zero truncation value, i.e. when blanks have been truncated at the end of a record, the blanks are restored when the record is read. In such cases the record length includes the truncation value. Truncation values are generated when, for example, the file has been created by LIBMAINT with a language type of FOR or DAT. In the following example, a library member is created by LIBMAINT and then read by FORTRAN 77 in interactive mode. Note that this member has variable length records, but is read by a fixed size FORTRAN 77 format. This leads to run time errors if the PAD='YES' specifier is not used in the OPEN statement. (Another method, without using PAD='YES', is to use a PROCL specifier in the READ statement).

Example 10-2:

S:     LIBMAINT SL, LIB=(MYLIB.SLLIB,DVC=MS/M452,MD=K103);
C:     EDIT;
R:     A
R:     C THIS FORTRAN77 PROGRAM READS ITSELF
I:               PROGRAM MYSUB
I:               CHARACTER *130 C(100)
I:               OPEN (UNIT=10,PAD='YES',FILE=
               'MYLIB..MYSUB:K103:MS/M452')
I:               DO 5 I=1,100
I:      5        READ (UNIT=10,FMT='(A130)',END=50) C
I:      50       STOP
I:               END
I:     [F
R:     Z MYSUB
R:     Q
C:     QUIT;
S:     FOR77 SOURCE=MYSUB,INLIB=(MYLIB,DVC=MS/M452,MD=K103);
S:     LINKER MYSUB;
S:     STEP MYSUB,TEMP;
S:     ENDDATE;
10.2.6 Writing SSF Files

If an existing SSF file or subfile is rewritten by a FORTRAN 77 program, its language type remains unchanged. If a new subfile is created by a FORTRAN 77 program, its language type is DAT except if the new subfile is a print file, where the language type is EDT (see the FORTRAN 77 Reference Manual, Section 12).
10.3 FORTRAN 77 MANIPULATIONS ON SARF FILES

SARF records have no special header but are composed exclusively of user data. This format is normally used for data files passed between FORTRAN 77 programs, but may also be used by the Stream Reader, LIBMAINT, the compilers, and the Output Writer.

10.3.1 SARF and the Utilities

A SARF subfile can be created from cards contained in an input enclosure. Normal practice should be to omit the TYPE parameter from the $INPUT statement (equivalent to TYPE=DATA). If this is done, the Stream Reader creates a temporary subfile in the system file SYS.IN and the cards are read into this subfile as a series of SARF records. That is, a standard SYSIN subfile is created for the duration of the job.

The standard SYSIN subfile may then be read by the LIBMAINT utility and may be moved to a user library. The following example illustrates this sequence:

```
$JOB...
 LIBALLOC SL,(SARF.LIB,SIZE=2),MEMBERS=13;
 LIBMAINT SL,LIB=SARF.LIB,COMFILE=*SARFENC;
$INPUT
     MOVE COMFILE:SARFMEMB,OUTFORM=SARF;
.
.$ENDINPUT;
$ENDJOB;
```

In this example, a resident source library SARF.LIB is set up by LIBALLOC with a size of two cylinders. The card images from the input enclosure are moved from the standard SYSIN subfile to the library SARF.LIB by the MOVE command of LIBMAINT. A new member SARFMEMB is created in the library SARF.LIB to contain the data. The EDIT and UPDATE commands of LIBMAINT cannot be used to alter the contents of SARF library members. However, if the SARF library member contains a FORTRAN 77 program, it can be processed by the FORTRAN 77 compiler. The use of the compiler is covered in Section 2. If the SARF library member contains JCL it can be used in an INVOKE, EXECUTE or RUN JCL statement (see JCL User Guide).
10.3.2 Reading SARF Files in FORTRAN 77 Programs

As mentioned previously, a FORTRAN 77 program can read any file in SSF or SARF format. SSF files are recognized by the presence of a control record of type 101.

10.3.3 Writing SARF Files in FORTRAN 77 Programs

When an existing SARF file is rewritten by a FORTRAN 77 program it remains in SARF format. When a new file (not a subfile) is created by a FORTRAN 77 program, it is created in SARF format.
10.4 SSF AND SARF WITH THE OUTPUT WRITER

The Output Writer can print or punch any SSF or SARF file. It is called by the JCL statements SYSOUT and WRITER. The Output Writer is normally used to output print or punch files produced by user programs. However, it can also print or punch files that have not been specially formatted for output, such as a library member containing a FORTRAN 77 program or a normal disk or tape file containing data.

The use of the Output Writer for print and punch files is discussed in Section 11.
11. Using Unit Record Files

This section describes the way in which the following unit record files are used:

- Print files.
- Punched card files.
- Diskette files.

11.1 PRINTING

Printing can be done in the following ways:

- Data can be stored in a SYSOUT file for printing later by the Output Writer.
- Data can be sent direct to the printer.

Print data should be output to a SYSOUT file. The direct use of printers slows down program execution and reduces the throughput of the printer.

11.1.1 Using SYSOUT Files for Printing

The following types of SYSOUT file can be used to store data to be printed:

- Standard SYSOUT subfile.
- Permanent SYSOUT file.

The standard SYSOUT file (SYS.OUT) is a system file. The SYSOUT file is created at system generation and is located on a resident disk. For each step, one or more subfiles is assigned for each unit record output file defined in the step. During execution of each step, data to be printed or punched is sent to subfiles of the standard SYSOUT file. No ASSIGN statement need be used for a standard SYSOUT subfile. Standard SYSOUT subfiles exist until the data in them has been printed or punched. When output processing is finished, the subfiles are automatically deleted.
A permanent SYSOUT file is a sequential disk or tape file or source library member which is not automatically deleted after Output Writer activity, or a permanent magnetic tape file (useful for large volumes of output). A permanent SYSOUT file must be assigned by the user.

SYSOUT files are normally written in a format known as "edited SYSOUT". This is done automatically if certain conditions, described below, are met. This has the following effect on output data:

- Records are formatted for the output device.
- The page is formatted (page headers, numbers, etc).
- Trailing blanks are suppressed.

An edited SYSOUT file cannot be handled as a normal SSF, SARF, or ASA file.

If the record size of the SYSOUT file is less than 600 bytes (specified when the file is allocated using the PREALLOC statement or later in the DEFINE statement) it is written as an SSF, SARF, or ASA file. If the record size is greater than or equal to 600 bytes the file is in edited SYSOUT format. SSF, SARF, or ASA files which are to be printed are edited subsequently by the Output Writer. Note that the use of a record size of 600 does not imply storage inefficiency, because the RECFORM will be VB (variable). However, editing SSF, SARF, or ASA files during printing, rather than when the file is being written, is inefficient and should be avoided. See the JCL User Guide for more information about SYSOUT files and the Output Writer.

There are certain situations in which a SYSOUT file should not be in edited SYSOUT format. If the SYSOUT file is to be processed before printing (e.g., by another FORTRAN 77 program or by the LIBMAINT utility), it should not be written in edited SYSOUT format. Note that a standard SYSOUT file is always in edited SYSOUT format.

Standard SYSOUT files are always printed automatically by the Output Writer unless the HOLDOUT parameter is specified in the $JOB statement. These ‘held’ outputs are printed when released by the operator (RO command).

Permanent files are printed automatically only if there is a SYSOUT JCL statement in the job step JCL and if this statement does not contain the WHEN = DEFER parameter. In all other cases, the permanent SYSOUT file should be printed in a separate job step by using the WRITER JCL statement.

All the SYSOUT files written by a FORTRAN 77 program have an SSF record format or an edited SYSOUT format. SSF format includes an 8-byte header in each record that enables form control information to be stored for each print line. The first character of the record encoded by the FORTRAN 77 program becomes the eighth character of the SSF header. This is also true for edited SYSOUT files.
11.1.2 Printing Directly

When the printer is used directly, an ASSIGN JCL statement must be present at execution time which links the internal-file-name used for the printer to the output device. For example:

**FORTRAN:**

```
CHARACTER *133 LINE
WRITE(20,99) LINE
99 FORMAT (A133)
```

**JCL:**

```
ASSIGN 20 , DEVCLASS = PR, MEDIA = I20001;
DEFINE 20 , MARGIN = 10;
```

The use of the DEFINE statement is optional. See the JCL Reference Manual for details of the relevant DEFINE parameters.

11.1.3 Form Control

A "vertical format tape" is a punched tape loop often used in printers to control vertical paper movement. Since DPS 7 printers do not use a vertical format tape, vertical paper movement is controlled by a software-simulated vertical format unit (VFU). This VFU works in the same way as a standard 12-channel vertical format tape, with a limitation of 20 stop levels per form, shared among the 12 channels.

VFUs are stored in a system file called SYUSRCEINIT. The user can add new VFUs to this file or modify existing ones using the utility URINIT. This process is described in the Unit Record Devices User's Guide. Also stored in SYS.URCINIT are the form height, margin, head of form, full form 1, and printing density. All this information is associated with a form number. This form number can be specified in the MEDIA parameter in the ASSIGN, OUTVAL, SYSOUT, and WRITER JCL statements in order to ensure that the correct VFU, form height, etc. is used when the file is printed. See the JCL Reference Manual for details of the MEDIA parameter in ASSIGN, OUTVAL, SYSOUT, and WRITER.

The VFU, form height, margin, head of form, full form 1, and printing density stored in SYS.URCINIT can be overridden at execution time by parameters specified in a DEFINE JCL statement. See the JCL Reference Manual for details.
11.2 CARDS

11.2.1 Reading Cards Directly

To read cards directly from the card reader, there must be an ASSIGN statement in the execution JCL that links the internal-file-name used for the card reader to the input device.

**Example:**

FORTRAN:

```
CHARACTER *80 CARD
READ (30,99) CARD
99 FORMAT(A80)
```

JCL:

```
ASSIGN 30 , DEVCLASS = CD/R, MEDIA = INDECK;
DEFINE 30 , OFFSET;
```

**CONSOLE MESSAGE:**

```
* hh.mm MOUNT INDECK FOR ron
```

where:

- hh.mm is the current time in hours and minutes.
- ron is the run occurrence number.

The use of the DEFINE statement is optional. See the *JCL Reference Manual* for details of the relevant DEFINE parameters.

The name specified in the MEDIA parameter is displayed on the operator's console at step initiation. This name should also be written on the card deck so that the operator can see clearly which card deck is to be used. The card deck must not be part of a job stream. It must be a separate deck and the last card must be an $EOS statement followed by at least one blank card. The card deck should be mounted in the card reader and the card reader should be switched to "ready".

11.2.2 Punching Cards

Punched cards can be output in the following ways:

- to a SYSOUT file;
- directly to the card punch.

Cards should normally be output to a SYSOUT file. Direct use of the card punch slows down program execution and reduces the throughput of the card punch. In either case, serious consideration should be given to use of a more compact and less fragile storage medium.
11.2.2.1 Using Sysout Files for Cards

Both standard SYSOUT and permanent SYSOUT files may be used to store data to be punched. They have the same characteristics as the printer SYSOUT files. The SYSOUT JCL statement is mandatory, otherwise the file will be printed instead of punched. This statement should specify a card punch device class.

Example:

SYSOUT H_CP, DEVCLASS = CD/P, MEDIA = PUNCHOUT;

The WRITER JCL statement, used to punch the files must also specify a card-punch device-class. For example:

WRITER C.H_CP, DEVCLASS = CD/P, MEDIA = PUNCHOUT;

11.2.2.2 Punching Cards Directly

To punch cards directly on the card punch, there must be an ASSIGN JCL statement in the execution JCL that links the internal-file-name used for the card punch to the output device. For example:

FORTRAN:

```
CHARACTER*80 CARD
WRITE(40,99) card
```

99 FORMAT(A80)

JCL:

```
ASSIGN 40 , DEVCLASS = CD/P, MEDIA = OUTDECK;
DEFINE 40 ,OFFSET;

CONSOLE MESSAGE:
  * hh.mm MOUNT OUTDECK FOR ron
```

where:

hh.mm is the current time in hours and minutes.

Ron is the run occurrence number.

The use of the DEFINE statement is optional. See the JCL Reference Manual for details of the relevant DEFINE parameters. The name specified in the MEDIA parameter is displayed on the operator’s console at step initiation. A deck of blank cards should be mounted in the card punch and the card punch should be switched to “ready.”
11.3  PAPER TAPE

**NOTE:** Paper tape is not supported for GCOS V3 and upwards.

The Input Reader and Output Writer do not handle paper tape so each paper tape file must be assigned directly to a physical reader or punch.

The format of the ASSIGN statement to be used is as follows:

```
ASSIGN internal-file-name, external-file-name,
DEVCLASS = PT/R/NHOLE
           SHOLE
           PG/P/NHOLE ; MEDIA = volume-name ;
```

When reading a paper tape, PT/R is used. When punching a paper tape, PT/P is used. The mandatory MEDIA parameter associates with the file being processed a specific initialization table which contains all the information about the characteristics of the media being processed: paper tape code and its translation from/to EBCDIC, usage of record marks, and inter-record gaps (see the *Unit Record Devices User's Guide*).
11.4 DISKETTES

In a FORTRAN 77 program diskette files are handled as UFAS or BFAS sequential files. See the Diskette and Stream Reader User's Guide for details.

11.5 PAUSE AND STOP LITERAL

The PAUSE literal statement suspends execution of the program until the operator enters a value which enables the program to continue. The STOP literal statement advises the operator that the program execution is terminated. For both cases, the literal is displayed

- at the main console in batch mode
- at the user console in IOF.

When this occurs the following message is displayed on the main operator's console:

nn/hh:mm ron literal [PLEASE TYPE:RESUME FORTRAN]

where:

nn is a message number which the operator must enter when replying to this message.

hh/mm is the time at which the message was displayed

ron is the run-occurrence-number

In order to restart the program the operator must enter the message number, one space, RESUME FORTRAN, and carriage return.
12. Differences Between GCOS FORTRAN and GCOS FORTRAN 77

This section deals only with the differences at execution time.

The source language differences are detailed in Appendix B of the FORTRAN 77 Reference Manual.

These differences will affect you if you have to change from Release 1E FORTRAN to GCOS 7 Release Vn FORTRAN 77. Remember that both the FORTRAN and the FORTRAN 77 compilers are available under GCOS 7 Release Vn to avoid any difficulties caused by the differences between Release 1E FORTRAN and GCOS 7 Release Vn FORTRAN. However the FORTRAN compiler is not available under GCOS 7 Release V3.

Some of the differences at execution time are removed by the use of the LEVEL=GCOS1E parameter in the FOR77 JCL statement. However, the use of this parameter results in a slower execution speed.

12.1 DIFFERENCES REMOVED BY THE PARAMETER "LEVEL=GCOS1E"

12.1.1 Differences on Reading End of File

FORTRAN: A READ statement (with the "END=" parameter) recognizes both the special record made of 32 characters X'FE', or the Data Management end of file.

FORTRAN 77: A READ statement (with the "END=" parameter) recognizes only the Data Management end of file.
12.1.2 Differences on Split Records

FORTRAN: The record size of the file (RECSIZE) may be less than the size of the FORTRAN record, in which case several Data Management records are written or read by a single FORTRAN WRITE or READ statement.

FORTRAN 77: The record size of the file may not be less than the size of the FORTRAN RECORD except when a statement OPEN (unit, SPANNING=’YES’) has been executed previously to the read or write.

12.1.3 Differences on Argument Passing

FORTRAN: Character arguments are passed by address.

FORTRAN 77: Character arguments are passed by descriptors; Hollerith arguments are passed by address (Refer to Section 6 of this manual). An exception occurs if a literal passed as argument is modified in a subroutine.

12.1.4 Differences on Local Variables

FORTRAN: Local variables and arrays are allocated in the order of their declarations, without any gaps.

FORTRAN 77: Local variables and arrays are reorganized with gaps for alignment where necessary, except if specified in an EQUIVALENCE statement.

Note for the preservation of local variables:

When a variable (or an array) is local to a function or a subroutine its value is lost after execution of a RETURN statement. Therefore reentering the subroutine and using the variable in a computation may lead to unpredictable results.

As well as the use of the "LEVEL=GCOS1E" parameter, the difficulty may be avoided in one of the following ways:

- declare the variable in the dummy argument list
- declare the variable in a COMMON statement
- declare the variable in a SAVE statement
- use the SAVE statement without list.
12.2 DIFFERENCES NOT REMOVED BY THE PARAMETER "LEVEL=GCOS1E"

12.2.1 Differences on ENDFILE Statement

FORTRAN: The ENDFILE statement writes a special record made of 32 characters 'FE'X.

FORTRAN 77: The ENDFILE statement produces the Data Management end of file.

12.2.2 Differences on Secondary Entry Points

When S1 is called where X is not declared as an argument of S1 then the value of X is undefined.

*Example:*

```fortran
SUBROUTINE S(I,J,X)
  DIMENSION X(I,J)
  ENTRY S1

  X(k,l) is undefined when S1 is called.
```

12.2.3 Differences in Service Routine Names

Some service routine names have changed between FORTRAN and FORTRAN 77.

<table>
<thead>
<tr>
<th>FORTRAN NAME</th>
<th>FORTRAN 77 INTRINSIC FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCHNG</td>
<td>IBCHNG</td>
</tr>
<tr>
<td>BCLR</td>
<td>IBCHLR</td>
</tr>
<tr>
<td>BSET</td>
<td>IBSET</td>
</tr>
<tr>
<td>FRWST</td>
<td>IFRWST</td>
</tr>
<tr>
<td>FRTIME</td>
<td>IFRTIM,KFRTIM</td>
</tr>
<tr>
<td>FRTSWT</td>
<td>IFRTWT</td>
</tr>
</tbody>
</table>
12.2.4 Differences in JCL ASSIGN

The FORTRAN step option that gives the connection between the ifn and the units is changed to an implicit preconnection.

Example:

```
STEP PROG TEMP OPTIONS = 'FILE_E 10'
ASG FILE_E EFN.....
```

is changed to

```
STEP PROG TEMP
ASG 10 EFN...
```

explicit preconnection of 10 VIA ifn FILE_E is changed to an implicit preconnection via ASG 10.

12.2.5 Blank Characters in Formatted Read Statement

12.2.5.1 Record Length

Theoretically, the length of a record must be sufficient to satisfy the list and the format of the statement that reads it.

Specifying PAD='YES' in an OPEN statement causes READ statements on the same unit to behave as though the records were padded with as many blanks as required to satisfy the list and the format. Specifying PROCL=integer in a READ statement has the same effect, but temporarily, for this statement only. Moreover, integer is set to the value of the record length.

In the following cases, READ statements behave as though PAD='YES' were specified in the OPEN statement on the same unit:

- records directly typed at a terminal
- records belonging to a subfile of a source library and read in formatted mode.

12.2.5.2 Blank Spaces Ignored

This case arises when a real or integer is read with format E, F, D, Q or I, and concerns the processing of blank spaces which follow the first number (or the sign, or decimal point).

FORTRAN: These blanks are treated as "0"
FORTRAN 77: These blanks are ignored.
12.2.6  Exceptions on Execution

12.2.6.1  R3STACKOV

This occurs when the stack size is insufficient for the executable program. Use the STACK3 command of the linker. Note that the stack contains all the non SAVE variables.

12.2.6.2  WRITE RIGHT Violation

This occurs when an executable program attempts to modify a constant.

Example:

CALL S(2)
...
SUBROUTINE S(I)
  I=I+1
  ...

FORTRAN 77
13. Modular Programming and Protection

This section describes extensions that have been made to the ANSI 77 FORTRAN in order to facilitate the writing of large applications. The latter part of the section discusses protection and its use in reducing time in debugging programs.

13.1 MODULAR PROGRAMMING

13.1.1 Encapsulation of Procedures into Modules

The ANSI 77 FORTRAN lacks the facility of describing, in the language itself, modules made up of several program units. Some languages partially support this facility by the concept of internal procedures. A more general approach, and the one used in DPS 7 FORTRAN, is to allow the building of modules that are not executable procedures, but containers used for logical separation.

A module is a group of procedures performing functions on a set of objects of the same Abstract Type. The classical example is the abstract type: stack. The functions to be formed on it are: push an element on the top, pop the element off the top, pull the element off the top,...

A module, in a programming language, must:

(1) specify the procedures performing the functions,

(2) enclose these procedures into a unique programming unit,

(3) allow these procedures to share data and utility procedures while keeping the scope of the data and utilities to within the module.

When these three characteristics are satisfied for the procedures performing the module functions, it can be said that the procedures have been encapsulated into the module. In addition, a module must:

(4) specify the objects on which the functions apply.
The most modern programming languages (MODULA: a PASCAL extension, ADA) offer the module facility. With ANSI 77 FORTRAN, the only way to build a module is to group program units externally to the language, for instance by the linker. This has several inconveniences:

- The modular structure is not very clear to anyone reading the source program and can be seen only in the form of CALL statements and comments.

- Subroutines and external functions belonging to a module must share data by the use of COMMON blocks, and must share utility procedures by the use of external procedures. This results in a situation where data and utility procedures are not only shared by the procedures of the module, but are visible to other modules. This means that they may be inadvertently used by other modules without any compilation or linkage error message. If the executable program is very large and programmed by several people, this may lead to difficulties in detecting interdependence between modules, resulting in severe problems if one module has to be changed, without requiring the modification of the whole executable program.

DPS 7 FORTRAN 77 has been given a modular structure by introducing the PACKAGE facility while keeping extensions to the ANSI 77 standard to a minimum. The constraint of keeping as close as possible to the ANSI77 standard has brought about the following limitations:

- The dummy arguments of the encapsulated procedures cannot be specified at the beginning of the package unit.

- Only one level of modularity is permitted.

- The possibilities offered by DPS 7 FORTRAN 77 are:

  - PACKAGE and ENDPACKAGE statements: program units (procedures or main programs) are encapsulated by using the PACKAGE and ENDPACKAGE statements. The package unit is the module.

  - EXTROUTINE statements: EXTROUTINE statements are used to specify which of the program units of the package unit are visible outside the package unit. They perform the package unit functions.

  - EXTCOMMON statements: EXTCOMMON statements specify the common blocks of the package unit which are shared with other package units. These common blocks can be specified without any restrictions.

Note that EXTROUTINE is an export directive, specifying that a procedure of the package unit is visible from the outside; but there is no import directive, specifying that a reference inside the package unit to a procedure of the outside is expected. This is consistent with ANSI 77 which has the ENTRY statement for an entry point in a procedure, but no directive describing the procedure subject to a CALL.

On the contrary, EXTCOMMON is an import/export directive, specifying that a COMMON is shared by all the package units where EXTCOMMON appears with this name.
Example 13-1:

C PROC1 is a privileged procedure of the package unit
C and gives its name to the compile unit and
C listing
   PACKAGE PROC1

C PROC2 and PROC3 are the two other procedures
C of the package unit
   EXTROUTINE PROC2,PROC3

C FACE 1 is a common block shared with other
C package units
   EXTCOMMON  FACE1

C this is a utility procedure private to the package
C unit, as its name is not specified in the PACKAGE or
C EXTROUTINE statements.
   SUBROUTINE SERVICE
      ...
   END

C this is the realisation of PROC1,PROC2,PROC3
C together with the specification of their
C dummy arguments
   SUBROUTINE PROC1 (X,Y)
   COMMON /FACE1/ FACE11,FACE12,FACE13
   REAL X,Y
   REAL LOCAL1,LOCAL2

C procedures of the package unit may be called,
C as well as external procedures
   CALL PROC2 (LOCAL1,LOCAL2)
   COMMON /PRIVATE1/ P1,P2
   CALL SERVICE
   CALL EXTERNAL
   RETURN
   END

   SUBROUTINE PROC2 (X,Y)
C   X, Y, LOCAL1 are not the same as those of PROC1

COMMON /PRIVATE1/ P1, P2

REAL X, Y

INTEGER Y

INTEGER LOCAL1

...

END

REAL FUNCTION PROC3

...

END

ENDPACKAGE

After compilation, the package unit produces a compile unit PROC1 with the symbols PROC1, PROC2, PROC3, FACE1 for linkage. The symbols PRIVATE1 and SERVICE are not accessible by other program units whose compile units are linked with PROC1. Therefore these symbols can be reused in these other program units with another meaning.

13.1.2 Independence Between Logical Modularity and Physical Support

A package unit can span one or several library members, and several package units can be contained in one library member.

The SOURCE parameter of the FOR77 JCL statement specifies the library members to be compiled. If a package is split across several library members they must all be specified in the order corresponding to the order of the package unit statements (you can use the FROM, TO specification in the SOURCE parameter of the FOR77 JCL statement: refer to Section 2).

Given this order of statements, a library member change is allowed only before each program unit, or PACKAGE and ENDPACKAGE statements.

If the number of source lines of the package unit is very large, only one library member for this package unit would result in too slow a response to the Librarian or Text Editor commands, when updating it. By using one library member for each program unit of the package unit, you can take advantage of both the modularity offered by package units and the convenience of small library members.
These are the points where a change of library member is possible.

SUBROUTINE S1
...
END

SUBROUTINE S2
...
END

PROGRAM P1
...
END

ENDPACKAGE

If this package unit spans 5 library members: one (PP1) containing the package specification statements, three (S1, S2, P1) containing respectively the program units S1, S2, P1, and EP1 containing ENDPACKAGE, then the FOR77 JCL statement must specify SOURCE = (PP1, S1, S2, P1, EP1). If this package unit spans just one library member (P1), the FOR77 JCL statement must specify SOURCE = P1.
13.1.3 Packaging Program Units after Independent Compilations

In some cases it is not desirable to program program units as package units:

• because the executable program already exists without package units,

• because it is desired to keep within the ANSI 77 standard,

• because a large executable program is being built using a "bottom up" approach. In this case, it is not known beforehand which program units should be packaged together,

• and lastly, the programmer may wish to experiment with packaging and then go back to independent compilation of program units.

In all these cases, the programmer can, a posteriori or temporarily, use the package units without modifying the FORTRAN program units. To do this, one library member must contain the PACKAGE, EXTRoutine and EXTCOMMON statements, and another must contain the ENDPACKAGE statement.

The rules on COMMON blocks, and IMPLICIT specification of DPS 7 FORTRAN 77 are such that, after encapsulation into the package unit, the execution leads to the same results as before encapsulation. This is provided so that the executable program is free of errors and that, before encapsulation, there is no unexpected access to private objects of conceptual modules. It is encapsulation that provides protection against unexpected accesses being inadvertently coded (Refer to the following subsection: Protection).

Note that compilation of a package unit deletes from the library of Compile Units the previous compile units whose entry points are specified by the EXTRoutine statements of the package unit. Conversely, independent compilation of one program unit that has an entry point specified in the EXTRoutine statement of a package unit deletes from the library of compile units the compile unit of the package unit.
Example 13:

The following program units were first independently compiled and now are encapsulated into a package unit. It is assumed that S3 and C2 are referred to by only S1 and S2 in the executable program.

```fortran
PACKAGE S1
  EXTRoutine S2
  EXTCOMMON C1
  SUBROUTINE S1
    COMMON/C1/A,B
    COMMON/C2/C,D
    INTEGER I
    CALL S3
    ...
  END

SUBROUTINE S2
  COMMON/C1/B,A
  COMMON/C2/C,D
  INTEGER I
  ...
END

SUBROUTINE S3
  COMMON/C2/C,D
  ...
END
ENDPACKAGE
```

Note that:

- The scope of S1,S2,C1 is the executable program.
- The scope of S3,C2 is the package unit.
- The scope of A,B,C,D,I is the program unit.

As a consequence:

- S3 and C2 are not accessible by other (program or package) units of the executable program.
- The I of S1 is different from the I of S2, as when S1 and S2 are independently compiled.
- The B of S2 occupies the same four numeric storage units as the A of S1. The A of S2 occupies the same four numeric storage units as the B of S1. This is as when S1 and S2 were independently compiled and then linked.
13.1.4 Static Testing of an Executable Program

13.1.4.1 Cross Reference Table of a Package Unit

When compiling a package unit with the XREF and MAP parameters of the $FOR77 JCL statement, a cross reference table is given for all the symbols of the package unit, suffixed by the name of the procedure in which they occur.

This is especially useful for the elements of COMMON blocks in order to know in which procedure they have been modified.

13.1.4.2 Cross Reference Table of a Large Executable Program

It is useful for a large executable program (e.g., 100,000 source lines) made up of several program units, each on a separate library member, to have a global cross reference. This is especially the case for the COMMON elements and procedure calls. But the compilation time or the compiler limits can preclude producing a compile unit for such a package unit. In this case, the programmer can try to encapsulate the whole executable program into a package unit by adding a library member containing a PACKAGE statement, and another containing an ENDPACKAGE statement. The XREF and NOBJ parameters of the FOR77 JCL statement and the library members in the SOURCE parameter must be specified. The star convention is useful to avoid enumeration of the library members (refer to Section 2). With NOBJ specified, only the analysis phases of the compiler are exercised, resulting in a very short compilation time and the compiler needing far fewer system resources.

13.1.4.3 Check of Procedure Calls

For an executable program, the linker checks for the consistency of types between the specification of procedures and their calls and between the dummy and actual arguments. However, as the linker allows linkage of several languages (FORTRAN, COBOL, GPL, etc.), the error messages do not use the FORTRAN 77 terminology. If the executable program is not too large, it may be programmed as one package unit. In this case, all inter-procedure checks are made by the FORTRAN 77 compiler, and error messages use the FORTRAN 77 terminology.
13.1.5 Packaging to Reduce the Execution Time and Number of Segments

Packaging reduces the system resources (CPU time and segments) needed by the encapsulated program units. These features are discussed in detail in Section 8: Efficiency Techniques, and Section 7: Segmentation and Addressing. They are only summarized here.

Encapsulation of program units into a package unit results in the following modifications:

- **Execution speed increased**
  - Faster procedure calls inside the package unit
  - Faster references to COMMON elements, if there is no, or more than one, common block specified in the EXTCOMMON statement; and if the DSEGMAX parameter of the FOR77 JCL statement is not used (refer to Section 2).

- **Avoidance of small segments**
  Small segments (less than 1 Kbyte) resulting from small program units are avoided, as the program units are grouped in a package unit. This decreases the overhead of Virtual Memory Management.

- **Avoidance of overflow of the number of segments**
  This is especially useful in two cases (non-exclusive):
  (a) In the case of a large executable program made up of about 100 or 200 program units. If they are independently compiled, this may lead to more segments than a process or process group may allow.
  (b) Where there are less than this many program units but when compilation is carried out with the DEBUG parameter of the FOR77 statement specified, additional segments are created for the Program Checkout Facility.

In all cases, by grouping program units into package units, the number of segments needed by the executable program will be reduced and overflow avoided. Note that overflow of the number of segments is a fatal error at linkage.
13.2 PROTECTION

13.2.1 What is Protection

Protection is the software plus hardware mechanisms that ensure the executable program is constructed (or may be generated) according to the rules specified in the FORTRAN 77 Reference Manual and the rules of GCOS specified in the GCOS manuals for non-FORTRAN points (Job Control Language, etc.). If some part of the program is not consistent with these rules, it is called an error.

Note that protection does not prevent the situation where, given the specification of a problem, someone builds a FORTRAN program that is consistent with DPS 7 FORTRAN 77 rules, but is not a solution of the problem. For instance, a correct single precision computation where double precision is required so that rounding errors are negligible. However protection ensures that, due to an inadequate algorithm, or a coding error or unexpected input data, the executable program performs inconsistently with the rules of the manuals. For instance, the previous single computation can lead to a zero intermediate result, but may not perform a zero divide. In this case, the protection is the hardware acceptation mechanism.

13.2.2 Requirements for Protection

Good protection ensures that:

(1) The protection mechanisms do not consume any computer resources (memory, time) when processing a correct program,

(2) An error must be signaled by the processing system as soon as it appears.

Point (2) has the following consequences:

• An error must be discovered at compilation rather than at linkage, or at linkage rather than at execution,

• If it may be discovered only at execution, it must be signaled when executing the erroneous statement, and not much later, when executing other parts of the executable program,

• If this is not possible, the following, at least, is necessary: when the erroneous statement is in some new part added (linked) to an executable program, the error must be signaled as a malfunction of this new part, and not as a perturbation inside previously debugged parts of the executable program.
13.2.3 Protection with DPS 7 FORTRAN 77

DPS 7 FORTRAN 77 emphasis on protection encompasses the following points:

13.2.3.1 Early Error Signaling

Features not allowed by the Reference Manual are detected by decreasing preference by the compiler, the Linker, the Run Time Package (an interpreter for statements that are not in-line generated) and the exception mechanism.

- The compiler systematically rejects or flags constructions that are tolerated on other systems.
- The observation messages flag features consistent with the rules of the FORTRAN 77 Reference Manual, but which generally result from an error.

These constructs are supported because they are part of ANSI 77, and observations are printed only if the OBSERV parameter is specified in the FOR77 JCL statement.

- The Linker flags mismatches between the types and length of actual and dummy arguments.
- Depending on the flow control of the program, the Run Time Package or exception mechanism will be used first.

13.2.3.2 Exception Mechanism

To ensure correspondence between the software rules of FORTRAN and the protections of the hardware, the compiler takes advantage of the segmented virtual memory of the DPS 7. For a subroutine independently compiled, the code and constants can be in one or many segments, the local data also can be in one or several segments, but each COMMON block will be in one segment (Refer to Section 7).

If a branching error, or data access error would cause an exit or lead to an access outside of the subroutine then an out of segment bounds exception occurs at run time.

If execution is run under the Program Checkout Facility (refer to Section 4), in batch or interactive mode, the exception activates this program.

This program prints a message describing the exception, the name of the procedure and the number of the source statement line that produced the exception.

In interactive mode, this is printed on the terminal, then returning control for debugging.

If execution is not run under the Program Checkout Facility, the message describing the exception, and the address of the machine instruction that produced it, is printed in the Job Occurrence Report in batch mode, or at your terminal in interactive mode.
13.2.3.3 Run Time Package

The Run Time Package offers additional protection when it is called by the executable program to interpret input/output statements, or when intrinsic functions implemented by a call to a procedure. (Intrinsic functions can also be implemented by code generated on-line in the procedure in which they are invoked. (See Appendix A).

When an error occurs, a run time error message prefixed by F7R is emitted. It describes the error, the name of the procedure, the number of the source line that produced the error, the unit number in case of I/O, or the intrinsic function name. This protection does not consume time for correct cases. The message can be directed to the unit corresponding to WRITE(UNIT=“), or to the Job Occurrence Report, by use of the LSETK intrinsic function.

13.2.3.4 SUBCK Parameter of the FOR77 JCL Statement

Strong protection is also offered by generation of additional checks in the object code:

- check array element subscripts, within the array bounds
- check substrings subscripts, within the string
- check integer variable of assigned GOTO, within the list of labels.

In case of error these checks lead to a call to the Run Time Package with messages similar to the above.

As these checks add some CPU time at execution, they are generated only if the unit is compiled with the SUBCK parameter of the FOR77 JCL statement (refer to Section 2).

13.2.3.5 Non-perturbation of Previously Debugged Module

The package feature is a DPS 7 FORTRAN 77 extension to ANSI 77 offered to encapsulate a group of procedures and common blocks that form a module, into a package unit (refer to previous subsection).

When you encapsulate procedures, previously compiled, independently into a package unit, the following occurs:

(a) Protection performed at compile time is enforced because inter-procedural checking is performed by the compiler.

(b) Protection performed at linkage is for the package unit interfaces.

(c) Protection performed at run time is:

- Enforced for the inter packages relations
- Released for the intra package relations.
Modular Programming and Protection

The reason is:

- Protection consumes system resources (CPU time for segment switching, and virtual memory segments). To prevent contention, the highest abstraction level (packages) is privileged against the lowest (procedures).

- Procedures and COMMON blocks of a package unit cannot be accessed from the outside unless they are declared in an EXTRoutine or EXTCOMMON statement or in the PACKAGE statement itself.

13.2.4 Protections of Independently Compiled Program Units

When a program unit is not encapsulated into a package unit the following protections are offered by DPS 7 FORTRAN 77.

13.2.4.1 Access out of a List (ARRAY, STRING, LABELS)

<table>
<thead>
<tr>
<th>Description of the error</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array subscript expression less than the lower array bound, or greater than the upper array bound except if asterisk</td>
<td>In-line code</td>
<td>SUBCK at compilation</td>
</tr>
<tr>
<td>Substring expression less than one, or greater than the length of the character variable or array element, even if asterisk length is specified.</td>
<td>In-line code</td>
<td>SUBCK at compilation</td>
</tr>
<tr>
<td>Integer variable referred in an assigned GOTO and not assigned with one of the labels specified in the label list of the GOTO</td>
<td>In-line code</td>
<td>SUBCK at compilation</td>
</tr>
</tbody>
</table>

13.2.4.2 Mismatch Between Actual and Dummy Arguments

<table>
<thead>
<tr>
<th>Description of the error</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual argument is of type different from the dummy argument</td>
<td>Linker (message no 2808)</td>
<td></td>
</tr>
<tr>
<td>In the same program unit the actual argument of a call is an array name, and the actual argument of another call is a variable name or an expression not reduced to an array element</td>
<td>Compiler (warning)</td>
<td>WARN</td>
</tr>
</tbody>
</table>
### Description of the error and Protection by Upon option

**Actual argument is:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>a variable name, or:</td>
<td>Linker</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>an expression not reduced to an array element nor to hollerith constant, or:</td>
<td>(message no 2808)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>an array element, and there are other calls to the same procedure, with the corresponding actual argument in case (1) or (2) (so we know that we do not pass an array section).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dummy argument is an array, not of character type.**

**Actual argument is:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>an array name, or:</td>
<td>Linker</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>an array element, and there is one call in the program unit to the same procedure, with the corresponding actual argument in case (1). (So we know that we pass on array section, starting from the specified array element).</td>
<td>(message no 2809)</td>
<td></td>
</tr>
</tbody>
</table>

**Dummy argument is a variable name, not of character type.**

**Actual argument is:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>an array name, or:</td>
<td>Linker</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>an array element name, and there is a call in the program unit to the same procedure, with the corresponding actual argument in case (1).</td>
<td>Note that as no description is passed for arrays, there is no run time check that the array section passed as actual argument is less than the dummy argument.</td>
<td></td>
</tr>
</tbody>
</table>

**Dummy argument is a constant array (as opposed to adjustable or assumed size array).**

The size, expressed in storage units, of the actual argument is greater than the size of the dummy argument.

If the actual argument is array element, the size considered is the total size of the array. (As the size of the array section starting from the array element is only known at run time).
<table>
<thead>
<tr>
<th>Description of the error</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual argument is of character type and is an array name or expression. Dummy argument is of character type, and is :</td>
<td>Linker (message no 2807)</td>
<td></td>
</tr>
<tr>
<td>(1) a constant array, or :</td>
<td>Note that there is no run time check that the array section or the substring passed as actual argument is less than the dummy argument</td>
<td></td>
</tr>
<tr>
<td>(2) a variable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The size expressed in storage units of the actual argument is greater than the size of the dummy argument. If the actual argument is an array element or a substring, the size considered is the total size of the character variable or character array.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual argument is a hollerith constant.</td>
<td>Compiler warning. No check at linkage nor run time.</td>
<td></td>
</tr>
<tr>
<td>Dummy argument is an assumed size array (asterisk upper bound). Actual argument is of same type and is :</td>
<td>No check.</td>
<td></td>
</tr>
<tr>
<td>(1) an array name, or:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) an array element, and there is no call in the program unit to the same procedure, with the corresponding actual argument in case (1).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual argument is an expression not reduced to a constant, array element, substring or variable. Dummy argument is assigned by the callee. Note that in CALL S((VAR)) the expression (VAR) is reduced to VAR for optimization reasons, and that you must write CALL S (EVALUATE(VAR)) with the EVALUATE intrinsic function to force copy of VAR. (Refer to Section 6: Calling and Called Programs, and to the FORTRAN 77 Reference Manual 6.6.3: Integrity of Parentheses).</td>
<td>No check, but the actual argument is copied before passed.</td>
<td></td>
</tr>
<tr>
<td>Attempt to modify the constants of another procedure.</td>
<td>Exception mechanism. Constant plus code of each procedure are in separate segments.</td>
<td></td>
</tr>
</tbody>
</table>
### 13.2.4.3 Unauthorized Flow of Control

<table>
<thead>
<tr>
<th>Description of the error</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO into another procedure.</td>
<td>Exception mechanism.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>There is one segment per procedure for code.</td>
<td></td>
</tr>
<tr>
<td>Assigned GOTO with the integer variable assigned to a label of the procedure, but not belonging to the GOTO label list.</td>
<td><strong>No check.</strong></td>
<td>In the absence of the SUBCK compilation option.</td>
</tr>
<tr>
<td>Reference to a procedure name (by EXTERNAL, CALL or function call) not belonging to the executable program.</td>
<td>Linker and if ignored: Exception mechanism.</td>
<td></td>
</tr>
</tbody>
</table>

### 13.2.4.4 Access to Data Not in the Scope of the Referring Statement, or Out of the Referenced List (ARRAY or STRING)

<table>
<thead>
<tr>
<th>Description of the error</th>
<th>Protection by</th>
<th>Upon option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification of a constant or of the executable statements.</td>
<td>Exception mechanism.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constants and code are allocated in non writeable segments.</td>
<td></td>
</tr>
<tr>
<td>Access to the SAVE local to another procedure.</td>
<td>Exception mechanism.</td>
<td>LEVEL = GCOS1F at compilation</td>
</tr>
<tr>
<td></td>
<td>There is one segment for SAVE data of each independently compiled program unit.</td>
<td></td>
</tr>
<tr>
<td>Access to the non-SAVE local data of an inactive procedure.</td>
<td>Exception mechanism.</td>
<td>LEVEL = DSEGMAX</td>
</tr>
<tr>
<td></td>
<td>This data is not allocated.</td>
<td>DSEGMAX at compilation but without possibility to choose the segmentation rule.</td>
</tr>
<tr>
<td>Access to non-equivalenced local datum A by mean of wrong subscripting of datum B.</td>
<td>Exception mechanism only if the segments to which A and B belong are different. So in the usual case there is no check.</td>
<td></td>
</tr>
</tbody>
</table>
13.2.5 Protection of Package Units

The protection for a whole package W is the same as the protection for an independently compiled program unit I:

- Constants and statements of W are globally protected like those of I.
- Common blocks of W specified in an EXTCOMMON statement are protected like common blocks of I. The other common blocks of W are globally protected like local data of I.
- Local data of all the program units of W are globally protected like the local data of I.
- Entries of W specified in EXTROUTINE are protected like entries of I. The other entries of W are protected like labels of I.

At the expense of system resources (CPU time and segments) the PSEGMAX and DSEGMAX parameters of the FOR77 JCL statement allow increased protection.

- PSEGMAX = 0 implies that each program unit’s executable code is allocated into a separate segment, protecting from erroneous assigned GOTO between procedures.
- DSEGMAX = n implies that data are grouped by segments of n Kbytes. However, this grouping does not correspond to program units, but to an optimization of the cache memory, and is of little use for protection.
FORTRAN 77
14. Optimizing with FORTRAN 77

14.1 INTRODUCTION

14.1.1 The Goals of the Optimizer

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

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

```
DIMENSION A(0:100), B(0:100)
DO 10 I=0,100
    DO 20 J=1,100
        A(i+J) = B(i+j)
    20      CONTINUE
10    CONTINUE
```

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

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

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

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

The local optimizer is used when the global optimizer is not requested. Before the 74 version, for local optimization, the Fortran 77 compiler had only two optimization levels, the source instruction optimization level and the extended linear sequence optimization level.

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. These instructions 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.

14.1.3 The Global Optimizer

The global optimizer extends the optimization reach for a whole procedure. It 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.

By analyzing the problem graph and the data flow, the compiler can operate an elaborate optimization. This is due to the manipulations that the optimization functions perform on an internal representation of the source code. These manipulations include deletion, insertion, and instruction replacement.

The global optimization functions are as follows:

4. Anticipation and temporization.
5. Deleting partially redundant expressions.
6. Removing invariant expressions in 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).

NOTES:
1. The optimization functions perform at the procedure level. There are no inter-procedural optimizations (for example, between the procedures of the same PACKAGE).

2. There are two types of local or global optimizing improvements:
   - Increased speed in program execution.
   - Decreased volume of code generation, except in optimization cases of loop unrolling (8) and procedure merging (9).

Restrictions in Optimizing

The optimizer follows these rules:

Efficiency Rule The optimizing functions work only if the application shows an improvement in storage or time efficiency in all possible execution cases of the program.

Coherence Rule An optimization function must never affect the semantics of a program. In particular, if a program executes correctly and conforms to the definition of a language without optimization, then optimization must not cause the program to abort.

Compromised Time and Storage Rule The optimizer gives greater importance to the optimization functions that contribute a gain in execution time than to those that contribute to the reduction of generated volume of source code.
14.1.4 Optimization Levels

The Fortran 77 compiler has five optimization levels. Each level is guided by one of the OPTIMIZE parameter levels, as follows:

OPTIMIZE=0  No optimization

OPTIMIZE=1  Local optimization, limited to the source statement.

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.
14.2 GLOBAL OPTIMIZER FUNCTIONS

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

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

14.2.1 Constant Folding and Copy Propagation

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

```
A = 1
IF (VALID) THEN
    X = A + 3
ELSE
    X = A + 1
ENDIF
```

This gives the following, after optimization:

```
A = 1
IF (VALID) THEN
    X = 4
ELSE
    X = 2
ENDIF
```

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

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

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

X = A + B + C
IF (A.GT.B) THEN
  X = 10
ELSE
  X = 20
ENDIF
Y = A + B + D

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

T = A + B
X = T + C
IF (A.GT.B) THEN
  X = 10
ELSE
  X = 20
ENDIF
Y = T + D

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

NOTE: This function is legal only if the value of the string and variable are the same. For example, refer to the following statement:

SUBROUTINE S (X, A, B)

If the variables X and A are dummy values, the optimization function can not always work. This is because the variables X and A could correspond to the same values in, for example, the following statement:

CALL S(A, A, A)
14.2.3 Deleting Useless or Inaccessible Code

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

```
A = 1
B = A - 1
IF (A.LT.B) THEN
  C = B
ELSE
  C = A
ENDIF
C = C * 2
```

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

```
A = 1
B = 0
IF (1.LT.0) THEN
  C = 0
ELSE
  C = 1
ENDIF
C = C * 2
```

**Deleting Useless Code**

When the optimization functions evaluate the above example, it creates some useless code. Deleting the useless code results in the following:

```
IF (1.LT.0) THEN
  C = 0
ELSE
  C = 1
ENDIF
C = C * 2
```

**Deleting Inaccessible Code**

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

```
C = 1
C = C * 2
```
14.2.4 Anticipation and Temporization

Two of the optimization functions reduce the object code, but do not shorten program execution time. These functions either bring forward or set back data expressions that use the IF-THEN-ELSE instruction in the program. They move the data expressions that are within the THEN and ELSE outside, towards the top or the bottom. In this way, the data 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:

```fortran
IF (U.GT.V) THEN
  X = A + B
  A = U
ELSE
  X = A + B
  B = V
ENDIF
```

This yields the following after optimization:

```fortran
X = A + B
IF (U.GT.V) THEN
  A = U
ELSE
  B = V
ENDIF
```

The following is an example of temporization:

```fortran
IF (U.GT.V) THEN
  A = U
  X = A + B
ELSE
  B = V
  X = A + B
ENDIF
```

This yields the following after optimization:

```fortran
IF (U.GT.V) THEN
  A = U
ELSE
  B = V
ENDIF
X = A + B
```
14.2.5 Deleting Partially Redundant Expressions

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

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

```
IF (X.EQ.1) THEN
    X = (A + B)
ELSE
    A = 1
ENDIF
X = A + B
```

In the example above, the assignment (statement) 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.EQ.1) THEN
    X = A + B
ELSE
    A = 1
    X = A + B
ENDIF
```

14.2.6 Removing Loop Invariants

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

**Example 1:**

```
DO 1 I = 1, 10
    X (I) = A + B
 1 CONTINUE
```

**Example 2:**

```
DO 1 I = 1, J
    X (I) = A + B
 1 CONTINUE
```
Example 3:

```fortran
do 1 i = 1, 10
   if (y.gt.0) then
      x (i) = sqrt (y)
   endif
1   continue
```

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

Example 1, from above, after optimization:

```fortran
t = a + b
   do 1 i = 1, 10
      x (i) = t
1   continue
```

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

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

Example 2, from above, after rearrangement:

```fortran
if (j.ge.1) then
   do 1 i = 1, j
      x (i) = a + b
1   continue
endif
```

Example 2, after optimization:

```fortran
if (j.ge.1) then
   t = a + b
   do 1 i = 1, j
      x (i) = t
1   continue
endif
```

It is not possible to remove the loop invariant, "SQRT (Y)", from the third example. This is because no rearrangement can be made that does not interfere with the coherence rule.
14.2.7 Strength Reduction and Processing of Loop Control Variables

14.2.7.1 Strength Reduction

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

**Step 1:**

The detection of all the variables in the loop, progressing at a constant step (increment) 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:

\[
X \times C
\]

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:

\[
T = T + K \times C
\]

The product of \( K \times C \) is evaluated at compile time. An example of this optimization function is:

```
DO 1 I = 1, 10, 2
   X = X + 4 * I
1   CONTINUE
```

After optimization:

```
T = -4
DO 1 I = 1, 10, 2
   T = T + 8
   X = X + T
1   CONTINUE
```
14.2.7.2 Processing of Loop Control Variables

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

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

\[
\begin{align*}
T &= -4 \\
I &= 1 \\
90 & \quad T = T + 8 \\
X &= X + T \\
I &= I + 2 \\
& \quad \text{IF (I.LE.10) GOTO 90 }
\end{align*}
\]

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

\[
\begin{align*}
T &= -4 \\
I &= 1 \\
90 & \quad T = T + 8 \\
X &= X + T \\
I &= I + 2 \\
& \quad \text{IF (T.NE.36) GOTO 90 }
\end{align*}
\]

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:

\[
\begin{align*}
T &= -4 \\
I &= 11 \\
90 & \quad T = T + 8 \\
X &= X + T \\
& \quad \text{IF (T.NE.36) GOTO 90 }
\end{align*}
\]

14.2.8 Loop Unrolling

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

For small size loops, the expansion is total. A small loop is one in which the number of iterations does not exceed 20. In other loops, the unrolling is partial, provided that the ratio of expansion is not great. The loop unrolling optimization function limits itself to only the lowest level loops, as shown in the following example.
DO 1 I = 1, 25  
   K = 25 * (I - 1)  
DO 2 J = 1, 25  
   X (K + J) = J  
   CONTINUE  
1   CONTINUE  

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

DO 1 I = 1, 25  
   K = 25 * (I - 1)  
DO 2 J = 1, 5  
   X (K + J) = J  
   J = J + 1  
   X (K + J) = J  
   J = J + 1  
   X (K + J) = J  
   J = J + 1  
   X (K + J) = J  
   J = J + 1  
   X (K + J) = J  
   J = J + 1  
2   CONTINUE  
1   CONTINUE  

The program can then be optimized using the algorithms described above.

14.2.9 Procedure Merging

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

In Fortran 77, the statement functions are the only internal procedures (although the GCOS7 PACKAGE extension allows procedure gathering in a PACKAGE unit). Statement functions are inserted on line at a different level if the value for OPTIMIZE is specified as 2 or greater. The other inter-procedural references are not done on line, except for most of the intrinsic service functions such as INT and MAX. The inter-package references are not inserted on line. However, this restriction is susceptible to change in another version.
14.3 USING THE GLOBAL OPTIMIZER

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

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

The global optimizer functions work independently with only one procedure at a time. There are no inter-procedural optimization functions, as there could be, for example, inside a PACKAGE.
# A. List of Intrinsic Functions

For each intrinsic function, this list specifies when it is implemented by firmware, and when it can be passed as an argument to a called procedure.

<table>
<thead>
<tr>
<th>Generic name</th>
<th>Specific name</th>
<th>Is it implemented by firmware?</th>
<th>Can it passed as an argument?</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFIX</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>INIT</td>
<td>INT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>IFIX</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>IDFINT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>IQINT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>SNGL</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>SNQLQ</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DBLE</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>DBFLOAL</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>DBLEQ</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>DREAL</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>QEXT</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>QFLOAT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>QEXT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>QEXTD</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>CMPLX</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>CSNGL</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>CINT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DCMPLX</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>CDBLE</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>CDINT</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IHOL</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>KTYPE</td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>CHAR</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>HCHAR</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>AINT</td>
<td>AINT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DINT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QINT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ANINT</td>
<td>ANINT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DNINT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QNINT</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

47 A2 16UL Rev03

A-1
<table>
<thead>
<tr>
<th>Generic name</th>
<th>Specific name</th>
<th>Is it implemented by firmware?</th>
<th>Can it passed as an argument?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONJG</td>
<td>CONJG</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DCONJG</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SQRT</td>
<td>SQRT</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DSQRT</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QSQRT</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
## List of Intrinsic Functions

<table>
<thead>
<tr>
<th>Generic name</th>
<th>Specific name</th>
<th>Is it implemented by firmware?</th>
<th>Can it passed as an argument?</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>EXP</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DEXP</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QEXP</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CEXP</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDEXP</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LOG</td>
<td>LOG</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>ALOG</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DLOG</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QLOG</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CLOG</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDLOG</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LOG10</td>
<td>ALOG10</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>DLOG10</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QLOG10</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LOG2</td>
<td>ALOG2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>DLOG2</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QLOG2</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SIN</td>
<td>SIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DSIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QSIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CSIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDSIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SIND</td>
<td>SIND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DSIND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QSIND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>COS</td>
<td>COS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DCOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QCOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CCOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDCOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>COSD</td>
<td>COSD</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DCOSD</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QCOSD</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TAN</td>
<td>TAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TAND</td>
<td>TAND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DTAND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QTAND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>COTAN</td>
<td>COTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DCOTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QCOTAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>ASIN</td>
<td>ASIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DASIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QASIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CASIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>CDASIN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>ACOS</td>
<td>ACOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DACOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>QACOS</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Generic name</td>
<td>Specific name</td>
<td>Is it implemented by firmware?</td>
<td>Can it passed as an argument?</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>FORTRAN 77</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CACOS</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDACOS</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATAN</td>
<td>ATAN</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DATAN</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QATAN</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CATAN</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDATAN</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATAND</td>
<td>ATAND</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DATAND</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QATAND</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATAN2</td>
<td>ATAN2</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DATAN2</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QATAN2</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATAN2D</td>
<td>ATAN2D</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DATAN2D</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QATAN2D</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>SINH</td>
<td>SINH</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DSINH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QSINH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CSINH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDSINH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>COSH</td>
<td>COSH</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DCOSH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QCOSH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CCOSH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDCOSH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TANH</td>
<td>TANH</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DTANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QTANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CTANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDTANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATANH</td>
<td>ATANH</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DATANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QATANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CATANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CDATANH</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>GAMMA</td>
<td>GAMMA</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DGAMMA</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QGAMMA</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ALGAMA</td>
<td>ALGAM</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DLGAM</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QLGAM</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>FACT</td>
<td>FACT</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DFACT</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QFACT</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generic name</th>
<th>Specific name</th>
<th>Is it implemented by firmware?</th>
<th>Can it passed as an argument?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERF</td>
<td>ERF</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DERF</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QERF</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ERFC</td>
<td>ERFC</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DERFC</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QERFC</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>STWO</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DTWO</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QTWO</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>LGE</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>LGT</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
List of Intrinsic Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>LLE</th>
<th>LLT</th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAND</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IOR</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IXOR</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ICOMPL</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ISHA</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ISHC</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ISHL</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>BTTEST</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>BITCLR</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IBSET</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IBCHNG</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IFRTIM</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>KTOD</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>KFRTIM</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IWAIT</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IFRTWT</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IFRSWT</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>KTEST</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TTEST</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>FRERR</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IPDUMP</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LSETK</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LSETI</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LSETS</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>EVALUATE</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LOCF</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Technical publication remarks form

Title: DPS7000/XTA NOVASCALE 7000 FORTRAN 77 User’s Guide Languages: FORTRAN

Reference N°: 47 A2 16UL 03 Date: July 1990

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required. If you require a written reply, please include your complete mailing address below.

NAME: ________________________________ Date: __________
COMPANY: ______________________________
ADDRESS: ______________________________

Please give this technical publication remarks form to your BULL representative or mail to:

Bull - Documentation D\textsuperscript{ap}t.
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 PATON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

<table>
<thead>
<tr>
<th>CEDOC Reference #</th>
<th>Designation</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
<tr>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
<td>_ _ _ _ _ _ ( _ )</td>
</tr>
</tbody>
</table>

( \_ \_ ) : 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.