DEVELOPING PROGRAMS WITH FORTRAN VII -

A Guide

48-010 F00 R04

Concurrent
Computer Corporation

Disclaimer

The information contained in this document is subject to change without notice. Concurrent Computer Corporation has taken efforts to remove errors from this document; however, Concurrent Computer Corporation's only liability regarding errors that may still exist is to correct said errors upon their being made known to Concurrent Computer Corporation.

Concurrent Computer Corporation assumes no responsibility for the use or reliability of this software if used on equipment that is not supplied by Concurrent Computer Corporation.

License

The software described in this document is furnished under a license, and it can be used or copied only in a manner permitted by that license. Any copy of the described software must include all copyright notices, trademarks, or other legends or credits of Concurrent Computer Corporation and/or its suppliers. Title to and ownership of the described software and any copies thereof shall remain in Concurrent Computer Corporation and/or its suppliers.

The licensed program described herein may contain certain encryptions or other devices which may prevent or detect unauthorized use of the Licensed Software. Temporary use permitted by the terms of the License Agreement may require assistance from Concurrent Computer Corporation.

Acknowledgments

Concurrent Computer Corporation
106 Apple Street
Tinton Falls, NJ 07724

Printed in the United States of America

About This Book

Overview

This book provides you with the necessary information to develop programs using the FORTRAN VII Language System. It guides you through each step of the program development process: coding, compiling, linking, executing, and debugging your program. It also presents supplemental information on the run-time library (RTL) routines, call recording analysis (CRA) system, and execution profile analysis (XPA) system.

Before You Start

When using this book, you must be familiar with the basic programming concepts, FORTRAN VII Language syntax and semantics, and OS/32 and multi-terminal monitor (MTM) environments. Familiarity with the command substitution system (CSS), and Edit/32 is also helpful.

Using This Book

The objective of this manual is to assist you in the development of FORTRAN VII programs under the OS/32 environment. New users should read the manual from cover-to-cover to become familiar with each program development procedures. Experienced users should use the guide as a reference tool to verify proper usage of the FORTRAN VII language.

Other Sources of Information

It may be helpful to supplement some of the information in this manual with the following:

- *FORTRAN VII Language and Syntax — A Reference (48-017)*

  This manual presents the FORTRAN VII Language syntax and semantic rules.

- *OS/32 System Support Run-Time Library (RTL) (48-152)*

  This reference manual describes the OS/32 Support RTL subroutines and functions.

- *OS/32 Link Reference Manual (48-005)*

  This manual presents the command description for the Link process.

- *OS/32 Patch Reference Manual (48-016)*

  This manual is a guide to using Concurrent's OS/32 Patch Utility. Patch allows the user to apply software changes to object or image code without reassembling the source module.

Document Organization

This manual is composed of 15 chapters grouped into 7 parts. It has two appendixes. A brief description of each chapter in the manual follows:

Part I - FORTRAN VII Environment

- Chapter 1 presents an overview of FORTRAN VII, the Development and Optimizing compilers, other related products, and the requirements for maintaining the environment.
- Chapter 2 presents a review of the MTM environment, use of CSS, and Edit/32. It illustrates the use of the different program development commands: COMPILE, LINK, COMPLINK, RUN, and EXEC.

Part II - Programming

- Chapter 3 introduces the available instream compiler directives. These directives are categorized by function and further details are found in later chapters. A table of both instream and start directives is also presented.
- Chapter 4 presents some useful guidelines for producing efficient code. It provides the programmer with information on inherent features of FORTRAN VII which, if not used properly, can produce inaccurate results. Specifically, it covers the following: use of dummy arguments, DO loop processing, use of computed and assigned GOTOs, use of array subscripts and parentheses, data type conversions, test for floating point values, etc.

  This chapter also describes the different optimizations that occur to a program when compiled using the optimizing compiler. These may either be built in (cannot be prevented from occurring) or optional (can be prevented from occurring). Guidelines for preparing code for optimization are presented.
- Chapter 5 discusses the procedures for interfacing FORTRAN VII programs with assembly language programs. It describes the standard FORTRAN VII calling sequence, how to insert assembly language code in FORTRAN source, and how to write a program development procedure for FORTRAN with embedded assembly code.

Part III - Compiling

- Chapter 6 describes the procedures for compiling programs using the development and optimizing compilers. Use of the start directives for both compilers are also detailed.

Part IV - Linking

- Chapter 7 details the procedure for linking a successfully compiled program. It further discusses linking programs with trap handling routines, programs that access shared data areas, and segments. The chapter also explains the procedures for linking large programs into segments (overlays).

Part V - Executing

- Chapter 8 discusses the procedures for loading and starting your program after it was successfully compiled and linked. Assignment of logical units are also covered.

Part VI - Debugging

- Chapter 9 provides guidelines for debugging programs using compiler directives. Specifically, it shows how to conditionally compile programs, trace executable statements, check array subscripts, check intermediate values, etc. It also describes how to analyze run-time error messages.

- Chapter 10 illustrates how to read program maps and listings. It defines the important information found on compiler listings and link maps.

Part VII - Supplemental Information

- Chapter 11 describes a few of the basic RTL routines. Others are found in the *OS/32 System Support Run-Time Library (RTL)* and *Math Run-Time Library (RTL) Reference Manuals.*

- Chapter 12 describes the use of the Execution Profile Analysis (XPA) system.

- Chapter 13 describes the use of the CRA system.

- Chapter 14 explains the different rounding techniques performed on floating point calculations. It discusses the different factors which might affect results of the calculations.

- Chapter 15 describes the phases involved in the optimization process performed by the F7O and F7Z compilers.
- Chapter 16 lists all the FORTRAN VII related error messages.
- Appendix A describes the FORTRAN subprograms which are not directly callable from FORTRAN code.

Contents

Chapters

6 Building a Command File to Compile Your Program

7 Building a Command File to Link a FORTRAN Program

8 Building a Command File to Execute a FORTRAN Program

9 Run-Time Debugging

13 FORTRAN VII CRA System

14 Floating Point Calculations

15 Universal Optimization

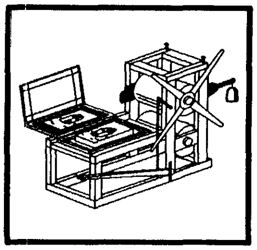
16 FORTRAN VII Error Messages

Appendixes

A RTL Subprograms

Figures

Tables



Revision History

Manual Revision	*Released*	*Minimum FORTRAN Software Version*	*Minimum OS/32 Software Version*
F00 R02	February 1986	R05-03	R08-01
F00 R03	July 1987	R05-05	R08-02
F00 R04	**October 1990**	**R06-00**	**R08-03**

What Changed?

The F00 R03 revision of this manual was reorganized and converted to the new Concurrent Computer Corporation's design format. Although most of the information was retained, the chapters have been rearranged to present a more logical sequence of information. The Environment for Sequential to Parallel Programming tool (E/SP), which allows you to analyze your FORTRAN program for possible parallelization, was added to the list of products supported by FORTRAN VII. Subsequently, a section was added which explains how to prepare your code for parallelization. Finally, we provide a discussion on the new compiler directives supported.

How Can I Track Changes?

Each time this manual is updated or reissued, it will be added to the chart above. This chart includes the manual printing date, functional variation (Fxx Rxx) and compiler version. It should aid in ensuring that the document and compiler version coincide. If they do not, call your local sales representative and ask for assistance.

How Are Changes Shown?

Changes to the documentation occur from one release of the compiler to the next. These changes cause the document to be reissued (the RXX number changes) or updated (the FXX number changes). The RXX and FXX numbers are located at the bottom of the page. Technical changes within the document are indicated by a vertical bar (|) shown in the page margins.

Reissues

Each time this document is reissued, it is replaced in its entirety, and the RXX number is incremented by one. (For example, if the first release of the document was R00 and the document was reissued for the next release, the RXX number would then be shown as R01.)

Updates

Update packages are issued between reissues and contain replacement and additional pages that are merged into the manual by you. Each time this document is updated, the FXX number is incremented by one. (For example, if the first update of the document occurred after the document was released at F00 R00, the Fxx number would then be shown as F01.) At the next reissue of the document, the update package is included in the document, and the document's FXX number returns to F00. (For example, if the document was updated and presently appeared as F01 R00, the next time it was reissued, the numbers would be shown as F00 R01.)

Conventions

Keywords

We use the following keyword conventions throughout this manual.

NOTE ▷ This symbol indicates information that highlights an exception or clarifies an idea or concept.

Type Style

We use the following type style conventions throughout this manual.

Bold Type

Bold type (1) shows information that you enter and (2) emphasizes a word or thought.

Italic Type

Italic type (1) cites references to other manuals, and (2) shows the information is not to be taken literally. For example, when the word *file* is used, you type the actual filename, not the word "file."

Constant Width Type

`Constant width type` (1) shows system output and (2) shows segments of code and program listings.

Syntax

We use the following syntax conventions throughout this manual.

Upper-Case Letters (A ... Z)

Upper-case letters show information that must be typed exactly as it appears; however, this information is not itself case-dependent. For example:

$TABLE

Underlining (_)

Underlining shows the minimum acceptable command abbreviation. For example:

$$\text{START, }\left[\begin{Bmatrix}\underline{\text{A}}\text{LST}\\ \text{NALST}\end{Bmatrix}\right]$$

Ellipses (... or ⋮)

Ellipses represent an indefinite number of elements or range of elements. For example:

$label_1$ [,$label_2$,...,$label_n$]

Braces ({ })

Braces enclose required parameters of which one must be chosen. For example:

$$\text{START, }\left[\begin{Bmatrix}\underline{\text{A}}\text{LST}\\ \text{NALST}\end{Bmatrix}\right]$$

Brackets ([ ])

Brackets enclose optional parameters. For example:

$$\text{START, }\left[\begin{Bmatrix}\underline{\text{A}}\text{LST}\\ \text{NALST}\end{Bmatrix}\right]$$

Commas (,)

Commas inside brackets ([,]) must be entered if you chose the optional parameter. For example:

$$\mathit{label}_1\ [,\mathit{label}_2,\ldots,\mathit{label}_n\ ]$$

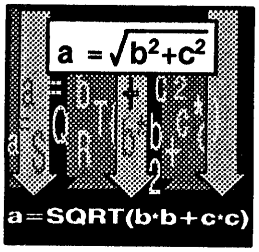
Commas outside brackets (,[ ]) must be entered whether or not you chose the optional parameter. For example:

$$\text{START, }\left[\begin{Bmatrix}\underline{\text{A}}\text{LST}\\ \text{NALST}\end{Bmatrix}\right]$$

Shading (▨)

Shading identifies default options. In the case shown below, 19 is the default.

$$\underline{\text{CONT}}\text{IN=}\begin{Bmatrix}n\\ \colorbox{lightgray}{19}\end{Bmatrix}$$



1

FORTRAN VII Overview

In this chapter

We present an overview on the FORTRAN VII compilers. In addition, we describe other related products that you may use with the compilers and the minimum requirements for maintaining the system.

Topics include:

- FORTRAN VII compilers
- FORTRAN VII support products
- Minimum system requirements

The FORTRAN VII Compilers

Optimum FORTRAN programming environment can be achieved by taking advantage of the FORTRAN VII F7O and F7Z compilers. The FORTRAN VII compilers modify and rearrange the source code during compilation to provide run-time efficiency. Coupled with the processing capabilities of the Series 3200 Processors, the optimizing compilers allow the development programmer to produce highly optimized code at the least possible cost.

To take advantage of faster compilation provided by the F7O and F7Z compilers, the NOPTIMIZE directive is available to turn off the optimizing capabilities of both compilers.

The FORTRAN VII compilers provide two levels of optimization. Global optimization, available on the F7O and F7Z compilers, optimizes individual program units using optimization techniques such as common subexpression elimination, invariant code motion, strength reduction of arithmetic operations, and loop test replacement.

Universal optimization, available on the F7Z compiler, goes one step further. F7Z optimizes code across program unit boundaries by incorporating subprogram code within the main program structure at the request of the user via compiler directives. As a result of greater emphasis on the use of structured programming, programs are gaining an increasing number of units, resulting in an increased amount of execution time spent in subprogram linkage. F7Z eliminates the linkage penalty while retaining the advantages of modular design.

The optimizations performed by the optimizing compilers are discussed in Chapter 4. A more detailed discussion on the optimizing compilers are presented in Chapter 15.

FORTRAN VII Support Products

Concurrent Computer Corporation (Concurrent) offers a number of products that can enhance the performance of the FORTRAN VII compilers and increase program development capabilities. These products include:

- FORTRAN VII Run-Time Library (RTL) - These software routines can be used to:
  - Manipulate strings

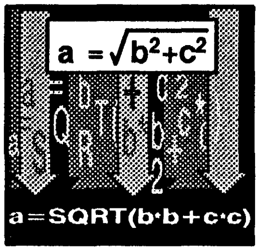
- — Perform basic input/output (I/O) functions

- OS/32 System Support RTL - This package provides the software routines that can be used to:
  - — Generate and handle task trap operations
  - — Perform analog - digital conversions
  - — Allow one program to control execution of another program
  - — Send messages from one program to another
  - — Create and delete files
  - — Handle timer expiration traps
  - — Control any 3200MPS Family of Processors
- System Mathematical RTL - This package supplies mathematical functions, which is a superset of the intrinsic functions required by X3.9-1978 American National Standard Institute (ANSI) FORTRAN.
- FORTRAN VII Enhancement Package (FEP) - FEP is a combined hardware/software package that provides an additional amount of writable control store (WCS) memory that can be loaded with microcode routines designed to enhance the performance of the compiler and the RTL math functions. These routines increase compilation rates for the F7O and F7Z compilers by 40% and improve the performance of certain assembly language RTL routines by 25% over the standard RTL.
- FORTRAN/RELIANCE™ Interface - This software consists of a set of subroutines that allow the FORTRAN programmer to access all the features of the integrated transaction controller (ITC) and the data management system (DMS) of Reliance.
- Common Assembly Language (CAL/32) Assembler - CAL/32 converts CAL output from the FORTRAN optimizing compilers into the Concurrent 32-bit object code. The F7O and F7Z compilers optionally produce CAL output.
- E/SP - The Environment for Sequential to Parallel Programming tool which allows you to analyze your FORTRAN program for possible parallelization. The analysis is done using a graphical interface that runs on a workstation.

Reliance is a trademark of Concurrent Computer Corporation.

Minimum System Requirements for FORTRAN VII

The minimum system that supports the features of the R06 release of FORTRAN VII is a Concurrent 32-bit processor running under OS/32 R08-03 or higher. A minimum of 1MB of total memory and one disk drive with at least 25MB available memory space is required on any Series 3200 Processor supporting the F7O and F7Z compilers.



2

Overview of the Program Development Process

In this chapter

We introduce you to the program development process to create, compile, link, execute, and debug your FORTRAN program. The program development commands that allow you to perform most of these are command substitution system (CSS) files maintained on the system account. To fully understand how these commands work, a review of the basic CSS features is presented.

Topics include:

- Description of the program development phases
- Review of the CSS
- Entering the FORTRAN VII environment
- Use of the basic program development commands to compile, link, and execute programs
- Description of the debugging phase

Program Development Process Phases

The program development process is divided into five major phases, as illustrated in the flowchart of Figure 2-1. These phases are Programming, Compiling, Linking, Executing, and Debugging.

Programming is the initial step of the development process. In this phase, you combine all your programming skills and your familiarity with the FORTRAN VII language to produce the source code. Awareness of certain guidelines applicable to the Concurrent environment will aid you in preparing efficient and valid programs. You must also be familiar with a text editor such as OS/32 Edit. The output of this phase is a complete FORTRAN program.

In the Compiling phase, you take the source program created during the Programming phase and input that source to the compiler. The compiler analyzes your source code, outputs a source listing, checks for compilation errors, and outputs the object code if no errors occurred.

Linking comes after a successful compilation process. The Link process converts the object code into task a image, outputs a Link map, checks for Link errors, and creates a task image file if no errors occurred.

Executing is probably the last phase of the development process if you achieve a successful run of the program. This means getting the required results from your code. Otherwise, you have to modify your program to produce the desired results. In this phase, you load the task image, assign any required logical units, and start the task. After completion, you get your task output.

The Debugging phase can come after any of the three previous phases, whenever errors are encountered in the Compiling, Linking, or Executing steps. You normally make use of program listings to perform this step and/or other sophisticated debugging aids in the case of run-time errors. After debugging, your source program may require some modifications and must go through the whole process again.

In the succeeding sections, you will have a glimpse of the individual steps of the development process. Program development commands are available to perform most of these phases. You will have a more detailed discussion of the development process in the succeeding chapters.

NOTE The program development commands used in this chapter are system CSS files. Check with your system administrator to determine whether they were altered. If so, the following documentation will be inconsistent.

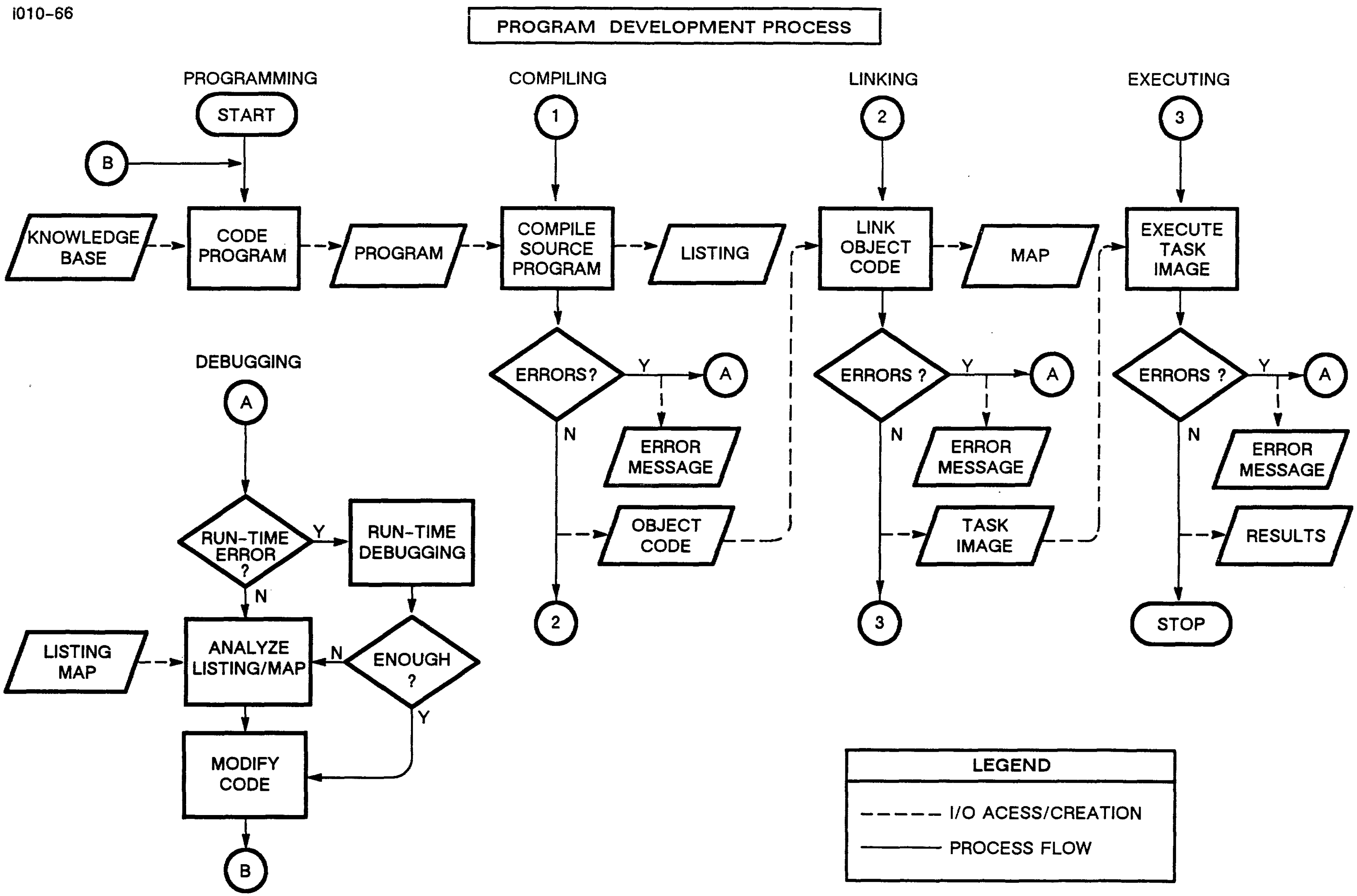


Figure 2-1. Program Development Flowchart

Review of the CSS

This section covers some of the basic features of the CSS sufficient to guide you through the development process presented in this chapter. For the more elaborate features of the CSS, see the *Multi-Terminal Monitor (MTM) Reference Manual.* You must be familiar with these concepts when reading the chapters dealing with building command files to compile, link, and execute your program.

The CSS allows you to write and store an operating system procedure to a file. Once created, the procedure can be executed by invoking the filename like an MTM command.

Consider the file CREATE.CSS containing the following:

```
1  >XALLOCATE TEMP.CMD,IN,80
2  >$wr FILE CREATED
3  >$EXIT
```

This file has the OS/32 XALLOCATE command that creates an indexed file named TEMP.CMD. Line 2 contains the CSS command $wr which echoes the string FILE CREATED onto the screen. The command $EXIT ends the CSS procedure. To execute this procedure, simply enter the filename with or without the .CSS extension.

```
*CREATE
FILE CREATED

*
```

To verify the creation of the TEMP.CMD file, invoke the OS/32 DISPLAY FILE command.

If you name the procedure file CREATE.ANY instead of CREATE.CSS, you must specify the full filename when you invoke it.

```
*CREATE.ANY
FILE CREATED

*
```

A CSS procedure can be written such that it allows you flexibility when you invoke it. This is possible through parameter substitution. The use of parameter substitution adds flexibility to any CSS procedure since the value of the parameter, denoted by the *@n*, can be specified when the CSS is called. You can modify the CREATE.CSS file to XALLOCATE any file that you specify.

```
>XALLOCATE @1,IN,80
>$wr FILE @1 CREATED
>$EXIT
```

In this example, @1 is the first positional parameter established by CREATE.CSS. When invoking CREATE.CSS, you need to specify a filename to satisfy this parameter.

```
*CREATE TEMP.NEW
FILE TEMP.NEW CREATED

*
```

A CSS file can have any number of positional parameters.

Whatever is entered at a positional parameter is automatically inserted wherever the parameter appears in the procedure. The first parameter is placed at @1, the second at @2, the third at @3, etc.

Positional parameters must be separated by commas when invoking the CSS.

You can also use a predetermined number of variables and a variety of commands within your CSS. See the *Multi-Terminal Monitor (MTM) Reference Manual* for a complete discussion on these topics.

Entering the FORTRAN VII Environment

The initial step of the development process is Programming. You are expected to produce a program source making use of your FORTRAN knowledge and your familiarity with some guidelines that are applicable to the FORTRAN VII environment.

You must initially sign on to the MTM environment. MTM gives you access to the program development environment in two ways: using the FORT or the LANGUAGE commands. Their syntaxes are as follows:

$$\text{FORT}\left[\begin{Bmatrix}\text{O}\\ \text{Z}\end{Bmatrix}\right]\left[\textit{voln:filename}\right]$$

$$\text{LANGUAGE}\left[\text{FORT}\left[\begin{Bmatrix}\text{O}\\ \text{Z}\end{Bmatrix}\right]\right]$$

Where:

FORT initializes the program development environment for the optimizing compiler with no optimization (NOPT directive) enforced. This environment is the equivalent of the development compiler of R05-05 and earlier. If the O or Z character is appended with no space in between (FORTO or FORTZ), the program environment is initialized for the F7O and F7Z compilers (optimizer is on by default), respectively. If you specify a filename to this command, one of two things can happen:

- If the specified filename exists, this filename becomes the current file on which any program development commands may apply; or
- If the specified filename does not exist, the FORT command automatically activates OS/32 Edit to allow you to create the file. Once created, that file becomes the current file on which any program development command may apply.

The current file can be changed by issuing another FORT command with a different filename.

LANGUAGE initializes the program development environment for the F7O with the optimizer disabled (NOPT directive), F7O, or F7Z compilers. Without any parameter, this command returns the current environment.

Example:

```
*FORT FILENAME
*
*  New Language Environment -- Fortran VII O R06
*
*
*Editing new file -- FILENAME.FTN  (APPEND mode set)
*
Concurrent Computer Corp OS/32 EDIT32 03-145 Rxx-yy
OPTION TAB=,7,73;OPTION INPLACE=OFF
GET FILENAME.FTN;OPTION COM=CON:;AP
    1   >
```

As seen in the previous example, the system automatically attaches the extension .FTN to the filename. FILENAME.FTN is used by the system to identify the source program throughout the programming session. FORT automatically activates the OS/32 Edit software, sets the append mode and sets the backslash character (\) as the tab character for columns 7 and 73.

Creating Your Source Program

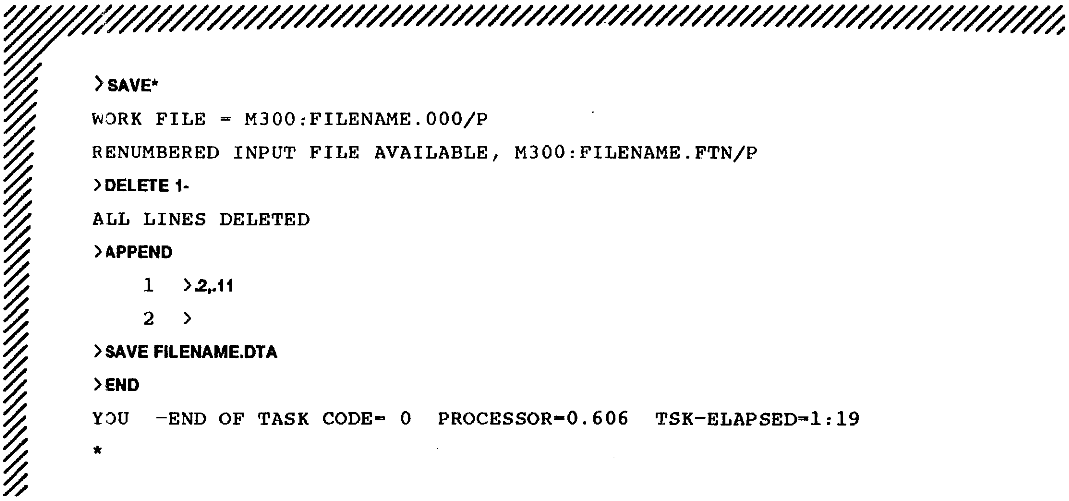
Continuing from the previous section, you can now create the source program and data files using the OS/32 Edit commands. Enter the source code as follows:

```
    1  >\READ(*,10) R,Y
    2  >\WRITE (*,20) R,Y
    3  >\H=R*COS(R)**4/(2*Y)
    4  >\WRITE(*,30) H
    5  >10\FORMAT(F2.1,X,F3.2)
    6  >20\FORMAT(1X,'R = ',F3.1,'  Y = ',F4.2)
    7  >31\FORMAT(1X,'THE VALUE OF H =',F4.2)
    8  >\STOP
    9  >\END
   10  >
>
```

Note that an error appears in the code listed above. The statement label of the format statement in line seven (31) does not match the statement label value called in line four (30). This causes the compiler to detect an error. This error was included intentionally to illustrate how the compiler output listing can be used to troubleshoot your source code. See "Checking the Compiler Listing" later in this chapter.

Creating a Data File

To create a data file, save the source program file to a disk and clear the edit buffer by deleting all lines currently in the buffer.



```
>SAVE*
WORK FILE = M300:FILENAME.000/P
RENUMBERED INPUT FILE AVAILABLE, M300:FILENAME.FTN/P
>DELETE 1-
ALL LINES DELETED
>APPEND
    1  >2,.11
    2  >
>SAVE FILENAME.DTA
>END
YOU  -END OF TASK CODE= 0  PROCESSOR=0.606  TSK-ELAPSED=1:19
*
```

In the previous example, FILENAME.FTN is saved and then cleared from the edit buffer. The edit APPEND command allows data to be entered in the data file. The data file is saved and the edit session is terminated with the END command.

See the *OS/32 Edit User Guide* for more information on the OS/32 Edit commands.

Assigning Logical Units

The current program is now ready for the program development commands. However, before using these commands, make certain that the default device assignments set at system generation (sysgen) are appropriate. The program development environment defines and sets global variables that are associated with particular devices. These devices have default logical unit (lu) assignments. The global variable names and their default settings are displayed when the user signs on to MTM. Table 2-1 shows the variable names, their default settings and lu assignments.

Variable Name	Device	lu
SSYSIN	CON:	1
SSYSOUT	CON:	2
SSYSPRT	PR:	3
SSYSCOM	CON:	5
SSYSMSG	CON:	7
SSYSLST	CON:	8

Table 2-1. Program Development Default Variable Settings and lu Assignments

To change any of the default device assignments, enter the appropriate variable followed by the new device assignment. For example, to change the input device SSYSIN from the terminal (CON:) to FILENAME.DTA, type:

```
SSYSIN FILENAME.DTA
```

If listings are to be sent directly to the terminal rather than the printer, type:

```
SSYSPRT CON:
```

These assignments cause data to be read from FILENAME.DTA and listings to be sent to the terminal.

You can now use the program development commands to compile, link, and execute your program.

Using Program Development Commands

Once you have created your source program, you can invoke the program development commands.

COMPILE Command

The COMPILE command compiles a source file as shown in the following example.

Example:

```
*COMPILE FILENAME
FORTRAN-VII R06-00.00
.MAIN         1 ERROR(S)    TABLE SPACE USED:      1 K
YOU  -END OF TASK CODE=  4    PROCESSOR=0.035  TSK-ELAPSED=0
```

Notice that the compiler detected an error which resulted in an end of task equal to 4. Refer to the compiler listing to determine the exact error.

The compiler output listing is directed to the device specified by SSYSPRT, which, in this case, is PR:. Thus, the compiler listing is printed by the device designated by PR:.

Compiler start options (called directives) may be passed on to the COMPILE command. See Chapter 6 for details on the compilation process.

Checking the Compiler Listing

From the listing generated, you can determine the compilation errors that occurred. A partial listing of the sample program is presented as follows:

```
 7  0000D8I  31   FORMAT(1X,'THE VALUE OF H =',F4.2)
 8  0000F8I       STOP
 9  000100I       END
ERROR # 300  ******************************************
        >>>  UNDEFINED LABEL
             30
WARNING # 300 *****************************************
             UNREFERENCED LABEL
             31
```

This listing indicates that an undefined label exists. To correct the error, return to the editor by entering EDIT at the prompt. The label for line 7 should be 30 instead of 31.

See Chapter 10 for a comprehensive description of compiler listings.

Modifying a Program

Use the OS/32 Edit command to make the necessary corrections. This command automatically makes FILENAME.FTN the current file and gets it for editing.

Example:

```
*EDIT
Concurrent Computer Corp OS/32 EDIT32 03-145 R08-02
OPTION TAB=,7,73;OPTION INPLACE=OFF
GET FILENAME.FTN;OPTION COM=CON:;SC
    1          READ(*,10) R,Y
    2          WRITE (*,20) R,Y
    3          H=R*COS(R)**4/(2*Y)
    4          WRITE(*,30) H
    5     10   FORMAT(F2.1,X,F3.2)
    6     20   FORMAT(1X,'R = ',F3.1,' Y = ',F4.2)
    7     31   FORMAT(1X,'THE VALUE OF H =',F4.2)
    8          STOP
    9          END
UNABLE TO TYPE FULL SCREEN
>AL 7 <carriage return>
    7     31   FORMAT(1X,'THE VALUE OF H =',F4.2)
    7   >  0 <carriage return>
    7     30   FORMAT(1X,'THE VALUE OF H =',F4.2)
    7   > <carriage return>
>S* <carriage return>
WORK FILE = M300:FILENAME.000/P
RENUMBERED INPUT FILE AVAILABLE, M300:FILENAME.FTN/P
>END <carriage return>
YOU    -END OF TASK CODE= 0  PROCESSOR=0.606  TSK-ELAPSED=1:19
*
```

To edit a source file other than the one just compiled, type the following:

***EDIT TEST.FTN**

This command automatically makes TEST.FTN the current file and gets it for editing.

Reissue the COMPILE command and your modified program should compile successfully. This process creates the object module contained in the file FILENAME.OBJ.

LINK Command

The LINK command links the object module to produce the task image in the FORTRAN environment. If no object module exists, the LINK command causes the source module to be compiled to yield the object module. Link does not date check, load, or execute a program.

Example:

```
*LINK FILENAME
Concurrent Computer Corp OS/32 LINKAGE EDITOR 03-242 Rxx-yy
YOU  -END OF TASK CODE= 0  PROCESSOR= 0.74   TSK-ELAPSED=2
```

This process generates a link map which is printed by the device designated by PR:. See Chapter 10 for a complete description of the link map.

Link options may be passed to the LINK command. See Chapter 7 for details on the linking process.

A successful link of the program creates the task image contained in the file FILENAME.IMG.

COMPLINK Command

The COMPLINK command compiles and links a program in one step. COMPLINK conditionally compiles and links by date checking the source, object, and task image files in the FORTRAN VII environment. This command does not execute the program. If compilation is required and there is an error, the process ends with a nonzero end of task code, the link procedure is not initiated and the process is aborted.

Example:

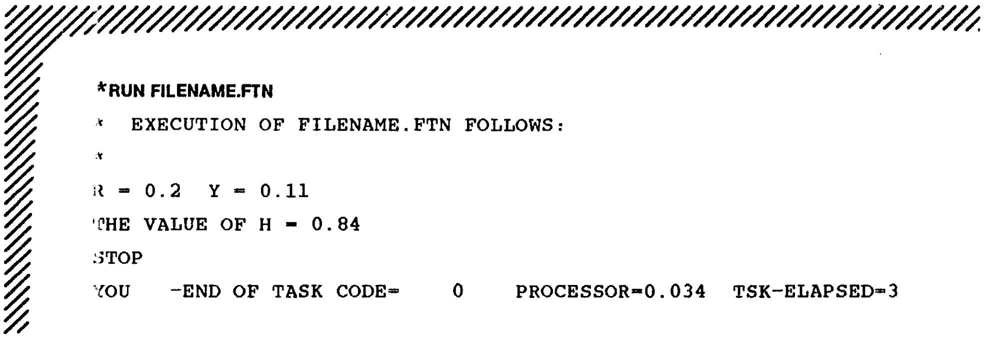
```
*COMPLINK FILENAME
```

The output of the COMPLINK command is the same as COMPILE and LINK except that only one command is issued. If a compile is required, the compiler listing is output to the designated output device and the object file is saved in the appropriate filename with the extension .OBJ. After a successful compile, the link sequence is automatically initiated.

RUN Command

The RUN command loads and runs the task image in the FORTRAN environment. This command does not date check, compile, or link. This command is used to execute a task only.

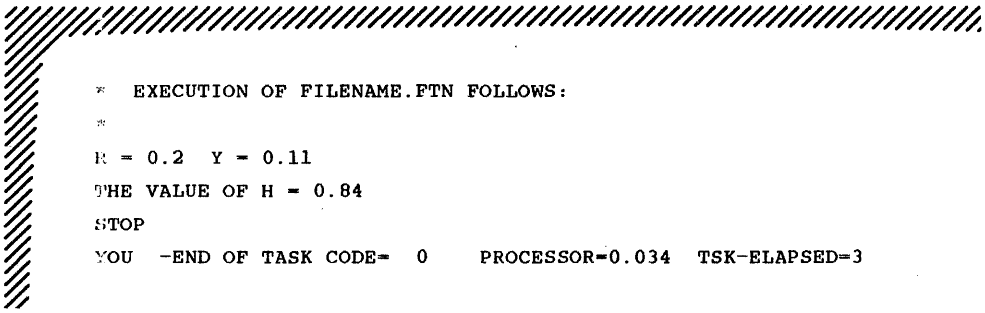
Example:



```
*RUN FILENAME.FTN
*  EXECUTION OF FILENAME.FTN FOLLOWS:
*
R = 0.2  Y = 0.11
THE VALUE OF H = 0.84
STOP
YOU   -END OF TASK CODE=    0    PROCESSOR=0.034  TSK-ELAPSED=3
```

EXEC Command

EXEC compiles, links, and executes the program in FILENAME.FTN to completion, yielding the following results:



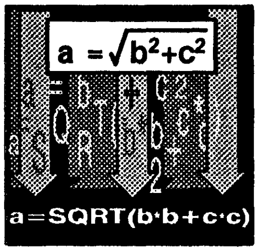
```
*  EXECUTION OF FILENAME.FTN FOLLOWS:
*
R = 0.2  Y = 0.11
THE VALUE OF H = 0.84
STOP
YOU  -END OF TASK CODE=  0    PROCESSOR=0.034  TSK-ELAPSED=3
```

A successful compilation ends with a zero end of task code. An end of task code other than zero indicates a compilation error. See Chapter 6 for an explanation of end of task codes. In the previous example, the end of task code is 4 which indicates that errors were encountered during compilation. The EXEC command does not proceed with the linking and execution processes if compilation errors occur.

The EXEC command recompiles the source program that was changed. If the source code was not changed, it is not compiled again. This is made possible by date checking. After a program development command is entered, the ease of use (EOU) command procedure checks the date and time that the file was last modified or created. Based on this information, the appropriate process is performed. For example, when the EXEC command is entered, the EOU first checks the last date and time that source file was modified. If the .FTN file is newer than the .OBJ file, this is interpreted to mean that the source file was modified since the last compile and that a recompilation is necessary before processing can continue.

Debugging Phase

A big percentage of the development process might be devoted to debugging your program. Rarely would you be able to code your program, compile, link, and execute it successfully without encountering errors along the way. This is especially true for large programs. The debugging process may pertain to a simple look at the compiler listing when a compiler error occurs or to an actual trace of the program execution when the program returns unexpected results. Thus, you should be well versed in reading the compiler listings and link maps that are generated by the compilation and linking processes, respectively. In addition, several compiler directives and a set of run-time library (RTL) routines are available to support run-time debugging. See Chapters 9 and 10 for a more detailed discussion on debugging.



3

Controlling Compilation Through Directives

In this chapter

We introduce you to the different compiler directives that allow you to control the compilation process. You can use most of these directives as options to the START command or you can embed them in your FORTRAN program. Some directives can only be embedded in the source program. A complete summary of all directives and how each can be used are presented in tabular form.

Topics include:

- Introducing the two types of directives
- Controlling compiler input
- Controlling compiler list output
- Inserting Common Assembly Language (CAL) clocks
- Controlling compiler optimization
- Controlling in-line expansion
- Debugging the source code
- Preparing your code for parallelization
- Using other instream directives

Introducing the Two Types of Directives

If you initiate compilation using the program development commands EXEC, COMPILE, and COMPLINK, the compiler will:

- Perform a batch compilation on all program units submitted,
- Allow up to 19 continuation lines per statement,
- Output 60 lines per page at 132 columns per line,
- Produce a complete source listing including all program statistics, compilation errors, and warning messages,
- Assign the name .MAIN to a main program unit that was not named by the PROGRAM statement,
- Refer to all subprograms by the name given to them in the source program,
- Title each page of a source listing with the first statement of the program module,
- Generate segmented object code, and
- Perform all global optimizations and send a summary of all optimizations to the device or file designated by SSYSPRT.

The FORTRAN VII compilers are not limited to the operations provided by the program development commands. Other commands, called compiler directives, can be used to modify or add to these operations. When compiler directives are inserted in the source code, they are referred to as instream directives. Instream directives allow the user to:

- Control compiler input,
- Determine what listings should be output to the list device and how they should be formatted,
- Insert assembler code within the FORTRAN source,
- Control the optimization capabilities of the compilers,
- Control certain compiler functions, and
- Debug the program.

Compiler directives can also be specified in command substitution system (CSS) procedures that are used to compile FORTRAN programs.

Example:

```
*COMPILE PROGNAME.FTN, COMP INFORM
```

When used in this manner, compiler directives are referred to as start directives (also referred to as start options). Start directives are discussed in Chapter 6.

A summary of all directives appears on the following pages. The "x" in the start column indicates that the directive may also be specified as a start directive. The syntax presented below is strictly for instream directives. For the correct start directive syntax, see Chapter 6, "Using the Compiler Start Directives."

The optional N ([N]) preceding some directives is used to negate the effect of the directive. The definitions for these directives apply to their use without the N.

DIRECTIVES			
	Instream	**Start**	**Definition**
Compiler Input	$[N]BABORT	x	Aborting batch compilation in case of error.
	$[N]BATCH	x	Batch compilation feature.
	$BEND		Indicates end of source input.
	$INCLUDE		Include source from specified file or device.

	DIRECTIVES		
	Instream	**Start**	**Definition**
Compiler Output	$[N]ALST	x	Produce a CAL listing after successful compilation.
	$EJECT		List subsequent source on the next page of current listing.
	$[N]ELIST	x*	Output an extended listing for in-line expanded subprograms. The default depends on $[N]LIST. If $LIST is specified and $NELIST is not, $ELIST is in effect.
	$[N]INFORM†	x	Output optimization messages. The default depends on $NLIST. If $LIST is specified and $NINFORM is not, $INFORM is in effect.
	$LCNT *n*	x	Specifies number of lines per page. The default value is 60.
	$[N]LIST	x	Output a source listing.
	$TITLE		Specify a title for the source listing. If $TITLE is not specified, the compilers print the first line of the program as a title for each page.
	$[N]WARN	x	Output warning messages to the list device.
	$WIDTH		Specify the maximum width of a line in the listing. The default is 131.
	$[N]XREF	x	Generate a cross-reference listing of labels and identifiers.

* For the F7Z compiler only.

† See discussion of in-line expansion directive later in this chapter fo default values.

	DIRECTIVES		
	Instream	**Start**	**Definition**
CAL Blocks	$ASSM		Indicates the beginning of an embedded CAL block.
	$FORT		Indicates the end of the CAL block.
	$GOES		Lists the labels of FORTRAN statements to which CAL code branches.
	$REGS		Indicates the registers modified by the CAL block.
	$SETS		Informs compiler which variables are modified inside the CAL block.
	$USES		Informs compiler which variables are used in the CAL block.
Optimization	$[N]BASE	x	Makes base addresses candidates for global registration allocation.
	$[N]OPTIMIZE	x	Switch for global optimization.
	$[N]TCOM	x	Declares specified common blocks, common or global entities as sharable.
In-line Expansions†	$DISTINCT		Identifies CAL symbols to be replaced by unique compiler generated symbols.
	$INLIB	x	Specifies a source file to be searched for in-line expansion.
	$INLINE *name*	x*	In-line expansion of a subprogram.
	$INSKIP		Inhibits separate compilation.

* For the F7Z compiler only.

† See discussion of in-line expansion directives later in this chapter for default values.

DIRECTIVES			
	Instream	**Start**	**Definition**
Debugging	$[N]COMP	x	Conditional compilation of debugging statements flagged by an X in the first column.
	$[N]TEST	x	Check array subscripts and substrings against their declared bounds.
	$[N]TRACE	x	Trace the value of a variable and/or labeled statements.
E/SP tool	$[N]OBJ	x	Controls generation of object code for E/SP. The default is $NOBJ if $TABLES is specified, otherwise, the default is $OBJ.
	$[N]OVERLAP	x	Aliasing of dummy arguments.
	$[N]SAFE	x	Embedded assembly code can be parallelized. The default is $SAFE if the program has no embedded assembly code and $NSAFE otherwise.
	$[N]TABLES	x	Dump tables for E/SP. The default is $TABLES not in effect.
	$[N]XFORT	x	Contains special E/SP-generated, nonFORTRAN constructs. The default is $XFORT not in effect.
Miscellaneous	$[N]APU	x	Generates informative messages for supervisor call (SVC) instructions.

DIRECTIVES			
	Instream	**Start**	**Definition**
Miscellaneous (Cont.)	$[N]CAL	x	Generates assembly language code instead of object code.
	$DCMD	x	Embed Link commands in object code.
	$DP	x	Treat all REAL and COMPLEX variables with non-explicit lengths as double precision variables. The default is the type associated with FORTRAN identifiers and constants.
	$[N]F66DO	x	All DO loops executed at least once.
	$[N]HOLL	x	Treats all quoted strings used as subprogram arguments as Hollerith constants.
	$IBYTE	x	Treat BYTE statement as INTEGER*1 statement.
	$INT2	x	Treat all integer variables with non-explicit lengths as INT*2. Not specifying this directive gives you the usual typing associated with FORTRAN identifiers and constants.
	$LBYTE	x	Treat BYTE statement as LOGICAL*1 statement.
	$LTORBIT	x	Count bit positions from left to right.

DIRECTIVES			
	Instream	**Start**	**Definition**
Miscellaneous (Cont.)	$PASSBYADDRESS	x	All scalar arguments to subprogram are passed by address.
	$PAUSE		Suspends compilation at the point where this directive is encountered.
	$PROG		Changes the name of a program unit. If $PROG is not specified, the compiler uses the main program unit.
	$[N]REENTRANT	x	Generate reentrant code. The default is $NREENTRANT.
	$RTOLBIT	x	Count bit positions from right to left. $LTOLBIT is the default.
	$[N]SEG	x	Generates segmented object code.
	$[N]SYNTAX	x	Performs syntax check of source without generating an object code.
	$TARGET *n*	x	Generates machine code optimized to the instruction set of the specified processor. If the directive is not specified, the compiler will output machine code targeted to the processor on which the compiler is running.

DIRECTIVES			
	Instream	**Start**	**Definition**
Miscellaneous (Cont.)	$[N]TRANSCENDENTAL	x	Generates 3280 transcendentals. If NTRANSCENDENTAL is specified, run-time library (RTL) calls are generated. The default is $TRANSCENDENTAL for a 328*x* processor and $NTRANSCENDENTAL for any other Series 3200 Processor.
	$[N]UNNORMALIZE	x	Generates unnormalized floating point load instructions. The default is $UNNORMALIZE for the Model 3203, 3205, and 3280, and $NNUNORMALIZE otherwise.

Notes on Using the Two Types of Directives

Take note of the following guidelines when using either of the directives:

- Because some directives may be both an instream and a start directive, instream directives override the values specified as start options, except that:
  - The NINLINE start option disables all $INLINE directives.
  - The ELIST and NELIST start options disable all $ELIST and $NELIST directives.
  - The APU and NAPU start options disable all $APU and $NAPU directives.
  - The UNNORMALIZE and NUNNORMALIZE start options disable all $UNNORMALIZE and $NUNNORMALIZE directives.
  - The TRANSCENDENTAL and NTRANSCENDENTAL start options disable all $TRANSCENDENTAL and $NTRANSCENDENTAL directives.
  - Specifying a processor that does not support unnormalized floating point loads in a TARGET start option or $TARGET directive forces the NUNNORMALIZE option.
  - Specification of a processor which does not support transcendental operations in a TARGET start option or $TARGET directive will force the NTRANSCENDENTAL option.
- Blanks surrounding the equal sign (=) within start options are not allowed (i.e., TARGET=3280 is allowed but not TARGET = 3280). Equal signs are not allowed between an instream directive name and the value, i.e., $TARGET=3280 is not valid.
- Shortened forms of start options are valid, but instream directives cannot be abbreviated.
- Errors in start options suppress the compilation; errors in instream result in warnings and do not terminate compilation.

How to Use the Instream Compiler Directives

An instream directive is specified by placing a dollar sign ($) in column one and providing the text of the directive in the following columns. Directive text cannot exceed 72 characters; a semicolon (;) and a comment may optionally follow the directive text. An instream directive cannot be continued to the following line and must not be placed between the initial line of a statement and any of its continuation lines. With this exception, most instream directives can be inserted anywhere in the source code. Some, however, must be placed before the first FORTRAN statement in the module to be effective.

Example:

```
C THIS PROGRAM SEGMENT CONTAINS COMPILER DIRECTIVES
C
$ALST
$XREF
$WIDTH 70
$TRACE J
      M =K+1
      DO   10 I=1,3
      J =I
      WRITE (6,1000) J,I
10    CONTINUE

$NTRACE
            .
            .
            .
      END
```

Directives that begin with $N are used to deactivate previously specified directives or override a positive default. In the previous example, $NTRACE deactivates $TRACE J for all code following $NTRACE. Some directives, such as $WIDTH, operate only on the code immediately following them. Other directives, such as $XREF, operate on the entire program unit. Their effect is completely neutralized over the entire unit if their $N counterpart is specified in the same program unit. The very last directive specified takes precedence over its counterpart directive.

Example:

```
C THIS EXAMPLE HAS BOTH $BABORT AND $NBABORT
C IN   .MAIN
C
$BABORT
      T = 1
      CALL SUBA
      CALL SUBB
      STOP
$NBABORT
      END
```

In this example, $NBABORT is in effect during the entire compilation. If their order is reversed (i.e., $NBABORT is specified first before $BABORT), then, $BABORT takes effect during the entire compilation.

The directives which act upon the entire program unit are:

$ALST/$NALST
$APU/$NAPU
$BABORT/$NBABORT
$BASE/$NBASE
$BATCH/$NBATCH
$BEND
$CAL/$NCAL
$DP
$ELIST/$NELIST
$F66DO/$NF66DO
$IBYTE
$INFORM/$NINFORM
$INLIB
$INT2
$LBYTE
$LTORBIT
$OPTIMIZE/$NOPTIMIZE
$PASSBYADDRESS
$REENTRANT/$NREENTRANT
$RTOLBIT
$SEG/$NSEG
$SYNTAX/$NSYNTAX
$TARGET
$TCOM
$TRANSCENDENTAL/$NTRANSCENDENTAL
$UNNORMALIZE/$NUNNORMALIZE
$WARN/$NWARN
$XREF/$NXREF

The rest of the directives affect only the block of code in which they are embedded.

The following sections discuss each instream directive and its effect on compilation.

Controlling Compiler Input

The following directives can be used to control how the source is input to the compiler:

```
$BABORT/$NBABORT
$BATCH/$NBATCH
$BEND
$INCLUDE
```

```
$BABORT/$NBABORT
```

$BABORT aborts a batch compilation if the program unit in which it appears has a compilation error. $NBABORT turns off the $BABORT feature. $NBABORT is the default.

```
$BATCH/$NBATCH
```

$BATCH turns on the batch compilation feature. $NBATCH turns off the batch compilation feature after compilation of the program unit in which $NBATCH appears is completed. $BATCH is the default.

```
$BEND
```

$BEND indicates the end of source input to the compiler.

```
$INCLUDE
```

$INCLUDE allows the user to switch input from one file or device to an alternate file or device. The format of $INCLUDE is:

$$\text{\$INCLUDE} \left[\begin{Bmatrix} \mathit{lu} \\ \mathit{lu,\ fd} \\ \mathit{fd} \end{Bmatrix}\right] \left[\mathit{,label\ range}\right] \left[\mathit{(options)}\right]$$

Where:

lu is a logical unit number from 9 to 15.

fd is the file descriptor of the file or device that contains the source code to be included. This file descriptor follows the OS/32 file naming convention.

label range indicates the range of source code to be incorporated from the include file. It can be specified in any one of the following forms.

**n1 causes the compiler to search for the module delimiter **n1 in the include file and include the source code following that module delimiter until the first module terminator. If **n1 is not found, a warning message is output and the directive is ignored.

**n1- causes the compiler to search for the module delimiter **n1 in the include file and incorporate the source code following that delimiter up to the end of file (EOF). If **n1 is not found, a warning message is output and the directive is ignored.

n1-n2
causes the compiler to search for the module delimiter **n1 in the include file and include the source code following that delimiter until the module terminator for **n2. If **n2 is not encountered before the EOF, the compiler unconditionally terminates $INCLUDE.

-**n2 causes the compiler to include source code from the current position on the file until the module terminator for **n2 is encountered. If **n2 is not encountered before the EOF, the compiler unconditionally terminates $INCLUDE.

— causes the compiler to include source code from the current position on the file until the EOF.

Not specifying label range causes the compiler to include source code from the current position on the file until the next module terminator.

A module delimiter has the form ***n* (starting in column 1). *n* is an alphanumeric string with no embedded blanks whose length may not exceed eight characters. A module terminator is either a /*, an END statement, or an end of file.

options indicates one or more of the following options specified in any order.

NEND This option tells the compiler that the END statement encountered in an included file is only a module terminator and not a program terminator. If NEND is not specified, an END statement encountered in an included file is treated as the end of the FORTRAN program.

NLIST This option prevents the compiler from outputting the included statements to the source listing. NLIST applies to the current $INCLUDE and all subsequent nested levels of $INCLUDE. If NLIST is not specified, the compiler will produce a listing of the included statements.

REW This option causes the compiler to rewind the file or lu before including the source from it.

Successive modules in an included file are separately compiled if the following four conditions exist.

- Label range specifies more than 1 module (i.e., **A-**B).
- The END statement is used as a module terminator for each of the modules.
- NEND option is not specified on the $INCLUDE directive.
- NBATCH is not in effect in any of the modules contained within the label range.

The following example illustrates the use of the different label range specifications.

Examples:

Consider the file INCFILE.FTN which contains the following code:

```
**RANGE1
       INTEGER*2 A
       .
       .
       .
       END
       .
       .
       .
**RANGE2
       COMPLEX B
       .
       .
       .
/*
(end of file)
```

$INCLUDE INCFILE.FTN,RANGE1**

This causes the compiler to search for the module delimiter **RANGE1 in INCFILE.FTN and include the code following **RANGE1 until the first module terminator END.

$INCLUDE INCFILE.FTN,RANGE1-**

This causes the compiler to search for the module delimiter **RANGE1 in INCFILE.FTN and incorporate the code following **RANGE1 up to the end of file.

$INCLUDE INCFILE.FTN,RANGE1-**RANGE2**

This causes the compiler to search for **RANGE1 in INCFILE.FTN and include the code following **RANGE1 until the module terminator for **RANGE2 which, in this case, is /*.

$INCLUDE INCFILE.FTN,-RANGE2**

This causes the compiler to include code from the current position of the file (in this case, the beginning of INCFILE.FTN) until the module terminator for **RANGE2 which is /*.

$INCLUDE INCFILE.FTN,-

This causes the compiler to include the source code contained in INCFILE.FTN.

The effect of $INCLUDE depends on how the arguments *lu* and *fd* are specified. The four distinct formats resulting from this are described below:

Format 1:

$INCLUDE *fd* [,*label range*] [(*options*)]

The compiler assigns a free lu to the file specified by *fd* and starts including the source.

Example:

```
C   THE $INCLUDE DIRECTIVE INCORPORATES
C   CODE FROM SUBFILE.FTN
C
$INCLUDE SUBFILE.FTN,**FINDB
      WRITE (6,*)B
      STOP
      END
```

The file SUBFILE.FTN contains the following code:

```
**FINDA
      GLOBAL A
      END
      .
      .
      .
**FINDB
      GLOBAL B
/*
```

The $INCLUDE directive in the preceding example causes the compiler to assign a free lu to SUBFILE.FTN, search for the program delimiter **FINDB and include source code following it up to /*.

Format 2:

$INCLUDE *lu,fd* [,*label range*] [(*options*)]

This directive causes the compiler to close the lu, assign it to the file specified by *fd*, and start including the source.

Example:

```
$INCLUDE 9,SUBFILE.FTN,**FINDA-(NEND)
```

This $INCLUDE directive causes the compiler to close lu9 and assign it to SUBFILE.FTN. The compiler then includes source code following the module delimiter **FINDA until the end of file, because NEND is specified. The END statement following **FINDA is treated as the module terminator of **FINDA and not as the end of program.

Format 3:

$INCLUDE *lu* [,*label range*] [(*options*)]

This $INCLUDE directive causes the compiler to start including the source from the specified lu. The lu may have been preassigned by the user or assigned to a file through an earlier Format 2 $INCLUDE directive.

Example:

```
$INCLUDE 9,**FINDA(REW NEND)
```

This $INCLUDE directive causes the compiler to rewind the lu9. The compiler then includes source code following the module delimiter **FINDA until the module terminator (END). The END statement is not treated as end of program, but as a module terminator.

Format 4:

$INCLUDE [,*label range*] [(*options*)]

This form of $INCLUDE can only be used after another $INCLUDE directive. The compiler will use the lu and the file of the previous $INCLUDE at the same level of nesting.

Example:

```
$INCLUDE SUBFILE.FTN,**FINDB
$INCLUDE **FINDA(REW)
```

In this example, the first $INCLUDE will include the module **FINDB from SUBFILE.FTN. The second $INCLUDE will include the module **FINDA from the same file after rewinding the file.

An $INCLUDE directive is said to be a parent of another $INCLUDE if the latter appears in a module being included by the former. The former is said to be on a higher level than the latter. Two $INCLUDE directives are at the same level if both have no parent or both have the same parent. Nesting of $INCLUDE directives can be done up to 7 levels. Recursive $INCLUDE is possible, but definitely not recommended.

The lu specified in Format 3 of the $INCLUDE directive is never closed unless a Format 2 $INCLUDE is encountered for that lu.

A compiler assigned lu for Format 1 and Format 2 $INCLUDE is closed for a particular level when:

- another Format 1 or Format 2 $INCLUDE is encountered at the same level, or
- the parent of a Format 1 $INCLUDE completed inclusion of source, and a higher level $INCLUDE, if any, is made to proceed.

Controlling Compiler List Output

The compiler produces various listings and messages to the list device during compilation. To alter that output, use the following directives:

$ALST/$NALST
$EJECT
$ELIST/$NELIST
$INFORM/$NINFORM
$LCNT *n*
$LIST/$NLIST
$TITLE
$WARN/$NWARN
$WIDTH
$XREF/$NXREF

$ALST/$NALST

$ALST produces a CAL listing of the source program unit in which it appears after the program unit is successfully compiled. The assembly listing is sent to the list device. $NALST turns off this feature. $NALST is the default.

$EJECT

$EJECT causes the compiler to list the subsequent source statements on the next page of the current listing.

Example:

```
C THE SOURCE LISTING OF THIS PROGRAM IS
C PRINTED ON TWO SEPARATE PAGES


      READ (5,10)N
10    FORMAT (I10)
      WRITE (6,15)N
15    FORMAT (1X,'VALUE OF N READ IS', I3)
$EJECT
      CALL SUB1 (M,N)
      WRITE (6,15) N,M
      STOP
      END
```

In this example, page 1 lists all code through statement labeled 15; page 2 lists all lines of code from CALL SUB1(M,N) to END.

$ELIST/$NELIST

$ELIST causes the F7Z compiler to output an extended listing, if subprograms have been expanded in-line during compilation. $NELIST suppresses this listing. See Chapter 10 for a description of the extended listing. If neither $ELIST nor $NELIST is specified and subprograms were expanded in-line, an extended listing is output to the list device. These directives are supported only on F7Z.

$INFORM/$NINFORM

$INFORM produces optimization messages to be output. $NINFORM prevents optimization messages from being sent to the list device.

$LCNT

$LCNT allows the user to change the number of lines of output per page. The format of $LCNT is:

LCNT *n*

The parameter *n* must be greater than or equal to 10. If $LCNT is not specified, the compiler will automatically print 60 lines per page.

Example:

```
C THIS PROGRAM CHANGES THE NUMBER OF
C LINES PER PAGE OF OUTPUT FROM 60 TO 30.

$LCNT 30
      T=1
      CALL SUBA
      CALL SUBB
      STOP
      END
```

$LIST/$NLIST

$LIST causes the compiler to output a source listing until an $NLIST is encountered. $NLIST suppresses the source listing output so that only error messages and source statements that have errors are printed. $LIST is the default.

$TITLE

$TITLE allows the user to specify a title consisting of up to 66 ASCII characters for the source listing. The compiler automatically begins a new listing page whenever $TITLE is encountered. If $TITLE is not specified, the compilers print the first line of the program as a title for each page.

Example:

```
C THIS PROGRAM CHANGES
C THE TITLE OF THE PROGRAM
C TO 'TESTPROGRAM FOR USER SITE #2'
C AFTER ENCOUNTERING THE $TITLE DIRECTIVE
      T=1
3     CALL SUBA
      CALL SUBB
        .
        .
        .
$TITLE 'TESTPROGRAM FOR USER SITE #2
        .
        .
        .
       IF(T.EQ.0)GO TO 3
       STOP
       END
```

A partial listing of this program is as follows:

```
FORTRAN VII-O  Rxx-yy.zz
FORTRAN VIIO:  LICENSED  RESTRICTED RIGHTS AS STATED
  1             C THIS PROGRAM CHANGES
  2             C THE TITLE OF THE PROGRAM
  3             C TO 'TESTPROGRAM FOR USER SITE #2'
  4             C AFTER ENCOUNTERING THE $TITLE DIRECTIVE

  5  000000I          T=1
  6  000028I    3     CALL SUBA
  7  00004CI          CALL SUBB
  .      .              .
  .      .              .
  .      .              .


FORTRAN VII-O Rxx-yy.zz   TESTPROGRAM FOR USER SITE #2

FORTRAN VIIO: LICENSED RESTRICTED RIGHTS AS STATED IN
                $TITLE TESTPROGRAM FOR USER SITE #2
  .     .               .
  .     .               .
  .     .               .
 20  0000F8I          IF(T.EQ.0)GO TO 3
 21  000100I          STOP
 22  000108I          END
```

$WARN/$NWARN

$WARN allows the compiler to output warning messages to the list device. When this directive is used with $NOBJ, $TABLES, and/or $XFORT, the compiler outputs E/SP warning messages in addition to the regular messages. These warning messages flag language constructs in the program that produce a complicated graph or inhibit parallelism. See the appropriate manual in the E/SP documentation set for details on these directives.

NOTE Do not use the $WARN directive with $NOBJ, $TABLES, or $XFORT if you want to suppress these E/SP-related warning messages.

$NWARN prevents all warning messages. If neither $WARN nor $NWARN is specified, warning messages are output.

$WIDTH

$WIDTH allows the user to specify the maximum width of a line in the compiler listings. Lines which normally exceed the width specified are broken into multiple lines. The width of a line can range from 64 to 131 columns. If $WIDTH is not specified, the compiler will automatically output a line of width 131.

Example:

```
C THIS CODE CHANGES THE LINE WIDTH
C OF COMPILER OUTPUT FROM THE DEFAULT
C OF 131 COLUMNS TO 64 COLUMNS.

$WIDTH 64
      I=1
      CALL SUBA
      CALL SUBB
      STOP
      END
```

$XREF/$NXREF

$XREF generates a cross-reference listing of labels and identifiers appearing in the source program. $NXREF turns off this feature. If neither $XREF nor $NXREF is specified, no cross-reference listing is generated.

For more information on the source, extended, CAL, optimization, and cross-reference listings, see Chapter 10.

Inserting CAL Blocks

FORTRAN VII allows users to write FORTRAN programs containing blocks of CAL code. If CAL blocks are embedded in the FORTRAN source code, the compilers automatically produce CAL output instead of object code. The compilers send the CAL output to the device or file assigned to lu6. The CAL output can then be assembled into object code by the CAL assembler.

Directives used to embed CAL code in a FORTRAN source program are:

$ASSM

$FORT

$GOES

$REGS

$SETS

$USES

For more details on the use of these directives, see Chapter 5, "Interfacing Assembly Language Routines."

Controlling Compiler Optimization

All of the global optimizations performed by F7O and F7Z occur automatically during compilation. The optimization directives, however, can be used to suppress the global optimizations. These directives include:

$BASE/$NBASE

$OPTIMIZE/$NOPTIMIZE

$TCOM

NOTE This section presents an overview of the global optimizing directives. Before using these directives, you must have a good grasp of the global optimization techniques explained in Chapter 4.

$BASE/$NBASE

$BASE is used in conjunction with the global register allocation optimization. When specified, $BASE allows the base addresses of all local variables and named common blocks to be considered as candidates for register allocation. $BASE is in effect throughout the entire compilation of the program unit. $NBASE suppresses the effect of the $BASE directive over the entire program unit. $NBASE is the default.

$OPTIMIZE/$NOPTIMIZE

$NOPTIMIZE turns off the following global optimization features:

- Global register allocation
- Extended strength reduction
- Constant propagation
- Invariant code motion
- Test replacement
- Scalar propagation
- Folding and variable propagation
- Common subexpression elimination
- Dead code elimination

$NOPTIMIZE, which can be placed anywhere within the program unit, is in effect during the entire compilation of the unit. When this directive is used with any of the $NOBJ, $TABLES, and $XFORT directives, flow and data analyses, which are normally suppressed by $NOPTIMIZE, are still performed. These operations generate data needed by E/SP to construct a dependence graph of the program. See the appropriate manual in the E/SP documentation set for details on these directives.

$OPTIMIZE activates the F7O and F7Z global optimizations.

$TCOM

$TCOM declares named common blocks, common entities, or global entities as part of a task shared by two or more tasks. $TCOM prevents the compiler from allocating registers for these entities or eliminating code that references them. The format for the directive is a follows:

$$\mathrm{\$TCOM}\ \left[\,[/]\ \mathit{name}_1\ [/]\ \left[,[/]\ \mathit{name}_2\ [/],\ldots\ [/]\ \mathit{name}_n\ [/]\right]\right]$$

Where:

name$_1$, *name*$_2$ ... *name*$_n$ specify variables, common block names (i.e., /ABC/) or blank commons (i.e., //).

Example:

```
C THIS EXAMPLE USES $TCOM TO DECLARE
C WHICH COMMON BLOCK VARIABLES ARE PART
C OF A TASK COMMON AND SHOULD
C NOT BE CONSIDERED AS CANDIDATES FOR
C OPTIMIZATION

$TCOM/ABC/

       COMMON/ABC/EVENT
       LOGICAL EVENT
    10 EVENT = .FALSE.
       IF (.NOT.EVENT) GO TO 10
       STOP
       END
```

In this example, if $TCOM was not specified, the statement labeled 10 and the IF statement would be deleted by the compiler.

$TCOM can appear anywhere in the source code.

Controlling the F7Z In-line Expansion Feature

$INLINE, $INLIB, $INSKIP, and $DISTINCT are four directives designed to invoke the in-line expansion feature available on the F7Z compiler. In-line expansion of subprograms enhances optimization by allowing the compiler to optimize code across program unit boundaries.

NOTE This section presents an overview of the in-line directive syntax. Before using these directives, you must have a good grasp of the in-line expansion feature explained in Chapter 15.

Before the F7Z compiler can expand a subprogram (i.e., subroutine or function) in-line, it must be provided with the following information:

- which subprograms are to be expanded,
- where to find the subprogram, and
- which calls to expand.

NOTE In-line expansion is not the default. The user must explicitly request in-line expansion by specifying the $INLINE directive. In-line expansion and global optimizations are two different features of the FORTRAN VII Z compiler and are both individually controlled.

$INLINE/$NINLINE

The $INLINE directive supplies this information to the compiler as follows:

$$\text{\$INLINE } \textit{name} \left[\textit{:entryname}\right] \left[, \left[\begin{Bmatrix} \textit{fd} \\ * \\ - \end{Bmatrix}\right] \left[, \begin{Bmatrix}\text{ALL} \\ \textit{label}_1[,\ \textit{label}_2,\ldots,\textit{label}_n]\end{Bmatrix}\right]\right]$$

Where:

name is the name of the subprogram that is to be expanded in-line.

entryname is the name of an entry point in the subprogram specified by name. When *entryname* is specified, only those calls made to the entry point of the subprogram are expanded.

fd is the file descriptor of the file that contains the source of the subprogram specified by *name*.

* indicates that the source of the subprogram specified by name is on the same file as the calling program.

- indicates that the subprogram to be expanded is on a file specified by an $INLIB directive or any other $INLINE directive.

If neither *fd*, *, nor - is specified, the compiler will search *name*.FTN for the source of the subprogram specified by *name*.

ALL indicates that all the calls to the specified subprogram are to be expanded in-line.

$label_1$ $[,label_2,...,$ $label_n]$ indicates the statement label(s) containing calls to the specified subprogram. When this argument is specified, only those calls to the specified subprogram in the labeled statement(s) are expanded in-line.

If neither ALL nor a statement label is specified, the compiler will expand all calls to the subprogram.

The first argument of the $INLINE directive is the name of the function or subroutine that is to be expanded in-line. Each subroutine that is to be expanded must be specified by a separate $INLINE directive. If the calling program calls the subprogram at an ENTRY statement rather than the SUBROUTINE or FUNCTION statement, the entryname must be placed after the subprogram name, as shown in the following example.

Example:

```
C THIS PROGRAM REQUESTS INLINE EXPANSION
C OF SUBROUTINE A AND ENTRY B

C
C THE CALLING PROGRAM FOLLOWS
C
$INLINE A,*
$INLINE A:B,*
      INTEGER A1,B1
      COMMON A1,B1
      CALL A
      WRITE (*,5) A1,B1
      CALL B
      WRITE (*,5) A1,B1
5     FORMAT (1X,2I4)
      STOP
      END
C THE CODE FOR SUBROUTINE A FOLLOWS
C
      SUBROUTINE A
      COMMON A1,B1
         .
         .
         .
      ENTRY B
         .
         .
      RETURN
      END
```

The second argument to the $INLINE directive must tell the compiler where to find the subprogram. If the subprogram is on the same file as the calling program, the second argument must be an asterisk (*), as follows:

```
$INLINE A,*
```

If the subprogram is not on the same file as the calling program, $INLINE must specify the file or device in one of the following ways.

Examples:

```
$INLINE SUB2,M300:USER.LIB

$INLINE SUB3

$INLINE SUB2:B,-
```

The second argument in the preceding example indicates the file the compiler must search to find the subprogram. If the second argument is omitted, as in the second example, the compiler will search for a file having the same filename as the subprogram followed by the extension .FTN. In this case, the compiler will search SUB3.FTN for the source code of SUB3. In the third example, the compiler will search for ENTRY B in SUB2 on M300:USER.LIB as specified by the previous $INLINE directive for SUB2. If a hyphen (-) follows the comma, the compiler will search the file indicated by the previous in-line directive within the program unit.

The third argument to the $INLINE directive designates which calls to the subprogram are to be expanded. If ALL is specified, the subprogram will be expanded for all calls within the source program unit in which in-line expansion is requested. An $INLINE directive specifying ALL can be placed anywhere before the END statement.

If a subprogram is called more than once by a source program, the user can request that the subprogram be expanded only for selected calls. To do this, specify the label of the statements that contain those calls as the third argument to the $INLINE directive.

Example:

```
C THIS EXAMPLE SHOWS HOW TO EXPAND
C A SUBPROGRAM FOR SELECTED CALLS WITHIN
C THE SOURCE PROGRAM
C
C
      INTEGER A1,B1,X
      COMMON A1,B1
$INLINE A,*,10
10    CALL A (A1,B1)
      WRITE (*,5) A1,B1
      CALL A (A1,B1)
      WRITE (*,5) A1,B1
5     FORMAT (1X,2I4)
      STOP
      END
       .
       .
       .
```

In this example, subroutine A is only expanded for the call to A, in the statement labeled 10. Subroutine A is not expanded for the second call to A.

If neither ALL nor a statement label is specified, the compiler automatically expands all calls to the subprogram.

$NINLINE

It may be necessary to turn off the in-line expansion feature. This is accomplished through the $NINLINE directive. The format of the $NINLINE directive is:

$NINLINE *name* ,*label*$_1$ [,*label*$_2$,...,*label*$_n$]

Where:

name is the name of the subprogram that is to be excluded from in-line expansion.

label$_1$ [,*label*$_2$,..., *label*$_n$] indicates the statement label(s) containing calls to the specified subprogram. The calls in those statements are excluded from in-line expansion.

Specifying $NINLINE with the subprogram name without any labels turns off the in-line expansion for all calls to the subprogram name.

Specifying $NINLINE with no arguments turns off the in-line expansion feature over the entire calling program unit.

Example:

```
C THIS EXAMPLE USES THE $NINLINE DIRECTIVE
C TO TURN OFF INLINE EXPANSION
C FOR A SELECTED CALL WITHIN THE PROGRAM

$INLINE A,*
      READ (*,2) I
      CALL A
      READ (*,2) I
      CALL A
      READ (*,2) I
$NINLINE A,3
      CALL A
2     FORMAT (I2)
3     CALL A
      END
```

This example shows that to turn off in-line expansion for a selected call, the name of the called subprogram and the label of the statement containing a call to that subprogram must be specified as arguments to the $NINLINE directive. To turn off in-line expansion of all calls to A within a program unit, use $NINLINE A. To turn off in-line expansion for all calls to all subprograms within a program unit, use $NINLINE with no arguments. An $NINLINE directive can be placed anywhere in the calling unit.

$INSKIP

When a program consisting of more than one program unit is compiled in batch, each subprogram is compiled separately. If in-line expansion is requested for a subprogram, the subprogram is incorporated within the calling program. If all calls to a subprogram are to be expanded in-line, it is not necessary to have the subprogram recompiled after the main program unit. To prevent separate compilation of subprograms used only for in-line expansion, use the $INSKIP directive. $INSKIP is placed before the subprogram to inhibit its separate compilation.

The format of the $INSKIP directive is as follows:

$INSKIP [ALL]

If 'ALL' is omitted, the compilation of the program following the $INSKIP directive is skipped. If ALL is used, the compilation of all subsequent subprograms (except BLOCK DATA) is skipped.

Example:

```
C THE FOLLOWING PROGRAM USES $INSKIP
C TO PREVENT SEPARATE COMPILATION OF
C SUBPROGRAMS THAT ARE EXPANDED INLINE.

      PROGRAM MAIN
$INLINE AA,*,ALL
$INLINE BB,*,ALL
      READ (*,2) N
      CALL AA (N)
      READ (*,2) M
      CALL BB (M)
2     FORMAT (I2)
      END

$INSKIP
      SUBROUTINE AA(N)
      IF (N.LE.0) I = 4
      WRITE (*,4) N,I
4     FORMAT (2I2)
      RETURN
      END

      SUBROUTINE BB(M)
      IF (M.GT.0) I = 5
      WRITE (*,5) M,I
5     FORMAT (2I2)
      RETURN
      END
```

This code is compiled as follows:

```
      PROGRAM MAIN
      READ (*,2) N
2     FORMAT (I2)
      IF (N.LE.0) I = 4
      WRITE (*,4) N,I
4     FORMAT (2I2)
      READ (*,2) M
      IF (M.GT.0) I = 5
      WRITE (*,5) M,I
5     FORMAT (2I2)
      STOP
      END

      SUBROUTINE BB(M)
      IF (M.GT.0) I = 5
      WRITE (*,5) M,I
5     FORMAT (2I2)
      RETURN
      END
```

The preceding listing does not show the actual compiler listing, but shows the order with which the example code is compiled using the $INLINE and $INSKIP directives. The optimizing compiler generates unique labels for all labels in a subprogram that is expanded in-line. This prevents multiple definition of labels that can result from repeated expansion of a subprogram or when the main program uses a label which is identical to that used in an in-line expanded subprogram.

Notice that because $INSKIP was not placed above subroutine BB, BB is needlessly compiled. To skip all of the remaining subprograms in a file, write $INSKIP ALL after the END statement of the main program. In this example, $INSKIP ALL should be placed after the END statement of the main program unit. There should be no blank line or comment line between the END statement of the previous program unit and the $INSKIP directive. Otherwise, the compiler warns the user that the directive is ignored.

NOTE $INSKIP or $INSKIP ALL has no effect on BLOCK DATA subprograms. This type of subprogram can never be called from another program unit and thus, never gets expanded in-line. Therefore, BLOCK DATA subprograms appearing after $INSKIP or $INSKIP ALL are compiled as separate units.

$INLIB

Another method of preventing separate compilation of in-line expanded subprograms is to store all subprograms in a source code file.

To tell the compiler which source code file to search for the subprogram, use the $INLIB directive followed by the fd of the source code file or files.

NOTE The $INLINE directive expands only those files containing source codes and not those containing object code. In addition, all subprograms that are available for in-line expansion must reside on a direct access file.

The format of the $INLIB directive is:

$INLIB fd_1 [,fd_2 ,....,fd_n]

Where:

fd_1,....,fd_n are file descriptors of source code files.

Example:

```
C THIS EXAMPLE USES THE $INLIB DIRECTIVE
C TO TELL THE COMPILER WHICH FILE
C TO SEARCH FOR THE SUBPROGRAMS THAT ARE
C TO BE EXPANDED INLINE.

$INLINE AA,-
$INLINE BB,-
$INLINE DD,NEW.FIL
$INLINE DD:BD,-
$INLINE CC,-
$INLIB LIB.FIL
      CALL AA(A)
      CALL BB(B)
      CALL CC(C)
      CALL DD(D)
      CALL BD(E)
      END
```

Remember that when the hyphen is used as the second argument in the $INLINE directive, the compiler searches a file that was already specified elsewhere for the same routine by another $INLINE directive or else by an in-line library. If the file is not designated by an $INLINE directive, the compiler searches for a source code file specified by the $INLIB directive. In this case, $INLIB specifies the file to be searched for subprograms AA, BB, and CC; and $INLINE DD,NEW.FIL specifies the file to be searched for subprogram DD.

$DISTINCT

In the previous examples, the in-line directives are used to invoke expansion of subprograms consisting entirely of FORTRAN source code. To expand a subprogram containing embedded assembly code, the compiler must be informed which CAL symbols are to be replaced by unique compiler generated symbols in each expansion of that subprogram within the same program unit. This is done through the $DISTINCT directive.

Format:

$DISTINCT CAL*symbol*$_1$ [,CAL*symbol*$_2$,...,CAL*symbol*$_n$]

Where:

CAL*symbol* represents labels within an embedded CAL block which need unique labels in the generated in-line expanded code.

Example:

```
C THIS EXAMPLE USES THE $DISTINCT
C DIRECTIVE TO PREVENT MORE THAN
C ONE LINE OF ASSEMBLY CODE FROM
C HAVING IDENTICAL LABELS.

$INLINE B,*
      PROGRAM MAIN
      READ (*,2)I
      CALL B
      READ (*,2)I
      CALL B
      READ (*,2)I
      CALL B
2     FORMAT (I2)
      END

      SUBROUTINE B
      COMMON /I/
$ASSM
$USES
$GOES  10
$SETS I
$DISTINCT AROUND
         B     AROUND
         B     $P10
AROUND   EQU   *
         ST    B,I
$FORT
10    WRITE (*,12) I
12    FORMAT (I4)
      RETURN
      END
```

This example illustrates the conflict in label referencing that would result if $DISTINCT is not used when a subprogram containing embedded assembly code is expanded more than once within the same program unit. In this case, the label AROUND would be defined three times in the calling program. $DISTINCT tells the compiler to generate a unique symbol for AROUND each time subroutine B is expanded. Therefore, at each instruction generated for each expansion of B AROUND, the compiler will generate a unique label.

The $DISTINCT directive can appear anywhere before the first occurrence of the CAL symbol in the block. Symbols specified by a $DISTINCT directive must not appear in a FORTRAN statement.

$DISTINCT will not replace CAL type specifiers (Z, H, Y, X, etc.) with compiler generated symbols. For example, the directive:

$DISTINCT Z would have no effect on the following instruction:

```
DC Z(B)
```

Debugging the Source Code

Four compiler directives can be used as run-time debugging aids. These include:

```
$COMP/$NCOMP
$TEST/$NTEST
$TRACE/$NTRACE
```

```
$COMP/$NCOMP
```

$COMP allows the programmer to insert debugging statements within the source code without having to delete them individually after the debugging session is over. Debugging statements are identified by placing an X in column 1 of the first line of the statement. $COMP causes the compiler to compile these statements conditionally; they can be deactivated by $NCOMP without having to remove the debugging statements from the source code.

$NCOMP prevents the compiler from compiling statements identified with an X in column 1. When $NCOMP is specified, the source listing will indicate which statements was not compiled by replacing X with #. $NCOMP is the default.

```
$TEST/$NTEST
```

$TEST checks array subscripts and substrings during program execution against their declared bounds. $TEST causes the compiler to generate code that activates the .TEST RTL routine. This routine outputs an error message at any point in the program when the value of an array subscript or substring falls outside the declared bounds.

$TEST can be used to check the value of subscripts of specific arrays and substrings, all program subscripts and substrings, or all subscripts and substrings up to a specified statement in the program.

The $TEST option is intended to check the array subscript and substring bounds of array and character entities occurring in arithmetic, logical, and character expressions. It is not intended for checking boundary violations of arrays passed as arguments to subprograms or used as a buffer in ENCODE/DECODE statements. $NTEST is the default.

For more details on these directives, see Chapter 9.

$TRACE/$NTRACE

$TRACE allows the user to trace the value of a variable as the program executes. It is also used to trace the flow of control through the program. $TRACE causes the compiler to generate code that activates the $.TRACE RTL routine. This routine prints out a message every time a specified variable or array is assigned a value through an assignment statement. This message shows the value assigned to the variable at that point in the program. .TRACE also prints a message every time a labeled statement is executed.

$TRACE can be used to trace the value of specific variables, all labeled statements and variables throughout the program, or all variables and labeled statements up to a specified statement in the program. $NTRACE is the default.

Format:

$TRACE [$var_1$ [,var_2,...,var_n]]

Where:

var represents the variable whose values are traced during execution. If no arguments are specified, labeled statements and the values of all variables are traced throughout the program.

Preparing Your Code for Parallelization

The following directives prepare your program for the parallelization tool (E/SP):

$OBJ/$NOBJ
$OVERLAP/$NOVERLAP
$SAFE/$NSAFE
$TABLES/$NTABLES
$XFORT/$NXFORT

$OBJ/$NOBJ

$OBJ controls the generation of object code by the compiler. If $TABLES is not specified, $NOBJ is ignored. If $NOBJ and $TABLES are simultaneously in effect, the compiler skips register allocation and code generation. This causes the compiler to behave as the front-end to E/SP, saving compile time and offsetting the increase in compile time caused by the writing of the tables to a file. If $OBJ and $TABLES are simultaneously in effect, the compiler generates sequential object code or CAL code. The default is $NOBJ if $TABLES is specified, otherwise, the default is $OBJ.

$OVERLAP/$NOVERLAP

$OVERLAP indicates that the currently compiled module may be called in the program in such a way that it redefines one of its dummy arguments. If $TABLES is not specified, this directive is ignored. $NOVERLAP is the default.

$SAFE/$NSAFE

$SAFE flags whether it is safe to parallelize code with embedded assembly code. $SAFE tells the compiler that any assembly code does not contain SVC calls that could change the MPS system state. The default is $SAFE if the program has no embedded assembly code, otherwise $NSAFE. The default is $NSAFE if $TABLES is specified, otherwise, the default is $SAFE. This directive may be specified anywhere in a module. If both $SAFE and $NSAFE are specified in the same module, the last specified directive takes effect on the module. If a module contains more than one $ASSM block and at least one is considered "not safe," the entire module must be assumed to be $NSAFE.

$TABLES/$NTABLES

$TABLES generates dependence tables and transcribed source code before downloading a program to E/SP for restructuring. Unless specified, $TABLES will not be in effect.

$XFORT/NXFORT

$XFORT informs the compiler that the code may contain non-FORTRAN structures which require parsing. The default is $XFORT not in effect. This directive must appear before the first FORTRAN statement or may be used as a start option.

For more details on these directives and on E/SP, see the appropriate manual in the E/SP documentation set.

Miscellaneous Instream Compiler Directives

The following instream directives can also be used:

$APU/$NAPU
$CAL/$NCAL
$DCMD
$DP
$F66DO/$NF66DO
$HOLL/$NHOLL
$IBYTE
$INT2
$LBYTE
$LTORBIT
$PASSBYADDRESS
$PAUSE
$PROG
$REENTRANT/$NREENTRANT
$RTOLBIT
$SEG/$NSEG
$SYNTAX/$NSYNTAX
$TARGET *n*
$TRANSCENDENTAL/$NTRANSCENDENTAL
$UNNORMALIZE/$NUNNORMALIZE

$APU/$NAPU

$APU/$NAPU are provided for multi-processor applications. $APU causes the F7 compilers to look for FORTRAN features in the input source that are known to generate SVC instructions. Whenever they are detected, the compiler generates an informative message on the listing for those statements. The compiler also outputs a DCMD command in the object for LINK/32 to generate a similar message at LINK time. The format of the APU warning message is:

```
ABOVE STATEMENT GENERATES SVC CALL.
```

For a module, the compiler generates one DCMD whose text is:

```
****MODULE name INVOKES SVC.
```

Where:

name is the name of the module.

This message appears on the LOG file at Link time. Such messages have no effect on the end of task code of the compilers. $NAPU prevents the F7 compilers from generating the above messages for statements which generate SVC instructions. The $APU/$NAPU may appear anywhere in the source and can be used to toggle the option on and off.

```
$CAL/$NCAL
```

$CAL is provided for those users who require assembly language code instead of object code of a FORTRAN source program. $CAL causes the compiler to generate assembly code for a FORTRAN program. The resultant assembly code, which is output to the file or device assigned to lu6, can be assembled by CAL/32. $NCAL suppresses the CAL feature. These directives can be placed anywhere in a program unit except in an embedded CAL block.

```
$DCMD
```

$DCMD allows you to embed Link commands or comments within the object code for processing during linking.

```
$DP
```

$DP causes all real and complex items whose lengths were not specified explicitly in a specification statement to be treated as double precision items. Length specification of *4 and *8 (for REAL) and *8 and *16 (for COMPLEX) are still available if explicitly used in specification statements. Further, all REAL and COMPLEX constants will be treated as REAL*8 and COMPLEX*16 constants, respectively, when $DP is specified. If you do not specify this option, the default is the type associated with FORTRAN identifiers and constants. This directive must appear before the first FORTRAN statement of a module, including the FUNCTION or SUBROUTINE statements. If this option appears after the first statement of a module, you get a warning message and the compiler ignores the directive. The scope of $DP is limited to the module in which it appears.

$F66DO/$NF66DO

$F66DO causes all DO loops to be executed at least once. This directive supports compatibility with FORTRAN 66. Specifying this directive produces code that is not compatible with the ANSI Standard FORTRAN 77. This directive must appear before the first FORTRAN statement of a module, including the FUNCTION or SUBROUTINE statements. If this option appears after the first statement of a module, you get a warning message and the compiler ignores the directive. The scope of $F66DO is limited to the module in which it appears.

$HOLL/$NHOLL

$HOLL causes the compiler to interpret all quoted strings used as arguments to subprograms as Hollerith constants rather than character constants up to the point $NHOLL is specified. These quoted strings can then be passed to any type of dummy arguments except character. $NHOLL turns off the $HOLL feature.

$IBYTE

$IBYTE treats all entities appearing in the "var list" of the BYTE statement as INTEGER*1 entities. This directive must appear before the first FORTRAN statement in the module or as a start option.

$INT2

$INT2 treats all INTEGER variables whose lengths were not specified explicitly in a specification statement as INTEGER*2 variables. Similarly, all LOGICAL variables whose lengths were not specified explicitly in a specification statement are treated as LOGICAL*2 variables. All other length specifications (*1, *2, and *4) for both types are still available if explicitly used in specification statements.

4-byte entities (INTEGER*4 or LOGICAL*4) must be specified where such entities are required (i.e., variable of an ASSIGN statement).

If $INT2 is specified and a constant used is larger than the integer value 32767 or smaller than -32768, an INTEGER*4 constant element is created for that constant and a warning message is issued. Otherwise, an INTEGER*2 constant is created.

The $INT2 option must appear before the first FORTRAN statement of a module, including the FUNCTION or SUBROUTINE statements. If this option appears after the first statement of a module, you get a warning message and the compiler ignores the directive. The scope of $INT2 is limited to the module in which it appears. Not specifying this directive gives you the usual typing associated with FORTRAN identifiers and constants. Note that this feature conflicts with the "storage unit" standards of FORTRAN 77.

$LBYTE

$LBYTE treats all entities appearing in the "var list" of the BYTE statement as LOGICAL*1 entities. This directive must appear before the first FORTRAN statement in the module or may be used as a start option.

$LTORBIT

$LTORBIT causes the bit positions in a word to be counted from left to right. In a 4-byte word, the leftmost (most significant) bit position is marked as 0 and the rightmost (least significant) bit position is marked as 31. If neither $LTORBIT nor $RTOLBIT is specified, the bit positions are counted from left to right. This directive affects all bit manipulation routines. $LTORBIT does not conform with Military Standard 1753 extensions and IRTF standard, but is compatible with FORTRAN VII R05-05 and earlier. This directive must appear either as an instream directive before the first FORTRAN statement in a module, or as a start option.

$PASSBYADDRESS

$PASSBYADDRESS causes the module to treat all of its noncharacter scalar dummy arguments as if they were passed by reference. If $NPASSBYADDRESS is in effect, noncharacter scalar dummy arguments which are not enclosed in slashes in the FUNCTION or SUBROUTINE statement are treated as if they were passed by value-result.

This option has no effect on arguments passed to other subprograms; the choice of passing by reference vs. passing by value-result is determined solely by the coding of the FUNCTION/SUBROUTINE statement that receives the arguments. This directive must appear before the first FORTRAN statement of a module, including the FUNCTION or SUBROUTINE statements. If this option appears after the first statement of a module, you get a warning message and the compiler ignores the directive. The scope of $PASSBYADDRESS is limited to the module in which it appears.

$PAUSE

$PAUSE suspends compilation and causes the compiler to pause at the point where this directive is encountered. $PAUSE can be used to suspend compilation anywhere in the program. To resume compilation, enter the OS/32 CONTINUE command.

$PROG

$PROG can be used to change the name of a program unit. In the absence of $PROG, the compiler uses .MAIN as the name for the main program unit; all subroutines and functions are referred to by their subroutine or function name. $PROG overrides the name specified in the PROGRAM statement.

Example:

```
C THIS EXAMPLE ILLUSTRATES
C THE USE OF $PROG
$PROG PROG1
      READ (5,10)N
      WRITE (6,*)N
10    FORMAT (I3)
      STOP
      END
```

$PROG can appear anywhere in the program unit. The argument to $PROG is a symbol of 1- to 6-characters. The symbol chosen can be a legal FORTRAN symbol or a legal CAL symbol.

$REENTRANT/$NREENTRANT

$REENTRANT produces reentrant code which allows you to develop sharable (reentrant) libraries. $REENTRANT supports E/SP. Reentrant code also allows parallel execution of DO loops containing subroutine calls without replication of subroutine code. Use of $REENTRANT precludes the use of any construct resulting in the initialization or modification of static storage, i.e., DATA, GLOBAL, COMMON, or SAVE statements. It may also result in substantially less efficient code. The default is $NREENTRANT.

It may be necessary to increase the size of the run-time library (RTL) stack. The default size of the RTL stack is X'2000'. The size can be changed if more workspace is necessary by patching the RTL object file using the OS/32 PATCH utility. To modify the stack size, apply the following patch:

```
OBJ    F7LIB60.LIB,NEWRTL.LIB,LIB
GET    __U, COPY
BIAS   0:P
VER    14,0,2000
MOD    14,0,3000
SAVE   COPY,TERM
END
```

In the above example, the size of the stack was increased from 8192 to 12228 bytes. For more information on patches, see the *OS/32 PATCH Utility Reference Manual.*

NOTE The amount of workspace required by your task will increase if the size of the RTL stack is increased.

```
$RTOLBIT
```

$RTOLBIT causes bit positions in a word to be counted from right to left. In a 4-byte word the rightmost (least significant) bit position is marked as 0 and the leftmost (most significant) bit position is marked as 31. If neither $RTOLBIT nor $LTORBIT is specified, the bit positions are counted from left to right.

$RTOLBIT affects all the bit manipulation routines. Using this directive forces the interpretation of bit manipulation functions to be compatible with Military Standard 1753 extensions and IRTF standard, but incompatible with the earlier versions of FORTRAN VII. This option must appear before the first FORTRAN statement in a module.

If one module uses this option, $RTOLBIT will apply to all modules in the application for any bit manipulation routine whose code is not generated in-line. Thus, if $RTOLBIT was specified in any module:

- Bit manipulation routines passed as arguments will use the $RTOLBIT conventions.
- Bit manipulation routines declared externally will also follow the $RTOLBIT conventions.

In these cases, the program is linking to the RTL. It is suggested that code be developed with only one direction specified.

If your program is compiled with the $RTOLBIT directive, a DCMD LINK command is embedded in the generated object code. The text of the DCMD is:

```
****MODULE xxxxCOMPILED WITH $RTOLBIT
```

Where:

xxxx is the name of the program.

$SEG/$NSEG

$SEG causes the F7O and F7Z compilers to generate segmented object code. When object code is segmented, all local data is placed in the impure segment while all executable code is placed in the pure segment. See the *OS/32 Application Level Programmer Reference Manual* for more information on segmented tasks. Segmented code runs faster than nonsegmented code on the Model 8/32.

$NSEG prevents the F7O and F7Z compilers from generating segmented object code. All code is placed in the impure segment. When a program is compiled using $NSEG, the size of the resultant object code is reduced. Nonsegmented code runs faster than segmented code on Series 3200 Processors.

NOTE Object code located in an impure task segment cannot be shared.

$SYNTAX/$NSYNTAX

$SYNTAX causes the F7O and F7Z compiler to perform a syntax check of the source code without generating object code. $NSYNTAX turns off the $SYNTAX feature. $SYNTAX should be used to decrease compilation time when developing programs with the F7O and F7Z compilers.

$TARGET *n*

$TARGET causes the F7O and F7Z compilers to generate machine code specifically optimized to the instruction set available on the Concurrent 32-bit processor whose model number (8/32, 3205, 3210, 3230, 3240, 3200CPU, 3200APU, etc.) is defined by *n*. If *n*=0, the compiler outputs machine code capable of being executed on any one of the 32-bit processors. If *n*=3200, then the machine code is targeted for any of the Series 3200 Processors. If the directive is not specified, the compiler will output machine code targeted to the processor on which the compiler is running. To target code to the MicroThree, MicroFive, or 3280E, use $TARGET 3283, $TARGET 3285, or $TARGET 3288, respectively.

If $TARGET 3203, $TARGET 3205, or $TARGET 3280 is specified, the compiler, by default, generates unnormalized floating point load instructions. However, when $TARGET 3203 is specified, the code generated for the Model 3203 does not execute on any processor other than the Model 3203, 3205, or 3280 Systems. Similarly, when $TARGET 3205 is specified, the code generated for the Model 3205 does not execute on any processor other than the Model 3203, 3205, or 3280 Systems. If $TARGET 3280 is specified, the code generated executes on the 3280, 3283, 3285, and 3288 Systems. A change to $TARGET will force NUNNORMALIZE if the processor does not accept it. If the compiler generates any unnormalized floating point instructions, a DCMD LINK command is embedded in the generated object code. The text of the DCMD is:

```
****MODULE xxxx CONTAINS NON-NORMALIZING LOADS
```

Where:

xxxx is the name of the program where unnormalized floating point load instructions were generated.

NOTE If a program unit is compiled with $TARGET 3200 or higher, the resultant object code will not execute on a Model 8/32 processor. If a program unit is compiled with $TARGET 328*x*, the resultant object code will not execute on any other Series 3200 Processor.

$TRANSCENDENTAL/$NTRANSCENDENTAL

$TRANSCENDENTAL causes transcendental functions to be executed via single microinstructions in the machine code for the 328*x* processors (3280, 3283, 3285, 3288). $NTRANSCENDENTAL causes the compiler to generate calls to the RTL version of each transcendental function. The default is $TRANSCENDENTAL for a 328*x* processor and $NTRANSCENDENTAL for any other Series 3200 Processor. Specifying $NTRANSCENDENTAL on a 328*x* processor generates calls to the RTL version of each transcendental function. This produces consistent results across Series 3200 processors.

If your program is compiled with the $NTRANSCENDENTAL directive, a DCMD LINK command is embedded in the generated object code. The text of the DCMD is:

```
****MODULE xxxxCOMPILED WITH $NTRANSCENDENTAL
```

Where:

xxxx is the name of the program.

$TRANSCENDENTAL cannot override the TARGET start option or the $TARGET directive. Therefore, you must specify $TARGET if the processor on which the compilation is being done doesn't support nontranscendental functions.

Example 1:

In the following example, the $TARGET 3200 option is specified, and $NTRANSCENDENTAL is the default. The $TRANSCENDENTAL directive is overridden by the $TARGET 3200 option, and is, therefore, ignored.

```
$TARGET 3200
$TRANSCENDENTAL
```

Result: Transcendental functions are inhibited.

Example 2:

In the following example, the $TARGET 3280 option is specified, and $TRANSCENDENTAL is the default. In this case, the $NTRANSCENDENTAL directive overrides the $TARGET 3280 default.

```
$TARGET 3280
$NTRANSCENDENTAL
```

Result: Transcendental functions are inhibited.

Respecifying the $TARGET option for a model whose default is $TRANSCENDENTAL (i.e., 328*x* processor) will override the $NTRANSCENDENTAL directive.

$UNNORMALIZE/$NUNNORMALIZE

$UNNORMALIZE directs the compiler to generate unnormalized floating point load instructions. If the code is being targeted for the Model 3203, 3205, or 3280 processors, $UNNORMALIZE is the default. Otherwise, $NUNNORMALIZE is the default. If the compiler generates any unnormalized floating point load instructions, a DCMD LINK command is embedded in the generated object code. The text of the DCMD is:

```
****MODULE xxxx CONTAINS NON-NORMALIZING LOADS
```

Where:

xxxx is the name of the program where unnormalized floating point load instructions were generated. This comment on the DCMD is displayed on the LOG device by OS/32 LINK when the object code is used to build a task. See the *OS/32 LINK Reference Manual* for a discussion of the DCMD command.

$NUNNORMALIZE inhibits the generation of unnormalized floating point load instructions regardless of the $TARGET option. For processors other than the Model 3203, 3205, and 3280, $NUNNORMALIZE is the default.

$UNNORMALIZE cannot override the TARGET start option or the $TARGET directive. Therefore, you must specify $TARGET if the processor on which the compilation is being done doesn't support normalized loads.

Example 1:

In the following example, the $TARGET 3200 option is specified, and $NUNNORMALIZE is the default. The $UNNORMALIZE directive is overridden by the $TARGET 3200 option, and is, therefore, ignored.

```
$TARGET 3200
$UNNORMALIZE
```

Result: Unnormalized floating point instructions are inhibited.

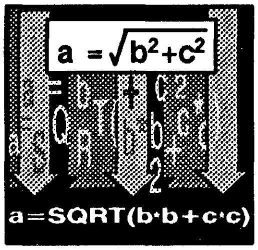
In the following example, the $TARGET 3280 option is specified, and $UNNORMALIZE is the default. In this case, the $NUNNORMALIZE directive overrides the $TARGET 3280 default.

Example 2:

```
$TARGET 3280
$NUNNORMALIZE
```

Result: Normalized floating point instructions are generated.

Respecifying the $TARGET option for a model whose default is $UNNORMALIZE (i.e., Model 3203, 3205, and 3280 processors) will override the $NUNNORMALIZE directive.



4

Preparing Your Source Code

In this chapter

We introduce you to certain programming practices that should be followed when preparing source code for use with the Concurrent FORTRAN VII compilers. This chapter shows how you can take advantage of the features of the FORTRAN VII language to produce valid and efficient code. It provides you ways to avoid the common pitfalls experienced by FORTRAN programmers. In addition, it discusses user-level optimization of input/output (I/O).

Topics include:

- Calling subroutines with dummy arguments
- Processing of DO loops
- Using the computed GOTO and assigned GOTO
- Using parentheses
- Converting data types
- Defining program entities
- Testing values of floating point variables
- Equivalencing integer variables to floating point variables
- Improving program readability
- Optimizing I/O operations
- Preparing code for optimization

Calling Subroutines with Dummy Arguments

The following precautions must be taken into consideration when using dummy arguments in subroutine calls.

- When calling a subroutine, avoid writing code that passes the same actual argument to two different dummy arguments, as shown in the following example

Example:

```
           CALL TRYONE (C,C)
                 .
                 .
                 .
           END
           SUBROUTINE TRYONE (A,B)
                 .
                 .
                 .
           A=. . .
      OR
           B=. . .
                 .
                 .
                 .
           END
```

In this example, dummy arguments A and B each become associated with the same actual argument C and therefore, with each other. It is not possible to say what the value of C is after returning from TRYONE. Neither A nor B should be modified by this CALL statement nor by any procedures called by TRYONE or its descendants.

- Avoid using different dummy arguments at multiple entry points to a program, as shown in the following example.

Example:

```
      SUBROUTINE MAIN (A,B,C)
                 .
                 .
                 .
      GO TO 100
                 .
                 .
                 .
      ENTRY MESS (A,B,D)
                 .
                 .
                 .
100   A=B*D
                 .
                 .
                 .
      END
```

The statement at label 100 uses dummy argument D, which is not defined if the subroutine is entered through MAIN, as shown in the previous example. Therefore, if the subroutine is entered through MAIN and D was not previously set by an earlier call to MESS, A is undefined after execution of statement 100. A similar situation exists for C if the subroutine is entered through MESS.

- Do not use a scalar argument to pass an array to a subroutine.

Example:

```
C THE FOLLOWING CODE IS ILLEGAL IN FORTRAN
      INTEGER  I(10)
      CALL SUBA (I)
         .
         .
         .
      END


      SUBROUTINE SUBA (J)
C NOTE THAT J IS NOT DIMENSIONED
      CALL SUBB (J)
      END

      SUBROUTINE SUBB (K)
      DIMENSION K(10)
         .
         .
         .
      END
```

In this example, array I is passed to SUBA using scalar argument J (note that J is not dimensioned and therefore, is a pass-by-value argument). See *FORTRAN VII Language and Syntax — A Reference* for an explanation of argument passing conventions. SUBA then passes the scalar variable J to the dummy array K in SUBB. Because FORTRAN VII adopts different argument passing conventions for array and scalar arguments, array K does not refer to array I.

- In general, when passing a character constant or expression to a subprogram, the corresponding receiving dummy argument must also be defined as type character. When passing a Hollerith constant as an actual argument, the corresponding dummy argument can be any data type except character

Example:

```
CHARACTER*8 PASS
DATA PASS/'HELLO'/
CALL A(PASS)
END
SUBROUTINE A(RECEIVE)
CHARACTER*8 RECEIVE
TYPE *,RECEIVE
RETURN
END
```

In the previous example, RECEIVE must be declared as a character type in order for SUBROUTINE A to execute properly. Otherwise, a value other than HELLO is printed by SUBROUTINE A.

Specifying the $HOLL directive in the calling program unit allows a character constant to be passed to a noncharacter dummy argument. When this directive is used, a character expression passed as an actual argument requires the corresponding dummy argument to be defined as type character; passing a Hollerith constant as an actual argument requires the corresponding dummy argument to be any data type except character. See Chapter 3 for more information on the $HOLL directive.

Processing of DO Loops

FORTRAN VII follows ANSI FORTRAN 77 specifications with respect to DO loops. These specifications differ from earlier versions of FORTRAN and must be taken into consideration when writing the source program.

The DO statement is used to repeat the execution of a group of statements. The range of a DO loop is the set of executable statements following the DO statement, up to and including the terminal statement of the loop.

Processing within a DO loop involves initialization of an index variable, increment of the index, test of the index, test of the variable, and exit of the loop.

The format of the DO statement is:

DO *n I=A,B,C*

Where:

n	indicates the label of the last statement of the loop.
I	is the index variable.
A	is the initial value of the DO variable.
B	is the final value of the DO variable.
C	is the increment value.

A DO index can be a variable or expression of type INTEGER*4, INTEGER*2, REAL*4 or REAL*8. *A* is the value initially assigned to *I*. *B* is the final value associated with the loop, and *C* is the increment value. The type of *I* should be consistent with the types of the index values, i.e., if *A*, *B*, and *C* are of type REAL*4, then *I* should be declared as REAL*4. Otherwise, the loop might perform differently from its intended usage. For more information on the iteration count of a DO loop, see *FORTRAN VII Language and Syntax — A Reference Manual*.

If the value of *C* is positive, iterations of the loop are repeated until the value of *I* is greater than *B*. When *I* > *B*, the DO loop becomes inactive and execution continues with the first executable statement following the last statement in the DO loop.

If *C* is a negative integer, the DO loop becomes inactive when the value of *I* is less than *B*. The final value of *I*, then, is the value that deactivated the loop, not the value of *I* before the last iteration.

Example:

```
      A=5
      B=3
      DO 10 I=-10,0,+1
      DO 20 J=A,B
      .
      .
      .
20    CONTINUE
C   BECAUSE A IS LARGER THAN B, THE J LOOP
C   WILL NOT BE EXECUTED. THIS IS NOT
C   FLAGGED DURING COMPILATION OR RUN-TIME.
C   THE FINAL VALUE OF J IS 5.
      .
      .
      .
10    CONTINUE
C    THE I LOOP WILL BE EXECUTED 11 TIMES
C    THE FINAL VALUE OF I IS 1
```

At the point of deactivation of a DO loop, the value of the DO variable may not be the expected value if you are not careful.

Example:

```
C THE FOLLOWING CODE RESULTS IN AN UNDEFINED VALUE
C FOR I AT THE EXIT OF THE LOOP
      INTEGER*2 I
      DO 10 I=0,32000,1000
      .
      .
      .
10    CONTINUE
```

In this example, the value of I is incremented by 1000 after each iteration. After the thirty-third time through the loop, I should have the value of 33000. However, since the maximum value for any INTEGER*2 variable is 32767, this example results in an overflow condition which is not flagged during run-time. I is undefined upon deactivation of the loop. Future uses of the DO variable I would not provide the expected value. You get a warning message on this condition during compilation.

DO statement indexes must not be changed during the execution of the loop. The following example includes an illegal statement that attempts to modify the DO loop index inside its respective loop.

Example:

```
C THE FOLLOWING CODE IS ILLEGAL
      EQUIVALENCE (I,J)
      DO  10 I=1,10
      .
      .
      .
9     IF (COND) J=11
      CALL ABC(I); ABC MUST NOT MODIFY I
10    CONTINUE
```

In this example, statement 9 is illegal since I and J are equivalent and J attempts to modify the value of I if the (COND) statement is true. Under no condition can the index of the loop be changed, but in this case, no warning message is issued by the compiler.

Using the Computed GOTO

If the value of the index variable appearing in a computed GOTO statement is less than 1 or greater than the number of statement labels appearing in the statement label list, FORTRAN VII transfers control to the statement following the computed GOTO statement. If the value of I is a constant, the F7O/F7Z optimizer messages alert you that this transfer of control has occurred because the value of I is out of range.

Example:

```
    GOTO (90,80,70,60)I
10  CONTINUE
```

In this example, control is transferred to statement 90 if the value of I is 1, to statement 80 if the value of I is 2, to statement 70 if the value of I is 3, and to statement 60 if the value of I is 4. Otherwise, control falls through to the statement labeled 10.

Using the Assigned GOTO

When using an assigned GOTO statement followed by a parenthesized list of statement numbers, care must be taken so that the statement number assigned to the integer variable is a member of the list.

Example:

```
ASSIGN 20 TO L
GOTO L, (10,20,30)
```

In this example, 20 must be included in the parenthesized list of the assigned GOTO, otherwise, the transfer of control is undefined.

Using Array Subscripts

Make certain that all subscripts are within the range of their declared bounds. Referenced array elements whose subscripts are not within their declared bounds can yield unexpected results.

Example:

```
   INTEGER A(9)
   DO 10 I=1,10
      A(I)=0
       .
       .
       .
10 CONTINUE
```

In the previous example, the value of A(I) is out of bounds in the last iteration of the loop.

Using Parentheses

Parentheses are used to override the precedence of operators and to establish the order of evaluating an expression.

Example:

```
C THE FIRST ASSIGNMENT STATEMENT
C MULTIPLIES C BY THE SUM OF A+B
C
C THE SECOND ASSIGNMENT STATEMENT
C ADDS THE PRODUCT OF B*C TO A
C
      Y = (A+B)*C
           .
           .
           .
      Y = A+B*C
```

Parentheses must be used to express a complex constant.

Example:

```
C THE FOLLOWING EXAMPLE USES
C PARENTHESES TO DESIGNATE A
C COMPLEX CONSTANT
C
      COMPLEX COMP
      COMP = (1.0,2.0)
      WRITE (6,100) (1.0,2.0), I+J
      .
      .
      .
```

Redundant parentheses used to enclose an entire I/O list can produce an error message if the parenthesized list is illegal. A parenthesized I/O list is illegal if the entities within the parentheses do not evaluate to a single value.

Example:

```
C THE FOLLOWING WRITE STATEMENT WILL RETURN
C A COMPILATION ERROR FOR INVALID COMPLEX
C ENTITY
C
      WRITE (6,100) (X,Y)
      .
      .
      .
100   FORMAT (2F10.0)
      .
      .
      .
```

In this example, (X,Y) is interpreted as an illegal parenthesized list since the variable X and Y do not evaluate to a single value. The following example illustrates legal uses of parentheses in I/O lists.

Example:

```
      .
      .
      .
      WRITE (6,100) (X+Y), ((4))
      WRITE (6,200) (1,2)
      .
      .
      .
```

The above statement contains legal redundant parentheses. The expression X+Y can be enclosed in parentheses since it evaluates to a single value.

Converting Data Types

Certain precautions must be taken when performing data type conversions within a program. Otherwise, the conversion may produce values different from what is expected. The necessary precautions are discussed in the following sections.

Allocation of Variables in Common

Storage allocated to variables in common is always aligned on proper byte boundaries depending on their data type. CHARACTER, INTEGER*1, BYTE, and LOGICAL*1 variables are aligned on one byte boundaries. INTEGER*2 and LOGICAL*2 variables are aligned on halfword boundaries. Variables of any other type are aligned on fullword boundaries. Therefore, when placing variables in a common block, you must be aware of the alignment constraints imposed on these variables.

Equivalencing may affect the mapping of varibles in common. See the "EQUIVALENCE Statement" in Chapter 3 for more information.

Example:

```
COMMON M,N
CALL SUBA
STOP
END
SUBROUTINE SUBA
INTEGER*2 I
COMMON I,J
STOP
END
```

In this example, the variables M and N, which are both INTEGER*4 variables, are allocated to two adjacent fullword locations. Whereas the variables I and J are allocated with a 2-byte gap between them because J must be aligned on a fullword boundary.

Figure 4-1 shows that INTEGER*2 variable I occupies two bytes of LOC 1. The variable I, therefore, does not see the same value as the variable M in the main program.

	Common	Area
	Fullword LOC 1	Fullword LOC 2
Common M,N	M	N
Common I,J	I |	J
Byte Offset	0 | 1 | 2 | 3	4 | 5 | 6 | 7

Figure 4-1. Sample Common Data Areas

Integer Conversion

When converting a larger size integer to a smaller size integer (i.e., an INTEGER*4 value to an INTEGER*2 value, INTEGER*2 value to an INTEGER*1, INTEGER*4 value to an INTEGER*1 value) the larger size integer value must be within the acceptable range of the smaller size integer (-32,768 through 32,767 for INTEGER*2 and -128 through 127 for INTEGER*1). INTEGER*2 and INTEGER*1 variables can only have values in these ranges because of their limited number of bytes. Assigning a value to an integer variable outside its allowable range results in an undefined value.

In Examples 1 and 2 below, the value of I2 is undefined.

Example 1:

```
C THE FOLLOWING CODE UNSUCCESSFULLY ATTEMPTS TO CONVERT
C THE VALUE OF I4 TO AN INTEGER*2
C VALUE; THE RESULTING VALUE IS OUTSIDE
C THE RANGE OF THE DATA TYPE

      INTEGER*2 I2
      I4 = 1
      I2 = ISHFT(I4,16)
```

Example 2:

```
C THE FOLLOWING CODE UNSUCCESSFULLY ATTEMPTS TO CONVERT
C THE VARIABLE WHOSE VALUE IS Y'0000FFFF'
C TO AN INTEGER*2 DATA TYPE.
C I2 CANNOT ACCEPT A VALUE GREATER
C THAN THE RANGE OF ITS DATA TYPE

      INTEGER I4/Y'0000FFFF'/,I2*2
      I2 = I4
```

Example 3:

```
C THIS EXAMPLE CONVERTS THE LOWER
C HALFWORD OF AN INTEGER*4 VALUE
C TO AN INTEGER*2 VALUE
      IF ( BTEST (I4,16)) THEN
          I2 = IOR (I4,Y'FFFF8000')
      ELSE
          I2 = IAND (I4,Y'00007FFF')
      ENDIF
```

Defining Program Entities

To avoid unpredictable results during program execution, you must ensure that all entities (variables and arrays) referenced within a program are initialized to the desired values before being used. Otherwise, these entities acquire values depending on where they are mapped in memory. This can produce erroneous results or run-time errors.

An entity is defined in a FORTRAN VII program after the program executes one of the following statements.

- Assignment statements - These statements evaluate an arithmetic, logical, or character expression and assign its value to the variable, array element or character substring to the left of the equal sign (=).
- Input statements (READ, ACCEPT, ENCODE) - These statements assign a value to an entity from the input medium, provided that the data type of the entity matches that of the data.
- DO - This statement defines the DO variable.
- Implied DO list in I/O statement - This statement defines the implied DO variable.

- DATA - This statement initializes variables, arrays, array elements, and substrings during compilation. The DATA statement can be placed anywhere in a program unit following any possible specification statements.
- ASSIGN - This statement defines an integer value with the value of a statement label.
- Subprogram invocation - This defines dummy arguments of the subprogram if the corresponding actual arguments are defined and agree with the dummy arguments in data type.
- Return from a subprogram - This defines the actual arguments in the calling program unit that were passed by value to the called subprogram, provided that the dummy arguments are defined in the subprogram.

When any one of these statements except ASSIGN defines an entity, all entities equivalent to it are defined. When a statement defines a complex entity, all real entities associated with the complex entity become defined. Conversely, if the real entities associated with a complex entity become defined, the complex entity is defined.

You must be aware of how entities become undefined in FORTRAN VII. This can occur in the following situations:

- at the beginning of program execution, unless the entities are initialized with a DATA statement,
- when an ASSIGN statement changes the value of an integer variable to a statement label so that the variable becomes undefined as an integer and can no longer be used as an integer value,
- when the RETURN statement is executed at the end of a subprogram (in this case all entities in the subprogram become undefined except entities in common, initialized entities that were not undefined or redefined and entities preserved through a SAVE statement), and
- when an error condition or end of file condition occurs during execution of an input statement, all entities specified in the input list of the statement become undefined.

When an entity becomes undefined, all associated entities of the same data type and all partially associated entities become undefined. Entities are said to be associated if they have the same storage sequence. An EQUIVALENCE statement causes association of entities only within one program unit, unless one of the equivalenced entities is in a COMMON block. Arguments and COMMON statements cause entities in one program unit to become associated with entities in another program unit.

Testing Values of Floating Point Variables

Because of round off errors that occur as a result of floating point computations (See Chapter 15 for different factors affecting floating point calculations), testing floating point entities for exact values (e.g., X.EQ.0.0) can yield inaccurate results. Instead, testing should be done for a range of floating point values (e.g., X.GT.-1E-6.AND.X.LT.1E-6).

Equivalencing Integer to Floating Point Variables

When equivalencing an integer value to a floating point value, the integer value must be capable of being normalized. Certain integers cannot be normalized and produce an arithmetic fault error when the value is loaded into the floating point register.

Example:

```
C THIS EXAMPLE WILL PRODUCE
C AN ARITHMETIC FAULT

      REAL A
      INTEGER I
      EQUIVALENCE (A,I)
      I=1
      A=A+1
```

Improving Program Readability

Unless blanks appear in a Hollerith field or a quoted literal, they are ignored by the FORTRAN compiler. Programmers can use this feature to improve the readability of a program.

Compare Example 1 with Example 2.

Example 1:

```
C THIS EXAMPLE USES BLANKS TO IMPROVE THE
C READABILITY OF THE PROGRAM

      DIST = 0.03 * 0.001
      WRITE(2,10) DIST, DIST AROUND
   10 FORMAT (1X, 2F10.6)
      STOP
      END
```

Example 2:

```
C THIS EXAMPLE IS WRITTEN WITHOUT REGARD
C TO THE READABILITY OF THE PROGRAM

      DIST=0.03*0.001
      WRITE(2,10)DIST,DISTAROUND
   10 FORMAT(1X, 2F10.6)
      STOP
      END
```

Optimizing I/O Operations

An I/O statement takes longer to execute than a non-I/O statement. Thus, I/O execution time is an important factor in program optimization. The following programming techniques can be used on the source level to reduce the time required for I/O operations:

- Use unformatted READ/WRITE statements to reference all files created by the source program to be used as temporary files or as input to other FORTRAN programs.
- Substitute arrays without subscripts for IMPLIED DO statements in READ/WRITE statements.

Example:

```
C THIS EXAMPLE REDUCES THE NUMBER
C OF DATA ITEMS IN AN I/O LIST BY
C USING AN ARRAY WITHOUT SUBSCRIPTS
C RATHER THAN AN IMPLIED DO.
C
C
         DIMENSION IARRAY (20)
          .
          .
          .
         WRITE (6,10) IARRAY,J
  10      FORMAT(I6)
```

This example reduces the number of data items represented by the WRITE statement to two compared to the 21 data items represented by:

```
    WRITE (6,10) (IARRAY(I),I=1,20),J
10  FORMAT(I6)
```

- Avoid the use of dynamic format statements.
- Use the COMMON BLOCK and EQUIVALENCE statements to reduce the number of data items that are referenced in a READ or WRITE statement.

Example:

```
C THIS EXAMPLE STORES J AND IARRAY
C IN A COMMON BLOCK THAT CAN BE
C REFERENCED AS A SINGLE ITEM IN
C AN I/O STATEMENT
C
         DIMENSION IARRAY (20),Q(21)
         COMMON/BLOCK/J,IARRAY
C
C THE FIRST ELEMENT IN COMMON IS
C EQUIVALENCED TO AN ARRAY
C WHICH IS USED IN THE UNFORMATTED
C WRITE STATEMENT
C
C
         EQUIVALENCE (J,Q(1))
         .
         .
         .
         WRITE (6) Q
```

Preparing Code for the Optimizing Compilers

The previous section introduced some user-level coding techniques that can improve the performance of the object code at execution time. This section discusses how the programmer can further enhance performance through the use of the optimizing compilers.

Before introducing coding techniques that can be used with the optimizing compilers, the specific optimizations performed by the compilers will be discussed.

Basic Optimization Concepts

The FORTRAN VII optimizing compilers incorporate optimizing techniques that order and modify the source code to produce an object program that executes faster. Optimizations performed by these techniques are summarized in Table 4-1. These optimizations can be classified as follows:

- Built-in optimizations - These optimizations are performed automatically during all compilations and cannot be turned off by the user through the NOPTIMIZE compiler directive.
- Optional optimizations - These optimizations are performed by the optimizer and can be turned off at compilation time at the discretion of the programmer by using the NOPTIMIZE compiler directive.

NOTE $OPTIMIZE/$NOPTIMIZE do not control the inline expansion feature of the FORTRAN VII optimizing compiler.

Optimization	Class	Cost	Savings	Optimized Entities
Constant computation	Built-in	-	T/S	Expressions
Type conversions	Built-in	-	T/S	Constants, expressions
Symbolic arithmetic simplification	Optional	-	T/S	Expressions
Linearized array references	Built-in	-	T/S	Array subscripts, character substrings
Short circuit logical computation	Built-in	-	T	Logical expressions
Machine instruction choice	Built-in	S*	T/S*	Machine instructions
Expression reordering	Built-in	-	S	Registers
Strength reduction	Built-in/Optional	S	T	Machine instruction
Global register allocation	Optional	-	T/S	Registers
Constant propagation and computation	Optional	-	T/S	Expressions
Invariant code motion	Optional	-	T	DO loop, blocks
Test replacement	Optional	-	T/S	DO loop index
Scalar propagation	Optional	-	T/S	Assignments
Folding	Optional	-	T/S	Assignments
Common subexpression elimination	Optional	-	T/S	Assignments, expressions
Dead code elimination	Optional	-	T/S	Assignments, DO loops

* In some cases, there will be a memory space cost, and in others, a memory space savings; but never both.

T = time

S = memory space

Table 4-1. Compiler Optimizations

Before any built-in or optional optimizations can be performed, the compiler collects information about the program, stores this information in various tables and translates the source code into an intermediate language. In addition, optional optimizations require complete information on the flow of control within a program and the definition and use information for each program variable. The recording of definition and use information for a program variable is known as p-graph analysis. This analysis determines information about all generators of a variable and all the uses of the variable that belong to each generator.

In general, the optimizations performed by the optimizing compilers have no effect on the results. However, if the order of evaluation of an expression greatly affects the results, the expression should be parenthesized to force a particular order of evaluation. See Chapter 14 for details on the possible effects of optimization on floating point calculations.

The following sections discuss the operation of each of the optimizations available on the optimizing compilers. In the following examples, a compiler generated variable is represented by *@n* and a compiler generated label is represented by $L*n*, where *n* is a number chosen by the compiler.

Constant Computation (Built-in)

All arithmetic, relational and logical expressions whose operands are explicitly stated constants are candidates for computation. This optimization saves execution time and memory space.

Example:

Source code before compilation:	Optimized code:
`TEMP = 16.*T/2.+(8.-3.)`	`TEMP = 8.*T+5.`

Compile Time Type Conversions (Built-in)

The compile time type conversion routine evaluates constants and performs type conversions in mixed mode expressions.

Example:

Source code before compilation: Optimized code:

```
Y = Y + 5 + 2                  Y = Y + 7.0
```

In this example, the optimization routine computes the integer expression 5+2 and converts the sum to a real number, 7.0. By performing type conversions during compilation rather than execution, the compiler decreases both time and memory space used during run-time.

Symbolic Arithmetic (Optional)

This optimization routine simplifies symbolic arithmetic expressions as follows:

```
K1*X + K2*X --------> (K1+K2)*X
        X+0 --------> X
        X*1 --------> X
        X/1 --------> X
        X-X --------> 0
        X/X --------> 1
        X*0 --------> 0
```

Example:

Source code before compilation: Optimized code:

```
X=0                        X=0
Y=1                        Y=1
Z=(Y+X)*(Y/Y)              Z=1
```

This optimization saves both execution time and memory space.

Array Linearization (Built-in)

Array linearization replaces multidimensional arrays with a one dimensional array of the same data type. The one dimensional array is then simplified into the following expression:

A(*constant part* + *variable part*)

Example:

```
Source code before compilation:  Optimized code:

REAL*4 A(10,10,10)               REAL*4 A(10,10,10)

DO 10 I=1,10                     DO 10 I=1,10
DO 10 J=1,10                     DO 10 J=1,10
DO 10 K=1,10                     DO 10 K=1,10

                                 @1=4*I+40*J+400*K
A(I,J,K)=A(I,J,K)+1              A(-444+@1)=A(-444+@1)+1

10 CONTINUE                      10 CONTINUE
```

In this example, `A(I,J,K)` is replaced by the linearized array:

A(((I-1)+(J-1)*10+(K-1)*100)*4)

The expression representing the linearized array subscript is then simplified by evaluating its constant and variable parts as follows:

constant part = (-1-10-100)*4
= -444

variable part = @1

@1 = (I+10*J+100*K)*4
= 4*I+40*J+400*K

In this example the linearized array subscript is converted to an actual byte address of a 4-byte location by multiplying the subscript by 4.

Short Circuit Logical Expression Computation (Built-in)

Short circuit logic evaluation is applied to a logical expression within an IF condition in an IF statement. The logical expression consisting of subexpressions connected by .AND., .OR., .EQ., or .NEQ. is evaluated left to right, subexpression by subexpression, until the value for the entire expression can be determined. At that point, the evaluation of any remaining subexpressions is bypassed.

Example:

```
Source code before compilation:           Optimized code:

A = -1                                         A = -1
IF(.NOT.(A.LT.0.OR.B.GT.0))GOTO 10             IF(A.LT.0)GO TO $L1
                                               IF(B.LE.0)GO TO 10
                                          $L1  CONTINUE
```

In the above example, the second IF statement in the optimized code is never executed.

NOTE Due to short circuit logic evaluation, functions contained in a bypassed subexpression will not be invoked.

Machine Instruction Choice (Built-in)

After a source statement is compiled, this optimization orders and modifies the resulting machine instructions so that when executed, the program will take full advantage of the performance features of the processor. The following example optimizes the source statement I = I*6 and, thereby, reduces its execution time.

Example:

```
Machine code after compilation:   Optimized machine code:

L   5,I                           L  5,I  R5=I
M   4,$CONST6                     LR 4,5  R4=I
ST  5,I                           AR 5,5  R5=I+I=2I
                                  AR 5,4  R5=2I+I=3I
                                  AR 5,5  R5=3I+3I=6I
                                  ST 5,I  I=6I
```

Expression Reordering (Built-in)

Another optimization performed on an arithmetic expression by the compiler is expression reordering. This routine reorders the variables in an expression so that the number of registers required for temporary storage is reduced.

Example:

```
Source code before compilation:   Reordered optimized code:

A+B*C                             B*C+A
```

In this example, the resulting machine code is:

```
LE 14,B   R14=B
ME 14,C   R14=B*C
AE 14,A   R14=B*C+A
```

Without reordering, the machine code would be:

```
LE  14,A   R14=A
LE  12,B   R12=B
ME  12,C   R12=B*C
AER 14,12  R14=A+B*C
```

Strength Reduction (Built-in/Optional)

The strength reduction optimizations reduce the strength of an operation by replacing it with another operation that executes faster. The optimizing compilers offer two optimizations for strength reduction. The built-in strength reduction optimization transforms an algebraic expression into another expression that requires fewer operations per computation. By factoring algebraic expressions into simpler terms, this optimization rearranges the code to produce a more efficient sequence of operations. Slower operations (exponentiation, division and multiplication) are replaced by operations that execute at a faster rate. For example, division operations are replaced by multiplication, and multiplication operations are transformed into addition.

Examples:

Source Code Before Compilation	Optimized Code	Net Change In Operations
40*I+4*J	4*(10*I+J)	Faster sequence of operations
(1/B)*A	A/B	Eliminates one multiplication
B**M * B**N	B**(M+N)	Replaces one exponentiation with addition
A**M * B**M	(A*B)**M	Eliminates one exponentiation
(A/C)+(B/C)	(A+B)/C	Eliminates one division
X/4.0	X*.25	Replaces division with multiplication
X**2 - Y**2	(X-Y)*(X+Y)	Replaces two exponentiations with one multiplication
A*X**3 + B*X**2 + C*X	(A*X+B)*X+C)*X	Eliminates two exponentiations

The optional strength reduction optimization changes a multiplication expression (e.g., I*K where I is the DO index and K is the loop invariant of type integer) to an addition.

The following example shows how optional strength reduction can increase optimization after array linearization is performed.

Example:

Source code before compilation:

```
   DIMENSION A(50,50)
   DO 10 I = 1,50
   DO 10 J = 1,50
   A(I,J)= A(I,J)+1

10 CONTINUE
```

Optimized code after array linearization:

```
   DIMENSION A (50,50)
   DO 10 I = 1,50
   DO 10 J = 1,50

   A(-204 + 4*I+200*J) = A(-204 + 4*I + 200*J)+1.0
10 CONTINUE
```

Optimized code after array linearization and optional strength reduction:

```
    DIMENSION A(50,50)
    @1 = 4
    DO $L1  I = 1,50
    @2 = 200
    DO $L2  J = 1,50

    A(-204 + @1 + @2) = A(-204 + @1 + @2) + 1.0

$L2 @2 = @2 + 200
$L1 @1 = @1 + 4
```

In the previous example, @1 replaces I*4 and @2 replaces J*200.

These examples show that the strength reduction optimizing techniques sacrifice memory space to save execution time.

Loop Test Replacement (Optional)

In the last example from the preceding section, the DO loop variables I and J are no longer used with the loop. Loop test replacement replaces unused DO loop variables resulting from strength reduction with compiler generated variables. In this case, the compiler generated variables are @1 and @2. Loop test replacement replaces the initial, final and increment values of the original DO loop with the initial, final and increment values of the most heavily used strength reduction temporary variable.

Example:

Optimized code after
strength reduction:

```
      DIMENSION A(50,50)
      @1=4
      DO $L1 I=1,50
      @2 = 200
      DO $L2 J=1,50
      A(-204 + @1 + @2)=
     1    A(-204 + @1 + @2) + 1.0
$L2   @2 = @2 + 200
$L1   @1 = @1 + 4
```

Optimized code after
strength reduction
and test replacement:

```
      DIMENSION A(50,50)
      DO 10 @1 = 4,200,4
      DO 10 @2 = 200,1000,200
10    A =(-204 + @1 + @2)=
     1 A(-204 + @1 + @2) + 1.0
```

This optimization routine saves both time and space.

Global Register Allocation (Optional)

To reduce the amount of memory space and/or execution time, the compiler can optionally store program variables, constants and dummy array addresses in available registers.

Using p-graph analysis, the compiler determines the definition and use of each variable throughout the program. The compiler then divides the program into sections in which the definition and use of that variable is independent of the definition and use of that variable in the other sections of the program. In other words, each variable is split into a number of separate and independent variables each of which has its own separate section in the program.

The global register allocation routine scans the entire program for definition and use information to determine the payoff value for each variable. The payoff value is equal to the sum of the frequency and expected frequency of the use and redefinition of the variable in a given section.

The payoff values are sorted and after all local register requirements are satisfied, the variables with the highest payoff values are stored in the remaining registers. For example, if variable A is referenced 10 times in a particular section, A has a payoff value of 10. Variable B, which is referenced 5 times, has a payoff value of 5 for that section. If only one register is available, variable A will be stored because of its higher payoff value.

The compiler computes the expected frequency of a variable from the loop structures within the program. Statements within DO loops that have a constant loop count are assigned an expected frequency value equal to the loop count. If the loop count value is not a constant, the compiler assigns a value of 10.

The expected frequency of a set of nested loops is the product of the DO loop counts within the set. The compiler automatically assigns a frequency value of 3 to statements within compiler recognized loops outside any DO statement. All statements outside either type of loop are assigned a frequency value of 1.

The global register allocation routine uses the same methods to store constants and dummy array addresses.

If the $BASE directive is specified, the base addresses of all variables and named common blocks will also be allowed to compete for available registers. The global register allocation routine will sort the payoff values of these base addresses along with the program variables, constants and dummy array addresses. Because of the additional entities competing for registers, $BASE can cause a slight increase in execution time. This increase, however, is offset by the reduction in memory space required by the object code which can now use the base addresses to replace RX3-type instructions with RX2-type instructions. For an explanation of machine instruction format, see the appropriate Concurrent 32-bit processor manual.

Invariant Code Motion (Optional)

This optimization examines a DO loop for invariant operations and moves these operations to a point immediately before the loop entry. An operation is invariant if its operands do not depend on variables that change within the loop. This optimization saves execution time.

For example, if during compilation a single multiplication is moved out of a loop that iterates 1000 times, 999 run-time multiplications are eliminated.

Example:

Source code before compilation: Optimized code:

```
   DO 10 I = 1,10                 @1 = X * Y * Z
   A(I,J,K)= X*Y*Z                DO 10 I=1 ,10
10 CONTINUE                       A(I,J,K) = @1
                               10 CONTINUE
```

An intrinsic function can also be invariant if its arguments do not change within the loop. External subprogram invocations are not considered for loop invariance because their argument values can be altered by the called subprogram.

Constant Propagation and Computation (Optional)

This optimization replaces all variables in an expression with their assigned constant values and then evaluates the expression.

Example:

Source code before compilation: Optimized code:

```
 T = 512.0                     T = 512.0
TEMP= 16.0*T/2.0 + (8.0-3.0)   TEMP= 4101.0
```

In this example, T was replaced by its assigned constant 512.0 to yield the expression:

16.0*512.0/2.0 + (8.0 - 3.0)

This expression was then evaluated to 4101.0.

Constant propagation and computation can be performed on any variable whose value is constant and is assigned through an assignment statement. For example, 4101.0 can be substituted in all subsequent uses of the variable TEMP.

Variables can be replaced by their assigned constants only if the statement that contains the variable does not redefine the constant (e.g., subprogram invocation argument), and the statement cannot be reached from any definition of the variable other than the given constant assignment.

Dead Code Elimination (Optional)

After all uses of a variable are replaced with its assigned constant through constant propagation, the original assignment statement becomes dead code and can be eliminated.

Dead code elimination can also be performed on conditional statements. For example, if an IF statement evaluates to false, the IF statement becomes dead code and is deleted. Conversely, if the IF condition in a statement such as:

```
IF(IA.EQ.2) GO TO 20
```

evaluates to true, the entire IF statement is eliminated and replaced by an unconditional branch to 20.

Invariant code motion can often reduce a DO loop to just two statements, DO and CONTINUE. Dead code elimination removes all such loops.

Examples of dead code that can be removed are:

- DO loops with an iteration count of 0 (in this instance, only the assignment of the initial value to the DO variable is preserved),
- Increment step and loop termination test for a DO loop with an iteration count of 1,
- Logical and arithmetic IF statements that branch to one block of code regardless of condition (these statements are replaced with unconditional GOTO statements),
- Assigned and computed GOTO statements that branch to one block of code (these statements are replaced with unconditional GOTO statements),
- GOTO statements that branch to the statement directly below it,

- GOTO statements branching to other GOTO statements (the program branches to the second GOTO statement label), and
- Blocks of code that become unreachable because of previously performed optimizations.

Scalar Propagation (Optional)

Scalar propagation eliminates assignment statements of the form X=Y (where X is a local variable) and replaces the value assigned to X in appropriate subsequent uses of X.

Example:

```
Source code before compilation:   Optimized code:
X = Y                             A = Y * C
A = X * C                         B = Y * D
B = X * D
```

Scalar propagation occurs only if the value of X was not redefined before using in an expression. X and Y must also be the same data type. The assignment X=Y is deleted if there are no more uses of X in the program unit.

This optimization saves both time and space.

Folding (Optional)

Folding eliminates assignment statements of the form:

X = expression

if X is used only once in the program.

Example:

```
Source code before compilation:   Optimized code:
X = A*B                           Y = A*B + C
Y = X + C
```

In this example, A*B was folded into the expression X+C to eliminate the assignment statement X=A*B. Folding occurs under the following conditions:

- The variable to be folded is used only once throughout the entire program.
- The program executes sequentially from the assignment statement to the statement that uses the variable.
- The components of the expression assigned to the variable are not redefined before the statement where the folding occurs.

Common Subexpression Elimination (Optional)

Common subexpression elimination creates an assignment statement of the form:

@n = subexpression

Where:

subexpression refers to an arithmetic, logical, or relational operation that is used more than once through the program unit. The compiler generated variable *@n* replaces all subsequent uses of the subexpression.

Example:

```
Source code before compilation:   Optimized code:

   A = B*C*D                          @1  = B*D
   X = B*D                            A   = @1*C
   IF  (B.GT.C) GO TO 10              X   = @1
   Y = B*D                            IF(B.GT.C) GO TO 10
   IF (B.LT.C) GO TO 10               Y   = @1
5  B = B+2                            IF(B.LT.C) GO TO 10
10 Z = B*D                         5  B = B+2
                                   10 Z = B*D
```

In this example, all but the last occurrence of the subexpression B*D were replaced with the compiler generated variable @1. B*D in statement 10 is not equivalent to the other occurrences of B*D, because B was assigned a new value in statement 5.

If any assignment is made to any of the components of a subexpression before it is used, the subexpression can no longer be considered for common subexpression analysis. Subexpressions are considered for common subexpression elimination if their operators (* or +) and operands are identical to other subexpressions.

In the code sequence:

```
Y=A+B
X=A
X=X+B
```

the compiler will not recognize the A+B and X+B as common subexpressions even though they have the same value. (It is assumed that folding has not occurred.)

It is not required that unparenthesized subexpressions that are equivalent in regard to operator and operand be identical in the sequence of their operands. For example, A+B+C is equivalent to B+C+A, and A*B*C is equivalent to B*C*A.

The compiler also recognizes partial common subexpressions as equivalent. For example, in the code sequence:

```
Q=A+B+D
R=B+A+C
```

A+B is recognized as a common subexpression. This is also true if the operator was *.

Sequence of Optimization

During compilation the compiler performs optimization in the following order.

```
BEGIN

  Perform symbolic arithmetic and constant computation
  Perform type conversion
  Perform array linearization

    RESTART:  DO until no change
              DO until no change
                 Perform flow analysis deleting unreachable code
                 Perform constant propagation, symbolic arithmetic,
                      and constant computation
                 Attempt to evaluate logical and arithmetic IF,
                      computed GOTOs and DO loops
                      Remove if possible
              END

      Perform p-graphing of scalars and arrays
      Perform scalar propagation, folding, and dead code
           elimination
      Perform array common subexpression analysis
      Perform strength reduction and test replacement
      Perform scalar common subexpression analysis
      Perform invariant code motion
      Evaluate consequences to flow of control; If all
           code is removed from a loop, GO TO RESTART

    END
  Perform register allocation

END
```

Preparing Source Code for the Optimizing Compilers

Certain precautions must be taken to avoid degrading the compilers' optimizing capabilities. To obtain optimum performance from the O and Z compilers, follow these guidelines when preparing the source program.

- Avoid using extended range DO loops. These loops inhibit the performance of the DO loop optimization techniques such as strength reduction, test replacement, and invariant code motion.
- To take advantage of the DO loop optimization techniques available, use DO loops whenever possible to replace IF and GOTO loop structures.

- When coding a logical expression for the IF condition of an IF statement, code subexpressions whose values can determine the value of the entire expression before coding the other subexpressions. For example, the statement:

  ```
  IF ((A.AND.B.AND.C).OR.D) GO TO 10
  ```

  would be more efficiently executed if it were written as:

  ```
  IF (D.OR.(A.AND.B.AND.C)) GO TO 10
  ```

  This increases the efficiency of short circuit logical expression computation.
- Make certain that all variables are defined at their point of entry into a program unit. This will aid the compiler in gathering definition and use information for the optimizer.
- Do not use the RTL EXIT routine. The optimizing compilers treat EXIT as any other subprogram. Consequently, a loop containing a CALL EXIT statement will be treated as an infinite loop or as a program module that has no program exit.
- Avoid excessive use of EQUIVALENCE statements. If two scalar variables are equivalenced, the two are treated as one scalar variable having the combined uses of both scalars. If a scalar is equivalenced to an array, the scalar is treated as an array. This degrades scalar optimization.
- Use the $BASE directive when the combined number of references to individual variables in a named common block is greater than the number of references to any of the local variables. $BASE causes the register allocation routine to store the address of the block rather than the individual block variables, which have less of a chance competing against the local variables for register space than the block itself. This procedure allows both an efficient use of registers and a more compact object program.
- Use the $TCOM directive when optimizing a program that references a common block shared by another running program. $TCOM prevents the optimizer from eliminating code referencing the block or allocating registers for the variables within the block.

Example:

```
C THIS PROGRAM IS DESIGNED TO RUN
C CONCURRENTLY WITH A SECOND PROGRAM
C CALLED EVENT.TSK WHICH ALSO REFERENCES
C COMMON ABC.
C THIS EXAMPLE USES $TCOM TO DECLARE
C WHICH COMMON BLOCK VARIABLES ARE
C PART OF A TASK COMMON SEGMENT ABC AND
C SHOULD NOT BE CONSIDERED AS CANDIDATES
C FOR OPTIMIZATION.
C
C
$TCOM/ABC/
C
      COMMON/ABC/EVENT
      LOGICAL EVENT
      EVENT = .FALSE.
10    CONTINUE
      IF(.NOT.EVENT) GO TO 10
      STOP
      END
```

In this example, $TCOM prevents dead code elimination of all code referring to EVENT.

IF $TCOM was not specified, the optimizer would have scanned the program for code that would have propagated the value of EVENT into the IF statement. The IF statement would evaluate to TRUE and would be converted into an unconditional GOTO to label 10. Consequently, the compiler would have eliminated the IF statement during compilation even though EVENT.TSK alters EVENT.

$TCOM also prevents the compiler from storing common block variables in available registers. If EVENT was stored in a register during compilation, the above program would reference the register; therefore, the program will not be informed when EVENT is changed by EVENT.TSK. Even with the $NOPTIMIZE option specified, it is important that $TCOM be used whenever shared COMMON exists. $NOPTIMIZE does not guarantee that the optimizing compilers do not perform register allocation.

The $TCOM statement prevents the optimizer from changing the meaning of the user program when task common references are involved. However, this statement does not provide mutual exclusion on access to shared COMMONs. Two RTL routines, LOKON and LOKOFF, are provided to enable the user to implement mutual exclusion and to avoid the problems of degraded optimization which can result from the excessive use of the $TCOM directive. These routines take a single argument which is normally an INTEGER*2 variable in task common. LOKON is a LOGICAL function which performs a "test and set" on its argument and returns .TRUE. if the sign bit of its argument is set. Otherwise it returns .FALSE. and sets the sign bit of its argument. The subroutine LOKOFF unconditionally resets the sign bit of its argument.

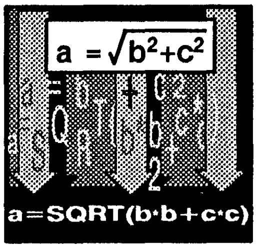
The argument of LOKON and LOKOFF is always treated as if it occurred in a $TCOM statement.

Example:

```
      SUBROUTINE UPDATE
      INTEGER*2 KEY
      COMMON /ABC/ KEY, IDATA (10)
          .
          .
          .
C
C     WAIT FOR COMMON TO BECOME ACCESSIBLE
   10 IF (LOKON (KEY)) GO TO 10
          .
          .
          .
C     USE OR SET IDATA ELEMENTS
          .
          .
          .
C
C     ALLOW OTHER TASK TO ACCESS COMMON AGAIN
C
      CALL LOKOFF(KEY)
          .
          .
          .
      RETURN
      END
```

This example illustrates two important points. First, KEY is treated as if it were mentioned in the $TCOM directive. Therefore, the optimizer can register allocate and optimize references to IDATA. This can be done safely assuming that other tasks referencing /ABC/ also observe the locking mechanism since the optimizer will not move any references to COMMON areas across calls to LOKON and LOKOFF. While LOKON is functionally identical to TESET and LOKOFF is identical to a reset bit of the high order bit of KEY, this property of blocking the optimizer from moving COMMON references is important in preserving the program's original intent. TESET should never be used as a substitute for LOKON. An assignment statement is never a substitute for LOKOFF.

The protection provided by LOKON and LOKOFF is not a positive protection method. It relies on the strict observance of the locking convention by every task referencing the task common. Members of the 'locked' common should not be referenced outside the scope of a LOKON/LOKOFF block.



5

Interfacing Assembly Language Routines

In this chapter

We provide information on how to interface FORTRAN programs with assembly language subprograms. This capability allows you to use any existing assembly-written routine that may be useful for your task. You can also write a routine in assembly language which is otherwise impractical to code in FORTRAN and interface this routine with your FORTRAN program.

Topics include:

- Learning the standard FORTRAN calling sequence
- Embedding assembly language blocks in FORTRAN source

Knowing Your Options

There are two options for interfacing assembly language subprograms with a FORTRAN program:

- Writing assembly language routines that can be called by the FORTRAN program as any other routine. These assembly language routines can be called by all FORTRAN VII programs if the subprograms have the proper interface.
- Directly embedding assembler code into the FORTRAN source code using the FORTRAN VII compiler directives.

Guidelines to using both options are discussed in the following sections.

Standard FORTRAN Calling Sequence

The FORTRAN VII compiler automatically sets up the necessary interface used by a FORTRAN program to communicate with a subprogram written in FORTRAN. Assembly language subprograms must use the same interface.

The following sections contain the calling sequences to FORTRAN VII.

Passing Arguments

The standard FORTRAN VII interface passes arguments through general purpose register 14 (GPR14). GPR14 contains the subprogram type field in the high order byte and the address of the argument list in the next three low order bytes. The address of the argument list is limited to the first 16MB (24-bit addressing). The argument list pointed to by GPR14 consists of two distinct data structures:

- Argument address list (AAL) and
- Argument descriptor list (ADL).

Figure 5-1 illustrates the argument list structure.

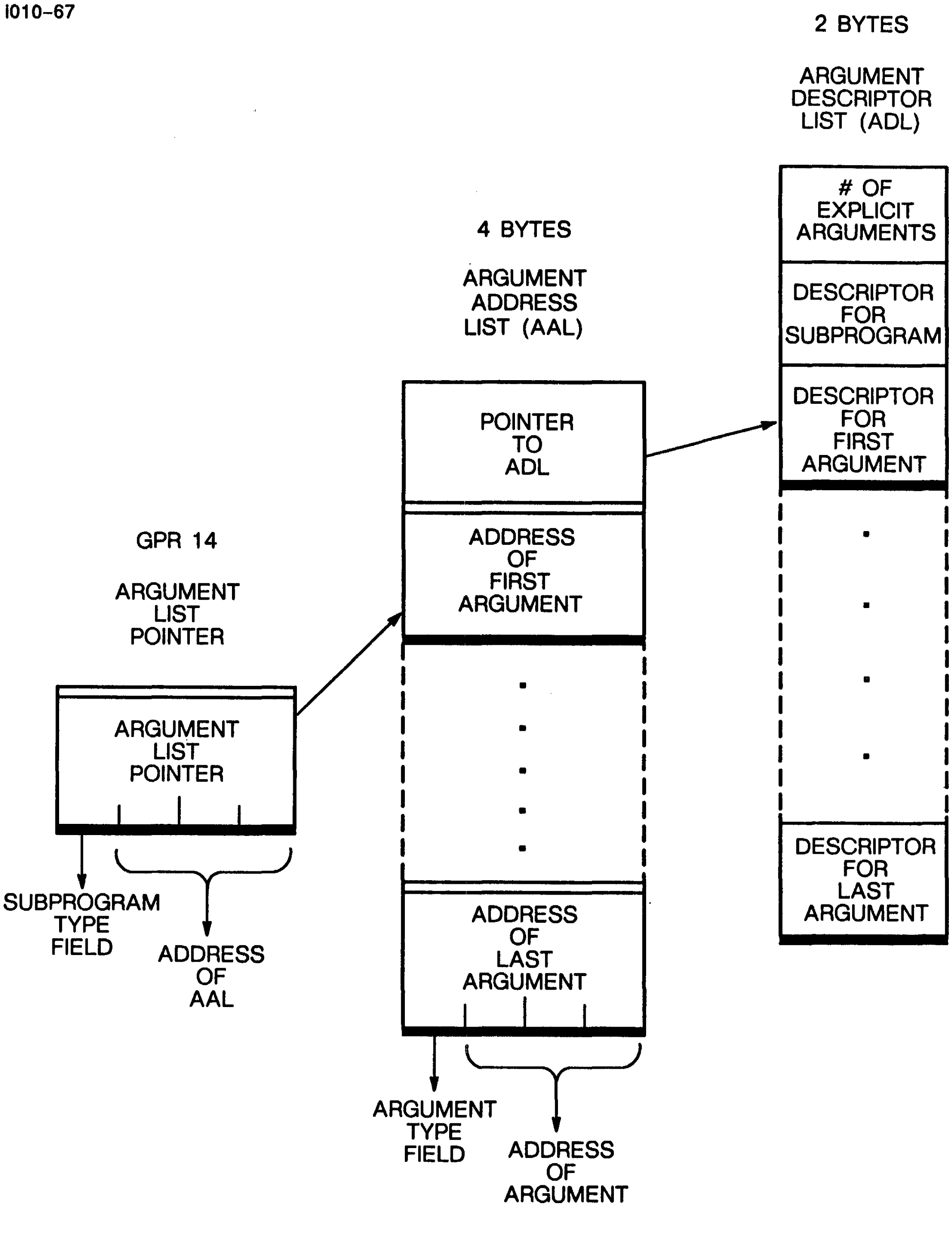


Figure 5-1. Argument List Structure

If no argument list exists for the subprogram, the first entry of the ADL, which contains the number of explicit arguments in the argument list, will be 0.

The subprogram type field in GPR14 is defined as in Table 5-1:

Bits	Definition	Value	Effect
0	Arguments passed	0 1	No Yes
1-3	Subprogram type	5 6 7	Function Subroutine 5 or 6
4-7	Function result type	1 2 3 4 5 6-7 8 9 10 11 12-13 14 15	INTEGER*2 INTEGER*4 LOGICAL*1 LOGICAL*4 CHARACTER Reserved COMPLEX*16 REAL*4 COMPLEX*8 REAL*8 Reserved LOGICAL*2 INTEGER*1

Table 5-1. GPR14 Subprogram Type Field

NOTE ▷ The subprogram type field in GPR14 is maintained for compatibility with FORTRAN Version or release R05-05 and earlier versions of the compiler. All information contained in this field is duplicated in the ADL as described below. The information is represented differently.

GPR14 points to the second entry of the AAL (corresponding to the address of the first argument). The first entry in the AAL is a pointer to the third entry in the ADL. The AAL must be fullword aligned and contain one argument per word. Except for the first entry in the AAL, each argument word consists of an argument type byte and the argument address in the next three low order bytes (24-bit addressing).

Figure 5-2 illustrates the AAL entry:

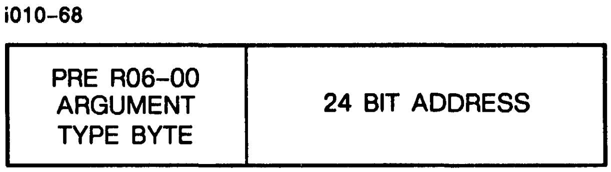


Figure 5-2. AAL Entry Structure

The argument type byte is defined in Table 5-2:

Bits	Definition	Value	Effect
0	Argument position	0	Not last argument
		1	Last argument
1-3	Argument class	0	Expression or constant
		1	Variable
		2	Array element
		3	Array
		4	Reserved
		5	Function
		6	Subroutine
		7	Either 5 or 6
4-7	Argument type	1	INTEGER*2
		2	INTEGER*4
		3	LOGICAL*1
		4	LOGICAL*4
		5	CHARACTER††
		6-7	Reserved
		8	COMPLEX*16
		9	REAL*4
		10	COMPLEX*8
		11	REAL*8
		12	Alternate return label
		13	Hollerith
		14	LOGICAL*2
		15	INTEGER*1

†† If the argument type is CHARACTER, the word following the address of the CHARACTER is the address of its length.

Table 5-2. AAL Argument Type Byte

Entries in the ADL consist of two bytes each. The first entry contains the number of explicit arguments in the argument list (this entry may be equal to 0). The second entry contains the subprogram descriptor as illustrated in Figure 5-3:

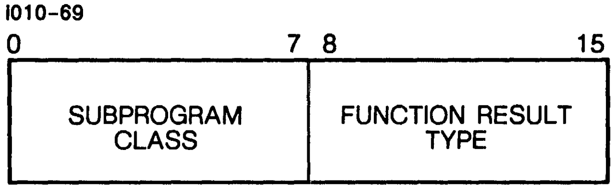


Figure 5-3. ADL Entry (Subprogram Descriptor) Structure

The subprogram descriptor is defined in Table 5-3:

Bits	Definition	Value	Meaning
0-7	Arguments passed Subprogram Class	0 1 5 6 7	No Yes Function Subroutine 5 or 6
8-15	Function Result Type	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	For SUBROUTINE LOGICAL*1 LOGICAL*2 LOGICAL*4 Reserved INTEGER*1 INTEGER*2 INTEGER*4 Reserved REAL*4 REAL*8 Reserved COMPLEX*8 COMPLEX*16 Reserved CHARACTER

Table 5-3. ADL Argument and Subprogram Descriptors

The succeeding entries in the ADL are the descriptors for the corresponding AAL entries, as illustrated in Figure 5-4.

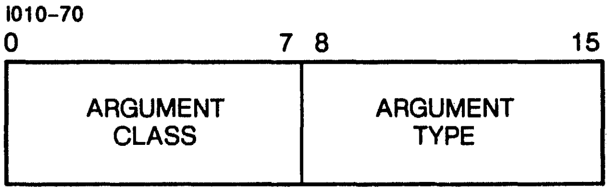


Figure 5-4. ADL Entry (Argument Descriptor) Structure

The value of these argument descriptors are defined in Table 5-4:

Bits	Definition	Value	Meaning
0-7	Argument Class	0	Expression or constant
		1	Variable
		2	Array element
		3	Array
		4	Reserved
		5	Function
		6	Subroutine
		7	5 or 6
		8-255	Reserved
8-15	Argument Type	0	Reserved
		1	LOGICAL*1
		2	LOGICAL*2
		3	LOGICAL*4
		4	Reserved
		5	INTEGER*1
		6	INTEGER*2
		7	INTEGER*4
		8	Reserved
		9	REAL*4
		10	REAL*8
		11	Reserved
		12	COMPLEX*8
		13	COMPLEX*16
		14	Reserved
		15	CHARACTER
		16	CHARACTER length
		17	Alternate return
		18	Hollerith
		19-255	Reserved

Table 5-4. ADL Descriptors Corresponding to AAL Entries

The argument list may contain arguments that may not be explicitly known to the user such as the length of character string arguments. These length arguments follow the address of the corresponding character argument in the AAL. The descriptor for each follows the descriptor for the corresponding character argument.

Passing the Return Address

The compiler generates the following code to branch to the subprogram PASS.

```
BAL 15,PASS
```

Every call to a subprogram by a FORTRAN task uses GPR15 to store the return address for the subprogram. Thus an assembly language routine should use the address in this register when returning to the calling FORTRAN routine.

Run-Time Library (RTL) Scratchpad

To store data during execution, subprograms need a scratchpad area. The FORTRAN VII compilers generate a call to the RTL routine .U to obtain RTL scratchpad area and initialize the FORTRAN environment. This scratchpad is an impure area 600 hexadecimal bytes in size. GPR1 always points to the top of this area. This register must never be corrupted. Individual routines use the scratchpad area by decrementing GPR1 by the desired number of bytes at the beginning of execution and restoring GPR1 to its initial value before returning.

Example:

```
SCRATCH   STRUC                        DEFINE SCRATCH AREA
SG14      DSF    2
WORK      DSF    2
          ENDS
          ENTRY  EXAMPLE
EXAMPLE   EQU    *
          SHI    1,SCRATCH             GET SPACE FROM SCRATCHPAD
          STM    14,SG14(1)            SAVE REGISTERS
          .
          .
          .
          LM     14,SG14(1)            RESTORE REGISTERS
          AHI    1,SCRATCH             RELEASE SCRATCHPAD AREA
          LR     13,13                 SET CONDITION CODE
          BR     15                    RETURN
          END
```

User-written routines can participate in this scheme in a like manner or they can obtain and release storage via supervisor call 2 (SVC2) or the assembly routines discussed later in this chapter. However, care must always be exercised to save and restore all registers except those that must be altered to conform to standard conventions. To terminate the FORTRAN task, the compilers generate a call to the RTL routine .V to release storage and end the task.

Function Results and Condition Codes (CCs)

A routine that is called as a function by a FORTRAN program must place the result of the function in a specific register as shown in Table 5-5.

Function Type	Register	Instruction Setting CC
LOGICAL*1	GPR13	LB followed by LR
LOGICAL*2	GPR13	LH
LOGICAL*4	GPR13	L
INTEGER*1	GPR13	LB followed by LR
INTEGER*2	GPR13	LH
INTEGER*4	GPR13	L
REAL*4	FPR14	LE
REAL*8	DPR14	LD
COMPLEX*8:		
real part	FPR12	No CC set
imaginary part	FPR14	No CC set
COMPLEX*16:		
real part	DPR12	No CC set
imaginary part	DPR14	No CC set

Table 5-5. Required Register and CC Settings for Assembly Language Functions

The register into which the result is stored depends on the function data type. In addition, all data types except complex and character require setting the proper CC before returning control to the calling program. In other words, the last instruction prior to the branch which returns control to the calling program should load the resultant value of the function into the proper register. For example, for the following code to evaluate properly, the CC must be set prior to BZ $P10.

```
BAL  15,ARTN
BZ   $P10
```

The above code is a partial translation of the FORTRAN statement:

```
IF(ARTN(I).EQ.0) GO TO 10
```

Where:

ARTN(I) is a function call with argument I.

The return code from routine ARTN may look like the following:

```
ENTRY ARTN
   .
   .
   .
L    14,xxxx        ;load result - set CC
BR   15             ;branch back to caller
END
```

Calling and Receiving Sequences

Before writing an assembly language routine, it is helpful to examine the instructions generated by the compiler when a program calls a subroutine written in FORTRAN. The following example illustrates code generated from a CALL with many different argument types. It can be used as a guide when writing assembly language routines to be included in a FORTRAN program.

Example of Calling Sequence:

```
        LOGICAL*4 LOG,LOG1*1   ;LOGICAL*4 AND LOGICAL*1 VARIABLES
        LOGICAL*2 LOG2         ;LOGICAL*2 VARIABLE
        INTEGER*2 I2,I*4       ;INTEGER*2 AND INTEGER*4 VARIABLES
        INTEGER*1 I1           ;INTEGER*1 VARIABLE
        REAL*4 REAL,DPREC*8    ;REAL AND DOUBLE PRECISION VARIABLES
        COMPLEX*8 COMPLEX      ;COMPLEX*8 VARIABLE
        COMPLEX*16 CMP16       ;DOUBLE PRECISION COMPLEX
        INTRINSIC SQRT         ;INTRINSIC ROUTINE SQRT
        CHARACTER*4 CHAR       ;CHARACTER VARIABLE OF LENGTH 4
        INTEGER*4 ARRAY(10)    ;INTEGER*4 ARRAY
        PARAMETER(IONE=1)      ;SYMBOLIC NAME OF CONSTANT 1
C
        CALL PASS(LOG,LOG1,LOG2,I,I1,I2,REAL,DPREC,
     +     COMPLEX,CMP16,5H12345,SQRT,*1,
     +     *10,CHAR,*102,ARRAY,IONE)
1       CONTINUE
10      CONTINUE
102     CONTINUE
        END
```

The assembly code translation of the above FORTRAN source code follows.

```
             BATCH
         TARGT 32
_MAIN    PROG  _MAIN
IONE     EQU   1
         ALIGN 4                 FULLWORD ALIGN
$LOCAL   DS    96
ARRAY    EQU   $LOCAL
CMP16    EQU   $LOCAL+40
COMPLEX  EQU   $LOCAL+56
DPREC    EQU   $LOCAL+64
REAL     EQU   $LOCAL+72
I        EQU   $LOCAL+76          DEFINE LOCAL STORAGE (IMPURE)
LOG      EQU   $LOCAL+80
I2       EQU   $LOCAL+84
LOG2     EQU   $LOCAL+86
CHAR     EQU   $LOCAL+88
I1       EQU   $LOCAL+92
LOG1     EQU   $LOCAL+93
$LOCEND  EQU   *
         ORG   $LOCEND
         PURE
         ALIGN 4                   FULLWORD ALIGN
$CONST   EQU   *                   DEFINE PURE CONSTANT 4
         DC    F'4'
         ALIGN 4                   FULLWORD ALIGN
         EXTRN __U
         EXTRN __V                DECLARE EXTERNS OF BOTH IMPLICITLY
         EXTRN _SQRT              AND EXPLICITLY REFERENCED
         EXTRN PASS               EXTERNAL ROUTINES
         ENTRY _MAIN
_MAIN    EQU   *
         LIS   14,2
         BAL   15,__U             SET UP ENVIRONMENT AND GET STORAGE
         LI    14,$L000+Y'D2000000'  LOAD G14 WITH CLASS OF CALL
         BAL   15,PASS            BAL TO SUBPROGRAM
$P1      EQU   *
$P10     EQU   *                  GENERATE FORTRAN LABELS(ADDRESSES)
$P102    EQU   *
         BAL   15,__V
         ALIGN 4      FULLWORD ALIGN
         ALIGN 4
         DC    A($L001)            BEGINNING OF ARGUMENT LIST (AAL)
$L000    EQU   *
         DC    A(LOG+Y'14000000')
         DC    A(LOG1+Y'13000000')
         DC    A(LOG2+Y'1E000000')
         DC    A(I+Y'12000000')
         DC    A(I1+Y'1F000000')
         DC    A(I2+Y'11000000')
         DC    A(REAL+Y'19000000')
         DC    A(DPREC+Y'1B000000')
         DC    A(COMPLEX+Y'1A000000')
         DC    A(CMP16+Y'18000000')
         DC    A($KONST+Y'0D000000')
         DC    A(_SQRT+Y'79000000')
         DC    A($P1+Y'0C000000')
         DC    A($P10+Y'0C000000')
         DC    A(CHAR+Y'15000000')
         DC    A($CONST+Y'02000000')
         DC    A($P102+Y'0C000000')
         DC    A(ARRAY+Y'32000000')
         DC    A($KONST+8+Y'82000000')      LAST ARGUMENT
         ALIGN 2
         DC    X'0013'                      BEGINNING OF ARGUMENT
                                            DESCRIPTOR LIST (ADL)
         DC    X'8507'
$L001    EQU   *
         DC    X'0103'
         DC    X'0101'
         DC    X'0102'
         DC    X'0107'
         DC     X'0105'
         DC     X'0106'
         DC     X'0109'
         DC     X'010A'
         DC     X'010C'
         DC     X'010D'
         DC     X'0012'
         DC     X'0709'
         DC     X'0011'
         DC     X'0011'
         DC     X'010F'
         DC     X'0010'
         DC     X'0011'
         DC     X'0307'
         DC     X'0007'
         IMPUR
         ALIGN 4                          FULLWORD ALIGN
         ALIGN 4
         ALIGN 4                          FULLWORD ALIGN
$KONST   EQU    *
         DB     C'12345    '              DEFINE IMPURE CONSTANTS
         DC     F'1'                      (HOLLERITH AND INTEGER*4 1)
         END    _MAIN
```

The compiler generates the necessary receiving sequence for a FORTRAN subprogram. The following receiving sequence illustrates the code output by the compiler for the subroutine PASS, which was called from the main program in the example above. Note that none of the arguments passed to PASS are actually used except I. Therefore, there is no need for the compiler to generate any code for any argument but I. The code shown is from the F7O compiler without optimization.

Example of Receiving Sequence:

```
C
       SUBROUTINE PASS(LOG,LOG1,LOG2,I,I1,I2,REAL,DPREC,
     +     COMPLEX,CMP16,HOL,FUNKY,*,
     +     *,CHAR,*,ARRAY,IONE)
       LOGICAL*4 LOG,LOG1*1   ;LOGICAL*4 AND LOGICAL*1 VARIABLES
       LOGICAL*2 LOG2         ;LOGICAL*2 VARIABLE
       INTEGER*2 I2,I*4       ;INTEGER*2 AND INTEGER*4 VARIABLES
       INTEGER*1 I1           ;INTEGER*1 VARIABLE
       REAL*4 REAL,DPREC*8    ;REAL AND DOUBLE PRECISION VARIABLES
       COMPLEX*8 COMPLEX      ;COMPLEX*8 VARIABLE
       COMPLEX*16 CMP16       ;DOUBLE PRECISION COMPLEX
       INTRINSIC SQRT         ;INTRINSIC ROUTINE SQRT
       CHARACTER*4 CHAR       ;CHARACTER VARIABLE OF LENGTH 4
       INTEGER*4 ARRAY(10)    ;INTEGER*4 ARRAY
       RETURN I
       END
```

The assembly code translation of the above FORTRAN source code follows.

```
              BATCH
         TARGT 32
PASS     PROG  PASS
         ALIGN 4                     FULLWORD ALIGN
$LOCAL   DS    80
CMP16    EQU   $LOCAL
COMPLEX  EQU   $LOCAL+16
DPREC    EQU   $LOCAL+24
ARRAY    EQU   $LOCAL+32
@100     EQU   $LOCAL+36
IONE     EQU   $LOCAL+40
CHAR     EQU   $LOCAL+44
FUNKY    EQU   $LOCAL+48             DEFINE LOCAL STORAGE (IMPURE)
HOL      EQU   $LOCAL+52
REAL     EQU   $LOCAL+56
I1       EQU   $LOCAL+60
I        EQU   $LOCAL+64
LOG2     EQU   $LOCAL+68
LOG      EQU   $LOCAL+72
I2       EQU   $LOCAL+76
LOG1     EQU   $LOCAL+78
$LOCEND  EQU   *
         ORG   $LOCEND
         PURE
         ALIGN 4                  FULLWORD ALIGN
$CONST   EQU   *
         ALIGN 4                  FULLWORD ALIGN
         ENTRY PASS               DECLARE PROGRAM NAME
PASS     EQU   *
         ST    15,$L001
         L     15,12(14)
         L     13,0(15)
         ST    13,I
         L     15,I
         ST    15,@100
         B     $L000
$L000    EQU   *
         L     15,$L001
         L     13,@100
         BR    15
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
         ALIGN 2
         IMPUR
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
$L001    EQU   *
         DC    Y'00000000'
         ALIGN 4                  FULLWORD ALIGN
$KONST   EQU   *
         END
```

The following is an example of a calling convention for a character function.

```
CHARACTER *10 CHFUNC
CHARACTER *10 RESULT
RESULT = CHFUNC(I)
END
```

The following is the assembly translation of the preceding FORTRAN source code.

```
         BATCH
         TARGT 32
_MAIN    PROG  _MAIN
         ALIGN 4                  FULLWORD ALIGN
$LOCAL   DS    24
@100     EQU   $LOCAL+14
I        EQU   $LOCAL
RESULT   EQU   $LOCAL+4
$LOCEND  EQU   *
         ORG   $LOCEND
         PURE
         ALIGN 4                  FULLWORD ALIGN
$CONST   EQU   *
         DC    F'10'
         ALIGN 4                  FULLWORD ALIGN
         EXTRN __U
         EXTRN __V
         EXTRN CHFUNC
         ENTRY _MAIN
_MAIN    EQU   *
         LIS   14,2
         BAL   15,__U
         LI    14,$L000+Y'E0000000'
         BAL   15,CHFUNC
         BAL   15,__V
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
         DC    A($L001)
$L000    EQU   *
         DC    A(RESULT+Y'15000000')
         DC    A($CONST+Y'02000000')
         DC    A(I+Y'92000000')
         ALIGN 2
         DC    X'0003'
         DC    X'8600'
$L001    EQU   *
         DC    X'010F'
         DC    X'0010'
         DC    X'0107'
         IMPUR
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
         ALIGN 4                  FULLWORD ALIGN
$KONST   EQU   *
         END   _MAIN
```

Sharing Data

In addition to passing arguments to subprograms, data can be shared through the use of COMMON statement.

Common blocks are declared in Common Assembly Language (CAL) by embedding the definitions between the COMN and ENDS statements. The COMN statement is labeled with the common block name truncated to eight characters. If the label has less than eight characters, the compiler will add a period after the last character (e.g.; ABC., DEF.). Blank common has the external name: "//". This is illustrated in the following example.

Example:

```
COMMON I2, DPREC, REALNUM (5,3) /A/I, LOG
INTEGER*2 I2
DOUBLE PRECISION DPREC
LOGICAL LOG
EQUIVALENCE (REAL23, REALNUM(2,3))
```

The program segment is translated to the following CAL representation:

```
         BATCH
         TARGT 32
_MAIN    PROG  _MAIN
//       COMN
         ALIGN 8
I2       EQU   ..
DPREC    EQU   ..+4
REALNUM  EQU   ..+12
REAL23   EQU   ..+56
         ENDS
A.       COMN
         ALIGN 8
A..      DS    8
I        EQU   A..
LOG      EQU   A..+4
         ENDS
         ALIGN 4                  FULLWORD ALIGN
         PURE
         ALIGN 4                  FULLWORD ALIGN
$CONST   EQU   *
         ALIGN 4                  FULLWORD ALIGN
         EXTRN __U
         EXTRN __V
         ENTRY _MAIN
_MAIN    EQU   *
         LIS   14,2
         BAL   15,__U
         BAL   15,__V
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
         ALIGN 2
         IMPUR
         ALIGN 4                  FULLWORD ALIGN
         ALIGN 4
         ALIGN 4                  FULLWORD ALIGN
$KONST   EQU   *
         END   _MAIN
```

An INTEGER*2 and LOGICAL*2 data type must be aligned on a halfword boundary. CHARACTER, INTEGER*1, LOGICAL*1, and BYTE data types are aligned on any byte boundary. All other types are aligned on a fullword boundary.

The EQUIVALENCE statement assigns the same storage area for the equivalenced variables. Thus, REAL23 and REALNUM(2,3) are assigned the location ..+56. This is implied for REALNUM(2,3) since REALNUM equates to ..+12. When equivalencing arrays in CAL, remember that FORTRAN is column-major (ie., the left-most indexes vary most rapidly).

Pseudo data types, such as labels and external subprogram names, are stored as 3-byte address constants, right-justified on a fullword boundary.

Calling Intrinsic Subprograms from Assembly Program

The FORTRAN VII RTL provides a number of intrinsic subprograms that can be called by an assembly language subprogram. The internal names for these RTL intrinsic subprograms are listed in Appendix A. When called by their internal names, intrinsic subprograms communicate with the calling program through a different interface. This interface stores arguments in registers as follows.

Subprogram Type	**Passing Argument Type**	**Register Argument Passed To**
Subroutine/Function	INTEGER**n* LOGICAL**n* REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	GPR14 GPR14 FPR14 DPR14 FPR12 and 14 DPR12 and 14
	Resulting Argument Type	**Register Result Passed To**
Function	INTEGER**n* LOGICAL**n* REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	GPR13 GPR13 FPR14 DPR14 FPR12 and 14 DPR12 and 14

Inserting an Assembly Block in Source Code

With FORTRAN VII, it is not necessary to write separate assembly language subprograms to use the standard FORTRAN subprogram interface. The user can develop an assembly subprogram with a SUBROUTINE or FUNCTION statement so that the compiler automatically sets up the necessary receiving sequence. An assembly block (enclosed between $ASSM and $FORT) may be enclosed between the SUBROUTINE/FUNCTION and END statements.

Example:

```
$TITLE FORTRAN VII WITH EMBEDDED CAL BLOCKS

C THIS FORTRAN PROGRAM ILLUSTRATES THE
C USE OF DIRECTIVES FOR EMBEDDING CAL
C BLOCKS
C FORTRAN CODE BEGINS HERE
      INTEGER  PR
      DIMENSION SAVE1(2)
      READ (*,10) J1,J2,JTEMP1,JTEMP2
10    FORMAT(4I)
C THE FOLLOWING DIRECTIVES ARE USED
C TO INSERT THE CAL BLOCK
C CAL BLOCK BEGINS HERE
$ASSM
$USES    J1,J2,JTEMP1,JTEMP2
$SETS    JTEMP1,SAVE1,JTEMP2
$GOES    20,30
*
* THIS ROUTINE INCREMENTS JTEMP1 BY 15
* AND SUBTRACTS 1 FROM JTEMP2.
* IF THE NEW VALUE OF JTEMP2 EQUALS ZERO, THE PROGRAM
* FALLS THROUGH OUT OF THE CAL BLOCK
* TO FORTRAN STATEMENT 30.  IF JTEMP2
* DOES NOT EQUAL ZERO, THE PROGRAM
* BRANCHES TO FORTRAN STATEMENT 20.
*
                     ST    3,SAVE1    SAVE REGISTERS
                     ST    4,SAVE1+4
                     L     3,JTEMP2   PUT JTEMP2 IN REGISTER 3
                     L     4,JTEMP1   PUT JTEMP1 IN REGISTER 4
                     AIS   4,15       ADD 15 TO JTEMP1
                     ST    4,JTEMP1   STORE NEW VALUE OF JTEMP1
                     SIS   3,1        SUBTRACT 1 FROM JTEMP2
                     ST    3,JTEMP2   STORE NEW VALUE OF JTEMP2
                     BZ    ZERO       BRANCH IF JTEMP2=0
                     L     3,SAVE1    RESTORE REGISTERS
                     L     4,SAVE1+4
                     B     $P20       GO TO STATEMENT 20
               ZERO  L     3,SAVE1    RESTORE REGISTERS
                     L     4,SAVE1+4
*
* CAL BLOCK ENDS HERE
*
$FORT
C FORTRAN CODE RESUMES HERE
30 CONTINUE
20 CONTINUE
        .
        .
        .
       END
```

The previous example illustrates five of the directives used to embed CAL blocks within the source program.

These directives have the following format:

$ASSM

$FORT

$GOES $\left[\mathit{arg}_1,\mathit{arg}_2,\ldots\mathit{arg}_n\right]$

$REGS R*x* ,F*y* ,D*z*

$SETS $\left[\mathit{arg}_1,\mathit{arg}_2,\ldots\mathit{arg}_n\right]$

$USES $\left[\mathit{arg}_1,\mathit{arg}_2,\ldots\mathit{arg}_n\right]$

$ASSM

$ASSM indicates the beginning of an embedded CAL block and is placed before the first line of CAL code.

$FORT

$FORT indicates the end of the CAL block. $FORT is placed immediately after the last line of the CAL block.

Because the compiler does not translate CAL blocks, it must be informed when the block uses or modifies the value of a FORTRAN variable. Otherwise, the correct value of the variable cannot be guaranteed outside the block. The compiler must also be informed of the FORTRAN statements to which the CAL block branches.

$GOES

$GOES lists the labels of FORTRAN statements to which the embedded CAL code branches. For example, in the sample program shown, the last line of the CAL block falls through to FORTRAN statement 30. Statement 30 must be an argument to $GOES. Statement 20 is also an argument of $GOES since the embedded CAL block conditionally branches to FORTRAN statement 20.

If a $GOES directive with no arguments appears in a CAL block, the compiler assumes that no transfer is made from the CAL block back to the FORTRAN source. If no $GOES directive appears in a CAL block, the compiler assumes that the CAL block only falls through to the FORTRAN statements following the CAL block.

If a $GOES directive is specified, and control of the program falls through to the first statement of the FORTRAN code after the block, a labeled statement must follow $FORT and the label of that statement must be specified in a $GOES directive.

$GOES, $USES, $REGS, and $SETS must be placed inside the CAL block between $ASSM and $FORT.

$REGS

$REGS indicates the registers that are modified by the CAL block. This information is used by the optimizer when allocating global registers. See Chapter 4 for more information on global register allocation. The format of the $REGS directive is:

$REGS R*x* ,F*y* ,D*z*

Where:

R*x* specifies the general purpose registers *x* through 15 that are modified in the CAL block. The number of the first register is indicated by *x*, which is an unsigned integer between 0 and 15 inclusive. *x* may not be 1.

F*y* specifies the floating point registers *y* through 14 inclusive that are modified in the CAL block. The number of the first floating point register is indicated by *y* which is an unsigned even integer between 0 and 14 inclusive.

D*z* specifies the double precision registers *z* through 14 inclusive that are modified in the CAL block. The number of the first double precision register is indicated by *z* which is an unsigned even integer between 0 and 14 inclusive.

If $REGS is not specified, the programmer must make certain that the CAL block saves the contents of any registers before they are changed and restores them before the program exits the block. If the CAL block in the sample program did not save and restore registers 3 and 4, $REGS must be coded above the first line of CAL code as follows:

```
    $REGS R3
```

This $REGS directive tells the compiler that general purpose registers 3 through 15 are modified in the CAL block. If the syntax of $REGS is incorrect, the compiler assumes that all registers are set in the CAL code.

$USES, $SETS, $REGS, and $GOES only apply to the CAL block in which they appear. They must be placed inside the CAL block between $ASSM and $FORT.

    $SETS

$SETS informs the compiler which FORTRAN variables are modified in the CAL block so that the modified value of those variables is available outside the CAL block.

If a $SETS directive with no arguments appears in a CAL block, the compiler assumes that no array or variable is modified in the CAL block. If no $SETS directive appears within a CAL block, the compiler assumes that all variables and arrays are modified in the CAL block. A $USES or $SETS directive must be used for every variable that appears only in a FORTRAN specification statement in a CAL block. These directives prevent the compiler from omitting the variable from the generated CAL code.

    $USES

$USES informs the compiler which FORTRAN variables are used in the CAL block so that the correct value of those variables is available inside the CAL block. Arguments cannot be names of COMMON blocks, but they may be names of variables within COMMON blocks.

If a $USES directive with no arguments appears in a CAL block, the compiler assumes that no array or variable is used in the CAL block. If the $USES directive is missing in a CAL block, the compiler assumes that all variables and arrays are referenced in the CAL block and the compiler will generate a warning message to that effect.

Guidelines for Embedding Assembly Blocks

Follow these guidelines when using the CAL directives to insert assembly blocks in a FORTRAN program.

- Do not use the same name to identify a variable in the FORTRAN block and a label in the embedded CAL.
- The following conventions must be followed when referencing FORTRAN generated symbols or FORTRAN labels.

$P*n* corresponds to the FORTRAN label *n*.

$L*n* are internal labels generated by compiler (user should avoid this type of symbol).

$CONST are pure generated constants.

$KONST are impure generated constants.

name. is a common block name appended with a dot.

name$ is a common block name corresponding to saved variables (user should avoid this type of symbol).

- Reference scalar arguments, except CHARACTER and those which are passed by address, by their variable names in a CAL block.
- The dummy arguments passed by address and dummy arrays in an embedded CAL block contain addresses, not values. In order to reference the value, load the address and then load the value using the address as a pointer.
- Do not use the following symbols as variable names in either the FORTRAN or assembler blocks.

ADC
LADC
ABSTOP
IMTOP
PURETOP

These names are reserved for use by the CAL assembler.

- Any FORTRAN variable that is to be used in the embedded CAL block must be specified in a $SETS or $USES directive. If neither $SETS or $USES is specified, the compiler assumes all FORTRAN variables will be both used and set by the embedded CAL blocks.

NOTE ▷ It is a good practice to declare all FORTRAN variables used in the CAL block with a $SETS or $USES directive for each embedded CAL block.

- If a variable is declared in FORTRAN, but never used in FORTRAN, its definition is omitted from generated CAL code. If this variable is referenced in CAL code, it must appear in at least one $SETS or $USES directive.
- If $USES is specified without a variable list, the embedded CAL cannot read the most current value for any FORTRAN variable in the program.
- If $USES is specified with a list of variables, the embedded CAL can read the most current value of only those variables in the list.
- If $SETS is specified without a list, the embedded CAL must not modify the value of any FORTRAN variable in the program.
- If $SETS is specified with a list of variables, the value of only those variables in the list may be modified by the CAL block.
- If a subprogram containing an embedded CAL block is to be expanded inline, any CAL label that is used only in the embedded CAL block must be specified by a $DISTINCT directive.
- Modified data areas inside CAL blocks must be in an impure segment to ensure proper segmentation when using the $SEG directive.
- All registers modified in the CAL block must be saved when entering the block and restored when exiting, unless a $REGS directive is used. $REGS should indicate the registers modified in the CAL block. See Chapter 3 for information on the use of this directive.
- Returning to the calling subprogram from an assembly block must not be done; if it is done, critical data may not be restored.
- GPR1 may not be used for any function other than its dedicated use as the RTL scratchpad pointer.
- If a subprogram containing an embedded assembly block is to be expanded inline, the lines of CAL code should be as short as possible.

- Declaring data areas in the embedded assembly blocks of a subprogram that is expanded inline more than once in a program should be avoided. Data areas which are only declared in embedded assembly blocks will not be shared by separate $INLINE expansions unless the user restructures the embedded assembly areas for this purpose.
- Branching from a CAL block into FORTRAN is permitted. All the FORTRAN labels to which the CAL block branches should appear on the $GOES compiler directive. If the user desires to have control fall through to the following FORTRAN code, a labeled statement must be placed immediately following the $FORT, and that label must appear on the $GOES of this CAL block:

```
      SUBROUTINE CAL
      DIMENSION SAVE1(2)
$ASSM
$GOES  10,20
         .
         .
         B      $P20
ZERO     L      3,SAVE1
         L      4,SAVE1+4
$FORT
10    CONTINUE
         .
         .
20    CONTINUE
         .
         .
```

  You must not violate any FORTRAN rules; e.g., branching into a DO loop.

- To compile your program with embedded Assembly code, you must follow certain procedures. See Chapter 6 for details on how to compile your program.

Get and Release Storage Assembly Routines

The I/O support routines in the R06-00 version of the RTL are rewritten in the C language. For this reason, use of this version may introduce incompatibility problems for applications which use the get and release storage SVCs (SVC 2, codes 2 and 3, respectively). To avoid this problem, the following assembly routines are to be used in place of the get and release storage SVCs. They are callable from the assembly level only.

NOTE Use of the assembly routines is not necessary to avoid incompatability problems if only the get storage SVC is used, and not the release storage SVC.

MALLOC Routine (Get Storage)

The MALLOC assembly routine is designed to get storage space. MALLOC requires one argument specifying the amount of space needed. Sample instructions needed to invoke MALLOC are as follows:

Example:

```
SIZE     EQU     200
GETBLK   DB      0,2
     DC      1
BLKSIZ   DAC     SIZE
       .
       .
       .
         LA      14,BLKSIZ
     BAL     15,_MALLOC
       .
       .
       .
```

GPR14 points to MALLOC's argument list; GPR15 stores the return address for the subprogram. The start address of the memory allocated is always returned in GPR13. If the allocation request failed, this value is 0. The start address must be recorded in GPR13 if the memory is restored (freed) or partially restored later. The argument list consists of an arbitrary location which contains the amount of space, in bytes, requested.

If the use of GPR13 through GPR15 presents a problem in your code, the following instructions provide an alternate interface:

```
SIS      1,12              Get space from the stack
STM      13,0(1)           Save the registers
LA       14,BLKSIZ         Set up arguments
BAL      15,_MALLOC        Get storage
LR       RX,13             Move address to desired register
ST       RX,SAVEADDR       Put space back on the stack
```

The "RX" is the register that previously received the start address from the get storage SVC.

MFREE Routine (Release Storage)

The MFREE assembly routine is is designed to release storage space. MFREE requires a single argument, specifying the start address of the storage space allocated by MALLOC. Sample instructions needed to invoke MFREE are as follows:

Example:

```
LA      14,SAVEADDR     Set up the argument list
BAL     15,_MFREE       Invoke it
```

GPR13 contains a 0 if no errors are detected; otherwise, GPR13 contains a 1.

The MFREE routine is not entirely equivalent to the release storage SVC. The SVC call returns storage starting from the top of dynamic storage, i.e., the storage that was last allocated. Invocation of the MFREE ' routine releases storage starting from the address given in the argument. This address must correspond to the address returned after a call to the MALLOC routine. Specifying an arbitrary address produces unpredictable results. The amount of storage released by MFREE is equal to the amount allocated by the corresponding MALLOC call.

If the use of GPR13 through GPR15 presents a problem in your code, use instructions similar to those provided in the discussion of the MALLOC routine to provide an alternate interface.

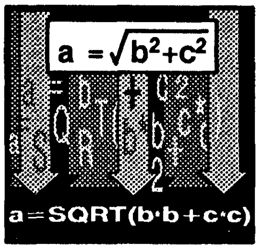
PFREE Routine (Release Partial Storage)

The PFREE assembly routine is designed to release a partial amount of storage space, which was previously allocated by the MALLOC routine. PFREE requires you to specify two arguments, the start address of the storage space and the amount of storage to release. Sample instructions needed to invoke PFREE are as follows:

Example:

```
SAVEADDR   DSF      1
AMOUNT     DSF      1
       .
       .
       .
      LA       14,SAVEADDR      Set up the argument list
      BAL      15,_PFREE        Invoke it
```

Storage is released starting from the top of the storage space. You cannot partially release more storage than was previously allocated. The argument list is simply two contiguous fullwords, the first being the address and the second the amount to release.



6

Building a Command File to Compile Your Program

In this chapter

We discuss compiling your program using the FORTRAN VII compilers. The available start directives which were briefly presented in Chapter 3 are described.

Topics include:

- Basic compilation process
- Using the F7O and F7Z compilers
- Allocating the input/output (I/O) files
- Using the start directives
- Compiling source with embedded assembly code

The Basic Compilation Process

The minimum functions that must be performed by a command procedure to compile a program are:

1. Load the compiler.
2. Allocate and assign the I/O files required for compilation.
3. Start compilation.
4. Check the end of task code.
5. If end of task code is equal to 0, begin the LINK process.
6. If end of task code is greater than or equal to 1 write an error message and terminate the program development procedure.

The system command procedure COMPILE.CSS performs these functions. If your program does not have any special requirements, such as embedded Common Assembly Language (CAL) code or larger work space, this command substitution system (CSS) is sufficient to compile it. Otherwise, you may have to tailor-fit your command procedure or provide the necessary compiler directives as discussed in the succeeding section.

Using the F7O and F7Z Compilers

The following code sequence performs the basic functions of a compilation procedure using the F7O and F7Z compilers.

Example:

```
 1   LOAD F7O,100
 2   ASSIGN 1,@1.FTN
 3   XALLOCATE @1.OBJ,IN,126/2
 4   ASSIGN 2,@1.OBJ
 5   XALLOCATE @1.LST,IN,132/2
 6   ASSIGN 3,@1.LST
 7   TEMP 8,CO,4000
 8   TEMP 4,IN,80/5
 9   ASSIGN 7,ERRORFIL            * use appropriate descriptor
10   START,@2
11   $IFG     3
12   $WRITE   COMPILATION ERRORS
13   $CLEAR
14   $ENDC
```

Call this command procedure COMPILE1.CSS. The succeeding sections refer to specific lines of this CSS file.

NOTE All references to compilation under F7O in this section apply equally to the F7Z compiler.

Loading the Compiler

To load the F7O and F7Z compilers, use the operating system LOAD command specifying a memory increment size of at least 100kB as in line 1 of COMPILE1.CSS.

LOAD F7O,100

The memory increment size of 100 represents the amount of workspace used by the compiler during source compilation. A minimum of 6kB of storage is required. (A minimum of 4kB is required if the number of continuation lines allowed for each FORTRAN statement is zero.) However, most programs require at least 100kB. If you specify an increment of less than 6kB (4kB if CONT=0), compilation terminates after the following message is sent to the list device.

```
NOT ENOUGH REAL MEMORY TO COMPLETE COMPILATION
```

If you specify an increment size greater than 6kB, but not large enough to contain all of the compiler generated symbol tables and logical unit 8 (lu8) is not assigned, compilation terminates after the following message is sent to the list device.

```
WORK FILE ERROR - LU NOT ASSIGNED
```

To avoid the occurrence of this error, the program development procedure must allocate a temporary contiguous work file and assign it to lu8. The compiler then sends the entire symbol table that did not fit within the workspace area to the work file after logging this message to the list device.

```
INTERNAL TABLES PAGING TO DISK
```

If, while paging to the disk, the compiler encounters end of medium (EOM) for the work file, it terminates compilation after sending this message to the list device.

```
END-OF-MEDIUM
ALLOCATE A BIGGER CONTIG FILE FOR PAGING-(LU8)
```

If this occurs, allocate a larger contiguous file for lu8.

If the internal table exceeds the maximum size, the compilation terminates with the following message to the list device.

```
COMPILER TABLE LIMIT EXCEEDED
```

NOTE No recovery is provided for a table limit error. You must reduce the total number of lines in the program unit being compiled.

By reducing the total number of lines of code, the amount of memory space required by the internal table is reduced.

If the internal graphing tables of the optimizer have exceeded their maximum size, the compiler sends this message to the list device.

```
OPTIMIZER TABLE SATURATION:  LIMITED OPTIMIZATION MAY RESULT
```

This message suggests that in recovering from table saturation, the compiler must forego optimizations it might have performed had the table size not been exceeded. Therefore, optimization of a program is impaired. To eliminate impairment due to table saturation, reduce the total number of lines in the program unit being compiled.

Allocating and Assigning I/O Files

The COMPILE procedure for the optimizing compilers must assign:

- lu1 to the input device or file containing the source program,
- lu2 to the file containing the compiler output object code,
- lu3 to the device or file to which the compiler outputs all listings, warnings, and diagnostic messages,
- lu8 to the temporary work file described in the previous section,
- lu4 to a temporary work file, and
- lu7 to the error file that contains the text for error messages generated by the compiler.

Thus, COMPILE1.CSS contains lines 2 through 9 as follows:

```
ASSIGN 1,@1.FTN
XALLOCATE @1.OBJ,IN,126/2
ASSIGN 2,@1.OBJ
XALLOCATE @1.LST,IN,132/2
ASSIGN 3,@1.LST
TEMP 8,CO,4000
TEMP 4,IN,80/5
ASSIGN 7,ERRORFIL
```

The compiler automatically assigns lu0 to a temporary work file if you have not assigned it.

Take note of the following when making the lu assignments:

- If you generate assembly code rather than object code, you must assign lu6 to a file that receives the assembly code for later processing by the CAL assembler. See "Writing a Program Development Procedure for FORTRAN with Embedded Assembly Language" section for details.

- The COMPILE procedure must assign logical units 1, 2, 3, 4, 7, and 8. It is often convenient to use the XALLOCATE command to avoid having to manually delete the temporary files each time the program is recompiled.
- When allocating the list file, you must specify a record length of 132 bytes.
- To decrease compilation time, do not allocate the source input, CAL and object files to the same disk that the scratch and list files are allocated.
- You must assign a scratch file to lu4. This file must have an 80-byte record length.
- When allocating the temporary work file, follow these three guidelines:
  - — Allocate a contiguous file.
  - — Locate the work file on the fastest disk with the greatest number of free contiguous sectors.
  - — Use the following formula to determine the number of sectors that should be allocated to the work file.

$$\textit{number of sectors} = \frac{4 \text{ x } \textit{number of program lines}}{6}$$

NOTE Before compilation, make certain that the error file is present on the system volume.

The F7O and F7Z lu assignments are summarized in Table 6-1.

lu	Device or File	File Extension	Record Length
0	Temporary work file	.TMP	80
1	Source input	.FTN	None
2	Object output	.OBJ	126
3	Listing output	.LST	132
4	Compiler scratch file	.TMP	80
5	Reserved - (MUST NOT BE ASSIGNED)		
6	CAL output	.CAL	80
7	F70 or F7Z error message file	.ERR	80
8	Work file	.TMP	256*

* Must be a contiguous file

Table 6-1. Logical Unit Assignments

Using the Compiler Start Directives

Directives to the optimizing compilers can be listed as parameters to the START command. These parameters can be specified simultaneously in the COMPILE procedure and as one of the parameters to the command that calls the procedure. In the following example, the start directives are listed in parameter position @2. If you use EXEC to call the COMPILE procedure, enter the following:

```
EXEC FILENAME,COMP TEST
```

COMP and TEST are directives to the compiler because they are located at positional parameter @2. FILENAME, which is located at positional parameter @1, is the name of the source program file.

The format for the START command used with the F7O and F7Z compilers follows.

NOTE If conflicting start options are specified (i.e., XREF and NXREF are both specified) the option which was specified last (e.g., rightmost option on the start line) overrides. No warning is given when this occurs.

$$\text{START,}\ \left[\begin{Bmatrix}\underline{\text{A}}\text{LST}\\ \underline{\text{NA}}\text{LST}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{BAB}}\text{ORT}\\ \underline{\text{NBAB}}\text{ORT}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{BAS}}\text{E}\\ \underline{\text{NBAS}}\text{E}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{NB}}\text{ATCH}\\ \underline{\text{B}}\text{ATCH}\end{Bmatrix}\right]$$

$$\left[\begin{Bmatrix}\text{CAL}\\ \text{NCAL}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{C}}\text{OMP}\\ \underline{\text{NC}}\text{OMP}\end{Bmatrix}\right]\left[\left\{\underline{\text{CONT}}\text{IN}=\begin{Bmatrix}n\\ 19\end{Bmatrix}\right.\right]$$

$$\left[\begin{Bmatrix}\underline{\text{NEL}}\text{IST}\\ \underline{\text{EL}}\text{IST}\end{Bmatrix}\right]\begin{bmatrix}\text{F66DO}\\ \text{NF66DO}\end{bmatrix}\left[\begin{Bmatrix}\underline{\text{H}}\text{OLL}\\ \underline{\text{NH}}\text{OLL}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{INF}}\text{ORM}\\ \underline{\text{NINF}}\text{ORM}\end{Bmatrix}\right]$$

$$\left[\text{LCNT}=\begin{Bmatrix}n\\ 60\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{NL}}\text{IST}\\ \underline{\text{L}}\text{IST}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{LTOR}}\text{BIT}\\ \underline{\text{RTOL}}\text{BIT}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{NO}}\text{PTIMIZE}\\ \underline{\text{O}}\text{PTIMIZE}\end{Bmatrix}\right]$$

$$\left[\begin{Bmatrix}\text{NSEG}\\ \text{SEG}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{S}}\text{YNTAX}\\ \underline{\text{NS}}\text{YNTAX}\end{Bmatrix}\right]\left[\underline{\text{T}}\text{ARGET}=n\right]\left[\begin{Bmatrix}\underline{\text{TE}}\text{ST}\\ \underline{\text{NTE}}\text{ST}\end{Bmatrix}\right]$$

$$\left[\text{TCOM}\right]\left[\begin{Bmatrix}\underline{\text{TR}}\text{ACE}\\ \underline{\text{NTR}}\text{ACE}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{NW}}\text{ARN}\\ \underline{\text{W}}\text{ARN}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{X}}\text{REF}\\ \underline{\text{NX}}\text{REF}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{INL}}\text{INE [= fd]}\\ \underline{\text{NINL}}\text{INE}\end{Bmatrix}\right]$$

$$\left[\text{INLIB}=\begin{Bmatrix}fd\\ (fd_1\ [fd_2 ... fd_n])\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{UN}}\text{NORMALIZE}\\ \underline{\text{NUN}}\text{NORMALIZE}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{AP}}\text{U}\\ \underline{\text{NAP}}\text{U}\end{Bmatrix}\right]$$

$$\left[\underline{\text{DP}}\right]\ \left[\text{INT2}\right]\left[\begin{Bmatrix}\underline{\text{IBYT}}\text{E}\\ \underline{\text{LBYT}}\text{E}\end{Bmatrix}\right]\left[\begin{Bmatrix}\text{OBJ}\\ \text{NOBJ}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{TAB}}\text{LES}\\ \underline{\text{NTAB}}\text{LES}\end{Bmatrix}\right]$$

$$\left[\begin{Bmatrix}\underline{\text{XF}}\text{ORT}\\ \underline{\text{NXF}}\text{ORT}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{REENT}}\text{RANT}\\ \underline{\text{NREENT}}\text{RANT}\end{Bmatrix}\right]\left[\begin{Bmatrix}\underline{\text{NTRANS}}\text{CENDENTAL}\\ \underline{\text{TRANS}}\text{CENDENTAL}\end{Bmatrix}\right]$$

$$\left[\underline{\text{PASSBYAD}}\text{DRESS}\right]$$

NOTE The INLINE, INLIB, and ELIST start directives are available with the F7Z compiler only.

The NOBJ, TABLES, and XFORT start directives are for E/SP use only.

ALST causes the compilers to produce a CAL listing of the source program after each program unit is successfully compiled.

NALST prevents the compiler from generating a CAL listing of the source program.

APU is provided for multiprocessor operations and causes F7 compilers to look for FORTRAN features in the input source that are known to generate supervisor call (SVC) instructions. The compiler will then generate an information message to that effect. It also causes the compiler to generate a DCMD in the object file so that a similar message is output at link time to the LOG file.

NAPU prevents the compiler from searching for FORTRAN features which generate SVC instructions.

BABORT terminates a batch compilation when compilation of any one of the program units ends in error.

NBABORT prevents termination of batch compilation from continuing when compilation of any one of the program units ends in error.

BASE causes the compiler to consider base addressing of all variables and named common blocks for register allocation.

NBASE prevents the compiler from considering base addressing of all variables and named common blocks for register allocation.

BATCH causes all program units within a source file to be compiled.

NBATCH causes the compiler to stop compiling the source file when a FORTRAN END statement is encountered.

CAL causes the compiler to generate assembly language output of the source program. This output can then be assembled by CAL/32 into object code.

NCAL causes the compiler to compile the FORTRAN source program into object code.

COMP compiles all FORTRAN statements, including those having an X in column 1.

NCOMP prevents compilation of all FORTRAN statements having an X in column 1 and replaces X with # when the source code is sent to the list device.

CONTIN=*n*
specifies the maximum number (*n*) of lines the compiler should accept for each FORTRAN statement. The variable *n* represents a decimal number from 0 to 100. If this parameter is not specified, the default is 19.

DP
causes all real and complex items whose lengths were not specified explicitly in a specification statement to be treated as double precision items. Length specification of *4 and *8 (for REAL) and *8 and *16 (for COMPLEX) are still available if explicitly used in specification statements. Further, all REAL and COMPLEX constants will be treated as REAL*8 and COMPLEX*16 constants, respectively, when the DP option is specified. If this option is not specified, the default is the type associated with FORTRAN identifiers and constants.

ELIST
outputs an extended listing. When LIST is in effect, ELIST is the default. This directive is available with the F7Z compiler only.

NELIST
prevents an extended listing from being sent to the list device. When NLIST is in effect, NELIST is the default unless ELIST is specified. This directive is available with the F7Z compiler only.

F66DO
causes all DO loops to be executed at least once. This option supports compatibility with ANSI '66 FORTRAN. If you do not specify this option, ANSI '77 FORTRAN is the default. Specifying this option produces code that is not compatible with ANSI '77 FORTRAN.

NF66DO
supports compatibility with ANSI '77 FORTRAN. This is the default.

HOLL
causes the compiler to interpret all quoted strings used as arguments to subprograms as Hollerith constants.

NHOLL
causes the compiler to interpret all quoted strings used as arguments to subprograms as character constants.

IBYTE
causes the BYTE statement to be treated as an INTEGER*1 statement. All entities appearing in the "var list" of the BYTE statement are treated as INTEGER*1 entities. This is the default setting for the BYTE statement.

LBYTE
causes the BYTE statement to be treated as LOGICAL*1 statement. All items appearing in the "var list" of the BYTE statement are treated as LOGICAL*1 entities.

INFORM
causes the compiler to send information on code optimization to the list device. If this directive is not specified when the LIST directive is in effect, INFORM is the default.

NINFORM
prevents information on code optimization from being sent to the list device. If this directive is not specified when NLIST is in effect, NINFORM is the default.

INLIB
specifies the file descriptor(s) of the source library files that the compiler is to search for the source of the subprograms to be expanded in-line. This directive is available with the F7Z compiler only.

INLINE=*fd*
causes the compiler to expand subprograms designated by the in-line directives contained in the file designated by the *fd*. If INLINE is specified without *fd*, all calls to subprograms are expanded and batch compilation is suppressed, except for the compilation of BLOCK DATA modules in the batch. This directive is available with the F7Z compiler only.

The file designated by *fd* consists only of instream in-line directives. To identify the program unit to which the directives apply, the program header ***name* is used, where *name* is the subprogram name. If the directive applies to the main program unit, ** is used. For more information on in-line directives, see the section entitled "Notes on Using the INLINE Start Directive."

Example:

The file INDIR.SAM contains the following:

```
**
$INLINE SUBA,*     ; INLINE DIRECTIVES FOR
$INLINE SUBB,*,10  ; MAIN PROGRAM UNIT
**SUBA             ;
$INLIB SUB1.LIB    ; INLINE DIRECTIVES FOR
$INLINE A,-        ; SUBPROGRAM SUBA
**SUBC
$INLINE SUBB,-     ; INLINE DIRECTIVE FOR SUBC
(END OF FILE)
```

To utilize these instream in-line directives, the START command must be invoked with the following option:

```
START,INLINE=INDIR.SAM
```

NOTE When INLINE is specified as a start directive without *fd*, F7Z will not compile any subprograms (except BLOCK DATA) following the first program unit that is compiled. INLINE does not expand calls made to an entry name in a subprogram.

NINLINE

causes the compiler to ignore any INLINE directives that may be encountered in the source code. No subprogram will be expanded in-line. This directive is available with the F7Z compiler only.

INT2

causes all integer and logical items whose lengths were not specified explicitly in a specification statement to be treated as INTEGER*2 and LOGICAL*2 items, respectively. Length specification of *1, *2, and *4 are still available if explicitly used in specification statements. 4-byte entities must be specified where INTEGER*4 or LOGICAL*4 entities are required. If this option is specified and a constant used is larger than the integer value 32767 or smaller than -32768, an INTEGER*4 constant is created for that element and a warning message is issued.

If you do not specify this option, the default type is the usual type associated with FORTRAN identifiers and constants. Note that this feature conflicts with the "storage unit" standards of FORTRAN 77.

LCNT=*n*

specifies the number of lines (*n*) per page the compiler outputs to the list device. If this parameter is not specified, the compiler automatically outputs 60 lines per page.

LIST

causes the compiler to send a complete listing of the source code and all compiler error messages and warnings to the designated list device. LIST is the default. LIST causes INFORM and ELIST (for F7Z only) to be in effect if they were not specified.

NLIST causes the compiler to send only warnings, error messages, and statements that have errors to the designated list device. NLIST causes NINFORM and NELIST (for F7Z only) to be in effect if they were not specified.

LTORBIT causes bit positions in a word to be counted from left to right. In a 4-byte word, the left most bit position is marked as 0 and the right most bit position is marked as 31. If this option is not specified, the bit positions are counted from left to right. This option affects bit manipulation routines.

RTOLBIT causes bit positions in a word to be counted from right to left. In a 4-byte word, the right most bit position is marked as 0 and the left most bit is marked as 31. If this option is not specified, the default is LTORBIT. This option affects bit manipulation routines. See the section in Chapter 3, "Miscellaneous Instream Compiler Directives," on the $RTOLBIT directive for more information.

OBJ produces object code. This is the default.

NOBJ suppresses the generation of object code. This is used in conjunction with the TABLES option to instruct E/SP to control compiler output at different phases of the parallel program development cycle.

OPTIMIZE activates all optimization routines available on the F7O and F7Z compilers.

NOPTIMIZE turns off the following optimizations:

— Global register allocation

— Extended strength reduction

— Constant propagation

— Invariant code motion

— Test replacement

— Scalar propagation

— Folding

— Common subexpression elimination

— Dead code elimination

When this directive is used with any of the NOBJ, TABLES, and XFORT directives, flow and data analyses, which are normally suppressed by NOPTIMIZE, are still performed. These operations generate data needed by E/SP to construct a dependence graph of the program. See the appropriate manual in the E/SP documentation set for details on these directives.

PASSBYADDRESS causes the module to treat all of its noncharacter scalar dummy arguments as if they were passed by reference. If $NPASSBYADDRESS is in effect, noncharacter scalar dummy arguments which are not enclosed in slashes in the FUNCTION or SUBROUTINE statement are treated as if they were passed by value-result.

This option has no effect on arguments passed to other subprograms; the choice of passing by reference vs. passing by value-result is determined solely by the coding of the FUNCTION/SUBROUTINE statement. that receives the arguments. If this option appears after the first statement of a module, you get a warning message and the compiler ignores the directive. The scope of PASSBYADDRESS is limited to the module in which it appears.

REENTRANT generates reentrant code. This option allows you to develop reentrant (sharable) libraries. If you do not specify this option, the default is NREENTRANT, i.e., the code generated is not reentrant. Use of REENTRANT precludes the use of any construct resulting in the initialization or modification of static storage, i.e., DATA, GLOBAL, COMMON, or SAVE. It may also result in substantially less efficient code.

SEG causes the compiler to generate segmented object code. All local data is placed in the impure task segment; executable code is placed in the pure task segment. See the *OS/32 Application Level Programmer Reference Manual* for more information on task segments.

NSEG prevents the compiler from generating segmented object code. All code is placed in the impure task segment.

NOTE Code placed in an impure segment cannot be shared.

SYNTAX causes the compiler to check the source code for syntax errors without producing optimized object code.

NSYNTAX — causes the compiler to check for syntax errors and generate optimized object code during compilation of the source code.

TABLES — generate dependence tables and transcribed source code before downloading a program to E/SP for restructuring.

NTABLES — turns off the TABLES option. This is the default.

TARGET=*n* — causes the compiler to generate machine code specifically optimized for the processor as denoted by *n*. If *n*=0, the compiler will output machine code capable of being executed on any one of the Concurrent 32-bit processors. If *n*=3200, the machine code is targeted for any of the Series 3200 Processors. If TARGET is not specified, the machine code will be targeted for the processor that ran the compiler when the program was compiled. If *n*=3205 or *n*=328*x*, the compiler, by default, generates unnormalized floating point load instructions. In this case, however, the code generated for the Model 3205 or 3280 systems executes only on these systems including the Model 3203, which also supports unnormalized floating point. When *n*=328*x*, the optimizing compiler generates in-line machine code instructions for the following math functions: sine, cosine, square root, log, log10, and atan. The object code generated does not run on any other 32-bit processors.

NOTE When TARGET=3200 or higher is specified, the resultant object code will not execute on the model 8/32 processor. If TARGET=3280 is specified, the resultant object code executes only on a 3280 system.

To target code to the MicroThree, MicroFive, or 3280E, use TARGET=3283, TARGET=3285, or TARGET=3288, respectively.

TCOM — informs the compiler that all named common blocks, common entities, and global entities are candidates for task common and prevents the compiler from allocating registers for these entities or eliminating code that references them. Unrestricted use of TCOM will impair optimization severely.

TEST
causes the compiler to generate code that checks the bounds of array subscripts and substrings; outputs an error message at run-time to the device or file assigned to a lu6 at any point in the program when the value of an array subscript or substring becomes out of bounds. It is not intended for checking boundary violations of arrays passed as arguments to subprograms or used as a buffer in ENCODE/DECODE statements.

NTEST
prevents the compiler from generating code that will perform the TEST function.

TRACE
causes the compiler to generate code that outputs a message at run-time to the device or file assigned to lu6 when:

- the value of any program variable is changed by a logical or arithmetic assignment statement, or
- a labeled statement is executed.

NTRACE
prevents the compiler from generating code that will perform the TRACE function.

TRANSCENDENTAL
causes transcendental functions to be generated as single microinstructions in the machine code for the 328*x* processor. Specifying NTRANSCENDENTAL on a 328*x* processor generates calls to the RTL version of each transcendental function. This option has no effect on any other Series 3200 Processor. TRANSCENDENTAL is the default for a 328*x* processor and NTRANSCENDENTAL for other Series 3200 processors.

NTRANSCENDENTAL
prevents transcendental functions from being generated as single microinstructions in the machine code for the 328*x* processor.

UNNORMALIZE
directs the compiler to generate unnormalized floating point load instructions. This option overrides the TARGET option for the generation of unnormalized or normalized instructions. If the code is being targeted for the Model 3203, 3205, or 3280 processors, UNNORMALIZE is the default.

If the compiler generated any unnormalized floating point instructions, a DCMD LINK command will be embedded in the generated object code. The text of the DCMD is:

```
****MODULE xxxx CONTAINS NON-NORMALIZING LOADS
```

where *xxxx* is the name of the program where unnormalized floating point load instructions were generated. This comment on the DCMD will be displayed on the LOG device by OS/32 Link when the object code is used to build a task.

NUNNORMALIZE inhibits the generation of unnormalized floating point load instructions regardless of the TARGET option.

NOTE Although unnormalized floating point instructions are currently implemented only on the Model 3205 and 3280 processors, the UNNORMALIZE/NUNNORMALIZE option is provided for future additional processors.

WARN causes the compiler to send warning messages to the list device.

When this directive is used with any of NOBJ, TABLES, and/or XFORT, it sends E/SP warning messages in addition to the regular messages. These warning messages flag language constructs in the program that produce a complicated graph or inhibit parallelism. See the appropriate manual in the E/SP documentation set for details on these directives.

NOTE Do not use NOBJ, TABLES, or XFORT with this directive if the E/SP warning messages are not desired.

NWARN prevents the compiler from sending all warning messages to the list device.

XFORT informs the compiler that the code may contain non-FORTRAN structures which require special parsing for E/SP.

NXFORT informs the compiler that there are no non-FORTRAN constructs. This is the default.

XREF causes the compiler to generate a cross-reference listing of variables and labels in the source code. This listing is output to the designated list device.

NXREF prevents the compiler from generating a cross-reference listing of the source program.

Using the In-line Start Directives

In-line expansion is not a default action of F7Z. You must explicitly request in-line expansion through the INLINE/$INLINE directive.

When you specify the INLINE directive (without *fd*) in the START command, special search methods are used by the compiler to find the source of the subprogram to be expanded in-line. The order of search is as follows:

1. The source of a subprogram is searched for in the file specified through the INLIB/$INLIB directive.
2. If the source of the subprogram is not found in the file, or if the $INLIB directive is not specified, the source is searched for in the file that contains the call to that subprogram.
3. If the source is not found in that file, the source is searched for in the input file.

If the source of the subprogram is not found in any of these files, the compiler outputs a warning message and excludes that subprogram from in-line expansion. $NBATCH is enforced at the completion of in-line expansion.

The file specified by the *fd* parameter of the $INLINE directive may consist of in-line directives and the names of the subprograms to which those directives apply. (The file can only consist of in-line directives.) To identify a subprogram to which the directives apply, the program header ***name* is used, where *name* is the name of the subprogram. For the main program, this should be just **, with no *name* (even if a PROGRAM statement is used to name the program).

The in-line directives that apply to this subprogram are those which follow the header until the next program or end of file is encountered.

If the first record on the file *fd* is not a program header, the in-line directives in that file before the first program header, if any, are applied to all the subprograms compiled. In addition to these directives, the directives following the program header of a subprogram up to the next header, if any, are also applicable to that subprogram. A $INSKIP directive, if any, must appear immediately after the header in the file *fd*. In this case, the subprogram with this name is not compiled separately. If the ALL option is specified by the $INSKIP directive, batch compilation is terminated after compiling all the subsequent BLOCK DATA subprograms in the batch input.

Testing End of Task Code

Compilation by the optimizing compilers can terminate under any one of the following seven conditions.

- All source code was compiled into object code with no syntax errors.
- All source code was compiled into assembly language output with no syntax errors.
- All source code was compiled and found to contain syntax errors.
- The directives specified in the START command are illegal.
- The memory size specified by the LOAD command is less than 6kB (4kB if CONT=0).
- An internal table exceeds the maximum table size.
- The compiler encounters either an EOM or an I/O error on the temporary work file.

These conditions and their end of task codes are listed in Table 6-2.

End Of Task Code	F7O and F7Z Termination Status
0	No compilation errors, object code produced.*
1	No compilation errors, CAL produced.
4	Compilation errors.
6	Illegal start directives.
8	Real memory insufficient.
9	Compiler table limit exceeded.
10	End of medium encountered on work file.

* If $SYNTAX is in effect, no object code is produced and, therefore, the program cannot be linked.

Table 6-2. End of Task Codes

The program development procedure must check for end of task codes 0 or 1 before proceeding with the LINK procedure. To do this, the procedure uses the CSS conditional commands. For further information on the CSS commands, see the *OS/32 Multi-Terminal Monitor (MTM) Reference Manual.*

Recall lines 11-13 of COMPILE1.CSS presented earlier,

```
$IFG      3
$WRITE   COMPILATION ERRORS
$CLEAR
$ENDC
```

These lines check for an end of task of 3 or greater. If it is equal to 0, the operating system skips over the next three commands and ends the compilation process. You can now LINK your program as described in Chapter 7. If the end of task code is 1, the CAL output produced by the compiler should be assembled with the CAL/32 assembler. You can expect this result only if you specify the CAL/$CAL directive. See the next section for the CSS procedure that performs the ASSEMBLY process. An end of task code greater than 3 causes the operating system to send the message COMPILATION ERRORS to the terminal. $CLEAR terminates the COMPILE procedure.

The program development procedure can also test for other end of task codes.

Example:

```
                     .
                     .
                     .
$IFG 3
  $IFE 4;$WRITE COMPILATION ERRORS IN SOURCE;$ENDC
  $IFE 6;$WRITE CHECK FOR MISSPELLED START DIRECTIVES;$ENDC
  $IFE 8;$WRITE MEMORY OVERFLOW: INCREASE LOAD SIZE;$ENDC
$ENDC
```

For more information on how to use the CSS conditional commands, see the *OS/32 Multi-Terminal Monitor (MTM) Reference Manual.*

Program Development Procedures With Embedded CAL

The program development procedure COMPILE1.CSS presented earlier in this chapter does not take into account programs with embedded assembly codes and when you specify the CAL/$CAL directive. For convenience, the procedure is presented again as follows:

```
LOAD F7O,100
ASSIGN 1,@1.FTN
XALLOCATE @1.OBJ,IN,126/2
ASSIGN 2,@1.OBJ
XALLOCATE @1.LST,IN,132/2
ASSIGN 3,@1.LST
TEMP 8,CO,4000
TEMP 4,IN,80/5
ASSIGN 7,ERRORFIL
START,@2
$IFG 3
$WRITE COMPILATION ERRORS
$CLEAR
$ENDC
```

When you specify the CAL/$CAL directive, the above procedure produces assembly language code rather than object code. Since this procedure does not produce the object code, the allocation of the object code file (@1.OBJ) and its assignment to lu2 are not necessary. However, the assembly code produced must be allocated to a file and assigned to lu6. Otherwise, an assignment error occurs and compilation aborts. The following two lines must be added to the program development procedure above:

```
    XALLOCATE @1.CAL,IN,80/5
    ASSIGN 6,@1.CAL
```

This file containing the assembly code, must be assembled by the CAL assembler to obtain the object code for linking. A typical ASSEMBLY procedure is shown below:

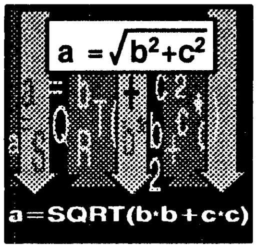
```
    LOAD CAL32,50
    XALLOCATE @1.LST,IN,132/4
    ASSIGN 3,@1.LST,EWO
    XALLOCATE @1.OBJ,IN,126/2
    ASSIGN 2,@1.OBJ,EWO
    TEMPORARY 4,IN,80/2
    ASSIGN 1,@1.CAL,SRO
    START,SQUEZ=99,NLIST,@2
    $IFNE 0
    $WRITE **CAL ERRORS**
    $CLEAR
    $ENDC
```

Where:

@1 = SOURCE FILENAME
@2 = CAL START OPTIONS

lu2 and lu3 must be assigned with 'EWO' privileges. lu7 may be assigned to a copy file.

NOTE If you invoked the SYNTAX compiler option, either as a start option or as an inline directive, and the compiler ends with EOT 1, CAL may pause with an I/O error 8800, unless you enter BATCH mode as a start parameter to CAL.



7

Building a Command File to Link a FORTRAN Program

In this chapter

We discuss how the Link process converts the object module created during the COMPILE phase into an executable task image. Link constructs a loadable task image from the object modules and object libraries that you specify.

Topics include:

- Building the Link command file
- Linking trap handling programs
- Linking shared data/segments
- Overlaying large programs
- Loading and executing Link

Introducing the Basic Link Development Procedure

The minimum functions performed by the LINK development procedure are as follows:

- allocate the input/output (I/O) files required by Link,
- build the Link .CMD file,
- load the Link software,
- start Link,
- check the end of task code, and
- write an error message and terminate the program development procedure, if end of task code is greater than 0.

The following command sequence performs the basic operating system functions for linking a FORTRAN program.

Example:

```
XDELETE   @1.TSK
XALLOCATE @1.MAP,IN,132/2
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
* The following sequence builds the link .CMD file    *
* using the CSS $BUILD ... $ENDB commands. This      *
* file is then passed to the Link task via the        *
* COMMAND= parameter of the OS/32 START command.      *
$BUILD LINK.CMD
 ESTABLISH TASK
 MAP @1.MAP,ALPHABETIC,ADDRESS,XREF
 OPTION DFLOAT,FLOAT,WORK=(X2800,X2800),SYSSPACE=XFFFF
 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * If you are using Link R00-01 or lower, change the   *
 * OPTION command to:                                  *
 *    OPTION DFLOAT,FLOAT,WORK=(C00,C00),SYSSPACE=FFFF *
 INCLUDE @1.OBJ
 LIBRARY F7RTL.OBJ/S
 BUILD @1.TSK
 END
$ENDB
* * * * * * * * * * * * * * * * * * * * * * * * * * *
* The following is the load and start sequence to   *
* invoke the LINK task                              *
LOAD LINK/S,20
START,COMMAND=LINK.CMD,LOG=CON:
* * * * * * * * * * * * * * * * * * * * * * * * * * *
* The following sequence tests for an EOT code of   *
* other than zero (0)                               *
$IFNE 0
$WRITE   LINK ERRORS
$CLEAR
$ENDC
$EXIT
```

You may save this command sequence to a command substitution system (CSS) file such as LINK1.CSS (avoid naming CSS files after existing system files; LINK.CSS is a system CSS file).

NOTE If you are using OS/32 Link version R00-01 or lower, take note of the following differences:

- Use ESTABLISH SHARED instead of ESTABLISH IMAGE when linking a partial image.
- Use the command SHARED instead of the Link RESOLVE command. The partial image file created gets an extension .SEG instead of .IMG.

Example:

SHARED *fd*.SEG   instead of   RESOLVE *fd*.IMG

- Any values to reflect a work area or system space need not be preceded by an X.

Example:

```
WORK=(C00,C00),SYSSPACE=FFFF
```

instead of

```
WORK=(XC00,XC00),SYSSPACE=XFFFF
```

Allocating Link I/O Files

Link requires the following I/O files.

- Object files containing the compiled source code.
- Task image file to which Link outputs the task image.
- Map file to which Link sends a listing of all program names and their addresses.
- Log file to which Link logs all commands given to it and any Link-generated diagnostic messages.
- .CMD file containing commands to Link. Without this file, you have to use Link interactively.

These I/O files are assigned and/or accessed by Link via Link commands that are contained in the .CMD file. You can create the Link .CMD file within the program development procedure LINK1.CSS (as shown in the previous section) or create it as a separate file using any text editor. This .CMD file must then be specified in the START command. What you can specify within the .CMD file is discussed in the next section.

The BUILD command for Link automatically allocates a file for the task image using the source filename followed by the extension .TSK.

The map and log files must be allocated by the program development procedure. In the previous example, a file is allocated for the Link map. Because all messages are being logged to the console, it was not necessary to allocate a log file.

Table 7-1 lists the logical unit (lu) assignments that are automatically made by the Link commands.

Link Command	Logical Units Assigned	I/O File
INCLUDE	1	OBJECT
BUILD	2	TASK IMAGE
MAP	3	LINK MAP
START		
,COMMAND=	5	LINK COMMAND
,LOG=	6	LOG

Table 7-1. Logical Units Assigned By Link

The Link commands used in this chapter are fully explained in the *OS/32 Link Reference Manual.* This chapter explains these commands adequately for you to follow the discussions. However, not all the possible uses and options of these commands are covered.

Building a Basic Link .CMD File

The LINK1.CSS procedure presented earlier in this chapter uses the CSS $BUILD ... $ENDB commands to create the Link .CMD file. This file contains the minimum commands required to link a FORTRAN object file. These commands are:

```
ESTABLISH
MAP
OPTION
INCLUDE
LIBRARY
BUILD
END
```

ESTABLISH depending on the option passed to it, this command tells Link to convert the object code into a loadable task image (TASK), partial image (IMAGE), or operating system (OS).

MAP allows Link to send a Link map to the specified file or device.

OPTION specifies the task options that must be activated during execution of the task image. The options to be specified are:

```
FLOAT
DFLOAT
WORK=
SYSSPACE=
```

If the program uses single precision arithmetic, FLOAT must be specified. The use of double precision arithmetic requires the DFLOAT option.

To determine the amount of additional workspace that should be given to a task by the WORK= option, use the following formula.

WORK = 8kB + I/O buffer space (in bytes) + (68B * # of LOGICAL UNITS) + 114B + 268B + *temporary space*

8kB is the workspace required for the run-time library (RTL) stack. This size can be changed using the OS/32 PATCH Utility. For more information, refer to the discussion of the REENTRANT directive, in Chapter 3, in the section entitled "Miscellaneous Instream Compiler Directives."

I/O buffer space — is the largest single record length (in bytes) needed for any one I/O multiplied by the maximum number of logical units assigned at any given time.

68B — number of logical units is used by the FORTRAN static communication area for maintaining information about each lu.

114B — is a constant that the FORTRAN static communications area uses for other data.

268B — is the maximum workspace required to support start options using the GETOPTS RTL routine.

temporary space — dynamically allocated space used by the RTL. The amount allocated depends upon the application being performed (usually between 2-5KB).

For most applications, a SYSSPACE option of X'FFFF' is sufficient. For tasks containing trap handling routines, see the section entitled "Linking Trap Handling Programs".

INCLUDE — specifies an object file that is to be included when the task image is built. This is the object file created by the COMPILE process.

LIBRARY — specifies the RTLs containing the RTL routines called by the FORTRAN program. These RTL routines, which Link incorporates into the task image module, can be user-written or those provided by the FORTRAN VII RTL.

Twelve versions of the RTL can be built from the F7RTLLIB.CSS, a command file which allows customization of the RTL based on the arguments specified. The format to build the RTL is as follows:

$$\text{F7RTLLIB} \left[\text{A} \left[, \left[\begin{Bmatrix} \text{X} \\ \text{W} \end{Bmatrix} \right] \left[, [\text{C}] \left[, \begin{Bmatrix} \textit{filename.ext} \\ \text{F7RTL}\textit{xx}\text{.LIB} \end{Bmatrix} \right] \right] \right] \right]$$

The arguments are positionally dependent and must appear in the order listed. Arguments are separated by commas. When omitting an argument, a comma must be included as a placeholder. The four arguments which the CSS accepts are described below.

A — specifies argument checking for all RTL routines and functions. The default is no argument checking.

X — allows interfacing to the 3200 FORTRAN Enhancement Package (FEP) microcode functions (WCS). The default is no WCS.

W — allows interfacing to the high-speed FORTRAN package for the 8/32 processor (WCS). The default is no WCS.

C — includes the FORTRAN VII R05 compatible RTL. This library provides object compatibility so that the object code generated by R05 compilers can be linked with object generated by R06 compilers.

filename.ext — specifies the filename on which the generated RTL is to be created. The default filename is F7RTL*xx*.LIB, where *xx* is the number associated with the current software revision level (i.e., F7RTL60.LIB for the R06-00 revision level). If a filename is specified without an extension, .LIB is assumed.

The following list describes the library version referenced given the F7RTLLIB command issued. In all cases described, the filename on which the RTL is generated defaults to F7RTL*xx*.LIB unless you specify a different filename using the *filename.ext* argument.

F7RTLLIB — all RTL routines and mathematical functions are included.

F7RTLLIB A — all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program.

F7RTLLIB ,W
all RTL routines and mathematical functions are included. The RTL interfaces to the high-speed FORTRAN package for the 8/32 (WCS).

F7RTLLIB A,W
all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program. The RTL interfaces to the high-speed FORTRAN package for the 8/32.

F7RTLLIB ,X
all RTL routines and mathematical functions are included. The RTL interfaces to the Series 3200 FORTRAN Enhancement Package (FEP).

F7RTLLIB A,X
all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program. The RTL interfaces to the Series 3200 FEP.

F7RTLLIB ,,C
all RTL routines and mathematical functions are included. The RTL has R05 object compatibility.

F7RTLLIB A,,C
all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program. The RTL has R05 object compatability (with argument checking).

F7RTLLIB ,W,C
all RTL routines and mathematical functions are included. The RTL interfaces to the high-speed FORTRAN package for the 8/32 (WCS) and has R05 object compatability.

F7RTLLIB A,W,C — all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program. The RTL interfaces to the high-speed FORTRAN package for the 8/32 and has R05 object compatability (with argument checking).

F7RTLLIB ,X,C — all RTL routines and mathematical functions are included. The RTL interfaces to the Series 3200 FORTRAN Enhancement Package (FEP) and has R05 object compatability.

F7RTLLIB A,X,C — all RTL routines and mathematical functions are included along with an argument checking routine that checks the arguments of all RTL routines called by a FORTRAN program. The RTL interfaces to the Series 3200 FEP and has R05 object compatability (with argument checking).

The twelve versions of the library presented above are referenced in the LIBRARY command as follows:

LIBRARY F7RTL*xx*.LIB

or

LIBRARY *filename.ext* (if a different filename was specified
using the *filename.ext* argument)

NOTE On the 328*x* Systems (except 3280E), use the mathematical functions provided in the nonWCS System Mathematical Library instead of those provided in the 3200 Series WCS library. The transcendental functions (sine, cosine, atan, log, log10, and square root) for the nonWCS version of the library use the machine instructions, whereas the WCS library uses a microcode interface to do the same.

BUILD instructs the linkage editor to begin building the task image from the object modules. BUILD allocates the task image file and stores the task image in it.

END terminates the linkage editor.

Using the options provided by these commands, you can increase the capabilities of the basic Link command file to meet the needs of each application. For example, the SEGMENTED option to the Link OPTION command specifies that the pure code of a user-task (u-task) can be shared by two or more tasks.

Example:

```
$BUILD LINK.CMD
 ESTABLISH TASK
 MAP @1.MAP,ADDRESS
 OPTION FLOAT,DFLOAT,WORK=(XC00,XC00),
          SYSPACE=XFFFF,SEGMENTED
 INCLUDE @1.OBJ
 LIBRARY F7RTL.OBJ/S
 BUILD @1.TSK
 END
$ENDB
```

In this example, the task created consists of both a private and a shared image. The private image contains the impure code which cannot be shared by other tasks. The shared segment contains the pure code which is available to other tasks.

While most FORTRAN programs can be linked using the commands in the basic Link command file described above, FORTRAN programs handling task traps, using overlays, accessing shared data areas or shared segments require a different set of Link commands. These are described in the following sections.

Linking Trap Handling Programs

Using the RTL ENABLE subroutine, you can write programs that handle task traps. See *OS/32 System Support Run-Time Library (RTL) Reference Manual* for more information on how to write these programs. When developing a LINK procedure for trap handling programs, you must reserve 768 (hexadecimal 300) bytes of main memory for user-dedicated location (UDL) storage and increase the amount of workspace allowed for task execution. This is done through the Link OPTION command as shown in the following example.

Example:

```
$BUILD LINK.CMD
 ESTABLISH TASK
 MAP @1.MAP,ADDRESS
 OPTION FLOAT,DFLOAT,WORK=(X1600,X1600),
          SYSPACE=XFFFF,ABSOLUTE=X300,NAFPAUSE
 INCLUDE @1.OBJ
 LIBRARY F7RTL.OBJ/S
 BUILD @1.TSK
 END
$ENDB
```

In this example, the Link OPTION command is used to:

- Increase absolute data space in memory by specifying a minimum of X'300' in the ABSOLUTE option.

NOTE With the R05-05 (or higher) version of the F7RTL, the ABSOLUTE=X300 option does not have to be specified. The F7RTL automatically defines this for the LINK procedure. However, if a task calls INIT and requires more than X'300' of memory, the link command must be specifically set up to override the 'OPTION ABS=X300' command passed to LINK by the RTL. To do this, use a LINK command sequence similar to the following:

```
        .  .  .
INCLUDE @1.OBJ
INCLUDE SYS:F7RTL.OBJ/S,.INIT
OPTION ABS=xxx
LIB SYS:F7RTL.OBJ/S
        .  .  .
```

Where:

xxx memory size other than X'300'.

- Increase task workspace for execution of trap handling routines by specifying a minimum of X'1600' in the WORK option.
- Allow the task to continue execution after an arithmetic fault by specifying NAFPAUSE.

A FORTRAN program can handle three types of task traps:

- Task queue service traps including those resulting from device interrupt, intertask communication, completion of I/O proceed calls, and termination of timer routines.
- Power restoration traps occurring after power is restored following a power failure.
- Arithmetic fault traps resulting from division by zero, fixed point quotient overflow, and floating point exponent underflow or overflow.

To determine the workspace required for programs containing trap handling routines, use the following formula:

WORK = 8kB + I/O buffer space (in bytes) + (68B * # of LOGICAL UNITS) + 114B + 268B + TASKQUEUE SIZE (in bytes) + MESSAGE RING SIZE (in bytes) + *temporary space*

Where:

8kB is the workspace required for the RTL stack. This size can be changed using the OS/32 PATCH Utility. For more information, refer to the discussion of the REENTRANT directive in Chapter 3, within the section entitled, "Miscellaneous Instream Compiler Directives."

I/O buffer space is the amount of space needed (in bytes) by FORTRAN for its buffered I/O multiplied by the maximum number of logical units assigned at any given time. If there are no trap handling routines that issue I/O, this buffer space is the largest single record length needed for any one I/O. Otherwise, I/O buffer space is determined by the following method:

- For all unformatted I/Os that may occur concurrently, take the sum of their physical record sizes as determined by the FORTRAN BLOCKSIZE parameter.
- For all formatted I/Os that may occur concurrently, take the sum of their logical record sizes as determined by the FORTRAN RECL parameter.
- Add these sums to get the total physical record length for concurrent I/O.
- Finally, take the larger of this sum and the largest single record length needed for any one I/O.

68B number of logical units is used by the FORTRAN static communication area for maintaining information about each lu.

114B is the size of the FORTRAN static communications area used for other data.

268B is the maximum workspace required to support start options using the GETOPTS routine.

temporary space dynamically allocated space used by the RTL. The amount allocated depends upon the application being performed (usually 2-5KB).

Additional workspace, measured in bytes, is required for the TASKQUEUE and/or the MESSAGE RING if the task is using the task queue service trap and either of these structures is specified as being larger than default size in the call to the INIT RTL routine. The number of queue entries defaults to 48 bytes. You may increase the size using the INIT routine. The message ring does not require any workspace if the default size of two is used.

If a nondefault size TASKQUEUE and/or MESSAGE RING is specified, use the following equations to calculate the required work space:

TASKQUEUE SIZE = 4 * no. of queue entries + 8 bytes

MESSAGE RING SIZE = 76 * no. of message buffers

When an arithmetic fault occurs, OS/32 automatically pauses task execution. To allow execution to continue so that the FORTRAN RTL routine can handle the trap, the task must be prevented from being paused. This is done by linking the program with the Link option NAFPAUSE.

For more information on task trap handling, see the *OS/32 Application Level Programmer Reference Manual.*

Overlaying a Program

During its lifetime, a program may become very large. Concurrent provides a means to execute a program in an area of main storage that is not actually large enough to contain the entire task at one time. Link is used to divide such a program into nodes, a collection of modules and common blocks, which are loaded as needed. Only one node, the root, must remain in main memory throughout the execution of the program; the other nodes reside on, and are fetched from, disk when needed.

To ensure the integrity of the overlayed program, an overlay structure must be carefully designed. You can create a tree structure to show which nodes of a program occupy the same main memory at different times. Figure 7-1 illustrates a tree structure. The sample program is composed of one main routine and six subprograms, B, C, D, E, F, and X. The main routine calls B and C. C in turn calls D, which calls E and F. All routines call X, and E and F share the global variable E_AND_F.

The main routine must reside in the root node throughout the execution of the task. Also, X should be placed in the root because all other routines call X.

The execution of B and C are mutually exclusive; that is, they never call each other directly or indirectly. Therefore, these two subprograms can occupy the same address space. C must remain in storage while D, E, and F are executing. However, E and F are mutually exclusive and they can occupy the same space. E and F can be placed in separate substructures below D; therefore, D is considered to be an ancestor of E and F. However, there is nothing to be gained by separating routines C and D since they must be present simultaneously, so C and D can be placed in the same node.

Sample Program

```
|--------
| Call B
| Call C
| Call X
| END; MAIN
|--------
| Subroutine B
|
| Call X
|
| END; B
|--------
| Subroutine C
| Call D
|
| Call X
|
| END; C
|--------
| Subroutine D
| Call X
| Call E
| Call F
|
| END; D
|--
| Subroutine E
| Global E_AND_F
| Call X
| END; E
|--
| Subroutine F
| Call X
| END; F
|--
| Subroutine X
|
| END; X
|-----
```

```
              Overlay Tree Structure

                   | routine X
       |------------------------------|
       |                              |
       |                              |

routine B                      routine C
                               routine D
                                  |
                                  |
                  |--------------------------|
                  |                          |
            routine E                   routine F
```

Figure 7-1. Sample Program with Overlay Tree Structure

The following Link command sequence can be used to implement the overlay structure of Figure 7-1.

Example:

```
INCLUDE MYPROG.OBJ,.MAIN
INCLUDE ,X
OVERLAY B,1
   INCLUDE ,B
OVERLAY CD,1
   INCLUDE ,C
   INCLUDE ,D
   OVERLAY E,2
      INCLUDE ,E
   OVERLAY F,2
      INCLUDE ,F
LIBRARY MYLIB.OBJ
LIBRARY F7RTL.OBJ
BUILD MYPROG
```

The OVERLAY command specifies the start of a node and the node's relative position within the tree structure. The two RTL files, MYLIB and the standard RTL, will be searched by Link (MYLIB first, then F7RTL.OBJ) for any routines containing entry points matching the unresolved external references of the program. It will place a copy of a library routine in the referencing node unless an ancestor already contains a copy.

Care should be taken to place all LIBRARY commands which reference user libraries before the RTL LIBRARY command. This ensures that each user library routine gets resolved against the standard RTL. Also, it should be remembered that the domain of a LIBRARY command is the entire Link command sequence. That is, its domain is not restricted to the overlay in which it was placed; only the order of the LIBRARY commands are significant to Link.

Each node has a fixed length in bytes. The total size of a task depends upon both the routine composition of each node and the structure of the overlay tree. An overlay structure can be represented by a set of parallel paths. A path can be defined as a particular set of nodes (one at each level) each of which is a descendant from the previous level. Therefore, the total size of a task is determined by the path whose node sizes add up to the greatest number of bytes. By using the cross-reference map from Link, one can manually build a call-tree representation of a program (similar to the one shown in Figure 7-1) as an aid in determining the smallest possible task size.

Normally, the placement of a common block or global block within an overlayed task is determined by where the block is referenced. Blank common is always positioned in .ROOT. Named common and global blocks, however, are initially positioned by Link no closer to the root than any particular reference to the block. In the sample program of Figure 7-1, subprograms E and F both reference the global variables E_AND_F. Link will place E_AND_F in the node containing subprograms C and D.

There are two consequences to this positioning policy. The first consequence is that named common and global entities are initialized every time the overlay is fetched from disk. The second consequence is that more than one copy of a common or global entity can exist on separate paths in the program. That is, two or more overlays can have their own separate and private copies of a common or global entity. These copies could then contain different values.

Link provides the POSITION command to reposition common or global entities into an overlay closer to the root than it would normally position them. Global E_AND_F, in the sample program, can be forced into the root node by inserting:

```
POSITION Common=E_AND_F,To=.ROOT
```

into the sample Link command sequence. Notice here that global entities are considered as common.

Common blocks and global entities are not the only entities affected by overlaying a program; implicitly saved local entities are also affected. A program containing an implicitly saved local entity depends upon the value of that entity to remain unchanged between invocations. Very subtle bugs can occur in an overlayed program if the value of the entity is well defined at one point during the execution of the program, but becomes undefined at another. The FORTRAN SAVE statement will avoid this problem. The local entities specified on a SAVE statement are repositioned by Link to the root node via a compiler generated DCMD command to Link in the object code. Thus, the values of entities specified on a SAVE statement are guaranteed to be the most recent. The SAVE statement may also be used to reposition common blocks and global entities to the root.

Linking Shared Data Areas

A FORTRAN program can reference data areas that can be read or written to by other tasks running on the same or different processors. Two requirements must be met for this particular situation.

- You must build a partial image containing the data areas to be accessed if it does not yet exist; and
- You must link your program using the Link RESOLVE command.

Shared data areas must be built and linked into shared image modules before they can be specified in the RESOLVE command. To do this, perform the following steps:

1 Build a data area by using a FORTRAN block data subprogram. Save this block in the file DATA1.FTN.

Example:

```
C THIS BLOCK DATA SUBPROGRAM BUILDS
C A DATA AREA CONSISTING OF BOTH
C NAMED COMMON AND GLOBAL COMMON
C VARIABLES

      BLOCK DATA    DATA1
      GLOBAL A,B,C,D,E
      COMMON /ABC/I,J,K
      COMMON /DEF/L,M,N
      REAL A,B,C,D,E
      DATA A,B,C,D,E,I,J,K,L,M,N/5*0.0,6*0/
      END
```

2 Compile this source to create the object file.

```
COMPILE DATA1.FTN
```

3 Establish a block data structure as a shared data area by using the following Link .CMD file:

```
ESTABLISH IMAGE,ACCESS=RW,ADDRESS=F0000
INCLUDE DATA1
EXTERNAL ABC.,DEF.,A,B,C,D,E
BUILD DATA1.IMG
END
```

The ADDRESS parameter in the ESTABLISH command selects the segment number to be assigned to the task common. Once this is determined and the common is established, LINK uses this information each time a task is built and the task common is resolved against it.

When defining a task common entry point or name, the EXTERNAL statement is used. The external name of a common block is the seven character ASCII name used in the COMMON statement with a period appended to the end. If the name is eight characters long, the external name of the common block is the name itself. Note how COMMON /ABC/ maps to the EXTERNAL ABC. in the previous example.

In this command sequence, DATA1 is not only the name of the block data subprogram, but also the name of the file containing the object code for the subprogram.

These Link commands establish DATA1 as a shared data area containing DATA1. Items within the shared area are arranged exactly as they are arranged within the block data structure. Each shared area can contain more than one block data structure. These structures are arranged within the shared area according to the order in which they are included by the Link INCLUDE command.

When establishing the shared area, all global variables as specified in the FORTRAN VII GLOBAL statement and named common blocks to be contained in that area must be listed in the Link EXTERNAL command. The compiler truncates common block names and GLOBAL variables to eight characters. If a name is less than eight characters, a period is appended to the name (e.g., ABC. and DEF.). Global entities must also be truncated to eight characters.

To link a FORTRAN task that references the shared data area contained in DATA1.IMG, create a link .CMD file as follows:

Example:

```
$BUILD LINK.CMD
 ESTABLISH TASK
 MAP @1.MAP,XREF
 OPTION DFLOAT,FLOAT,WORK=(XC00,XC00),
        SYSSPACE=XFFFF
 INCLUDE @1.OBJ
 RESOLVE DATA1.IMG
 LIBRARY F7RTL.OBJ/S
 BUILD @1.TSK
 END
$ENDB
```

When establishing a shared data area that is to be located in the global task common (memory shared by two or more distinct processors), use the name of the global task common as the argument to the Link BUILD command.

This name is determined by the TCOM command at system generation (sysgen). For example, if DATA1 is to be established as a shared area within a global task common named GTC, the Link BUILD command would be written as follows:

```
BUILD GTC
```

where GTC is the name given by the system administrator at sysgen time to that shared data area.

To link a FORTRAN task that references the shared data area containing GTC, the link RESOLVE command would be written as follows:

```
RESOLVE GTC
```

Because OS/32 does not support static initialization within global task common (TCOM) areas, block data subprograms used for structuring shared data within global task common must not contain data statements. Otherwise, these statements have no effect at run-time.

You have encountered most of the commands in LINK.CMD earlier in this chapter with the exception of the Link RESOLVE command. The RESOLVE command establishes a FORTRAN task image that references the shared area, DATA1.IMG.

Linking Shared Segments

If more than one task will be using a reentrant RTL, the RTL or individual modules of it, can be included in a shared segment. You can build this segment as shown in the following example.

Example:

```
ESTABLISH IMAGE,ACCESS=RE,ADDRESS=F0000
INCLUDE F7RTL.LIB
BUILD F7RTL.IMG
END
```

ESTABLISH This command specifies that a partial image is built with read/execute access privileges.

INCLUDE This command specifies that all object modules in the input file, F7RTL.LIB, are included in the image.

BUILD This command builds the partial image from the object modules specified in the INCLUDE commands and saves the image in the file F7RTL.IMG.

END This command terminates the linkage editor.

Once the partial image F7RTL.IMG exists, you can create the following .CMD file to resolve this partial image into your FORTRAN task.

```
RESOLVE F7RTL.IMG
INCLUDE MOD3
OPTION DFLOAT,FLOAT,WORK=X1770
MAP PR1:,ALPHABETIC,XREF
BUILD MOD3
END
```

Example 2:

The RTL routine .U, which initializes the FORTRAN environment cannot be placed in an RTL shared segment (as in Example 1) unless execution profile analysis (XPA) and call recording analysis (CRA) are not needed. This is because .U calls the XPA initialization routine .XPATIMR, which gets resolved to the entry point in the dummy module .XPADUMY. See Chapters 12 and 13 for a discussion on the XPA and CRA Systems, respectively.

To build a shared RTL that contains all modules except .U and .XPADUMY, enabling the use of XPA and CRA, use the OS/32 Library Loader as follows:

```
*AL F7RTL.TMP,IN.126/8          * allocate the file
*LO LIBLDR                      * load the Library Loader
*AS 1,F7RTL.LIB/S,SRO           * assign the system RTL to lu 1
*AS 2,F7RTL.TMP                 * assign the allocated file to lu 2
*AS 3,NULL:                     * assign null device to lu 3
*AS 5,CON:                      * assign the terminal to lu 5
*ST                             * start the Library loader
>DU 1,2 .XPADUMY                * dupe all modules until .XPADUMY
>CO 1,3                         * do a copy just to go past .XPADUMY
>DU 1,2 .U                      * dupe following modules until .U
>CO 1,3                         * do a copy just to go past .U
>DU 1,2                         * dupe the remaining modules
>END                            * end the Library Loader
```

The above sequence produces an RTL without the modules .XPADUMY and .U. See the *OS/32 Library Loader Reference Manual* for more details on the utility.

Now, use Link to produce the shared segment using the following commands:

```
ESTABLISH IMAGE,ACCESS=RE,ADDRESS=F0000
INCLUDE F7RTL.TMP
BUILD FORTLIB.IMG
END
```

By building the Link command file as shown in the following example, references to shared segments specified by the RESOLVE command are placed in the FORTRAN task. The shared segment, in this case F7RTL.IMG, must be available at program execution.

To link the object MOD3.OBJ with the partial image F7RTL.IMG, create the following .CMD file:

```
RESOLVE F7RTL.IMG
LIBRARY F7RTL.LIB/S
INCLUDE MOD3
OPTION DFLOAT,FLOAT,WORK=X1770
MAP PR1:,ALPHABETIC,XREF
BUILD MOD3
END
```

DCMD Messages

The define command (DCMD) is a Link command that enables execution of Link commands in the object modules. It also enables listing of embedded comments to the input or log device. The FORTRAN VII compilers generate, as a default, a DCMD in the object file for subprograms which generate supervisor call (SVC) instructions or nonnormalizing floating point load instructions. Link commands or comments may also be embedded using the $DCMD directive, which is discussed in Chapter 3.

When the APU compiler option is specified, the linkage editor will output the message contained in the DCMD on the LOG file. The format of this message is:

```
****MODULE xxxx INVOKES SVC
```

Where:

xxxx is the name of the module.

If the compiler generates a DCMD in the object file for a subprogram in which nonnormalizing floating point load instructions were generated, the Linkage editor will output the message contained in the DCMD on the log file. The format of this message is:

Where:

xxxx	is the name of the subprogram.

If $RTOLBIT or $NTRANSCENDENTAL are specified within a subprogram, the Linkage editor will output the message contained in the DCMD on the log file. The format of this message is:

```
****MODULE xxxx COMPILED WITH $directive
```

Where:

xxxx	is the name of the subprogram.
directive	is either RTOLBIT or NTRANSCENDENTAL.

Loading and Executing Link

To load Link into memory, use the OS/32 LOAD command. To start Link execution, use the Link START command specifying the name of the command file in the command parameter and the log device in the log parameter. If the START parameters are not specified, the parameters will default to the terminal.

Testing End of Task Codes for Link

The Link process can terminate under any one of the following four conditions.

- All object code was linked into task image code.
- All object code was linked into image code, but found to contain unresolved external references.
- Linking of the object code was terminated due to a Link error.
- Link aborted and no object code was linked; i.e., an error resulted from a command within the program development procedure.

These conditions and their end of task codes are listed in Table 7-2.

End Of Task Code	Link Termination Status
0	Normal termination
1	Link errors
2	Linking aborted some code linked
3	Linkage editor aborted before any code was linked

Table 7-2. Link End of Task Codes

The program development procedure must test for an end of task code 0 before proceeding to execute the task image. To do this, the procedure uses the CSS conditional commands as shown by LINK1.CSS.

Example:

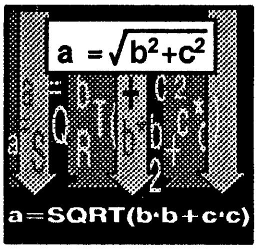
```
$IFNE 0
$WRITE   LINK ERRORS
$CLEAR
$ENDC
```

In the previous example, the procedure checks whether the end of task code is not equal to 0. If it is equal to 0, the operating system skips over the next three commands and begin task execution. If it is not 0, the operating system sends the message LINK ERRORS to the terminal. After the message is sent, $CLEAR terminates the program development procedure.

The program development procedure can also test for the other end of task codes.

Example:

```
$IFE 2
$WRITE   CHECK EXTERNAL REFERENCES
$CLEAR
$ELSE
$IFG 2
$WRITE   LINK ERRORS
$CLEAR
$ENDC
$EXIT
```



8

Building a Command File to Execute a FORTRAN Program

In this chapter

We teach you how to load and start the task created by the LINK process.

Topics include:

- Loading and starting the task
- Assigning logical units
- Testing end of task codes

Introducing the Basic Execute Procedure

The minimum functions that must be performed by the operating system to execute a program are as follows:

- Load the task image file.
- Assign logical units to the required input/output (I/O) files.
- Start execution.
- Check the end of task code.
- Write an error message if end of task code is greater than 0.

The following command sequence performs the basic operating system functions for executing a FORTRAN program.

Example:

```
LOAD @1.TSK
ASSIGN 1,DATAFILE.IN
ASSIGN 3,CON:
ASSIGN 6,PR:
ASSIGN 5,CON:
START
$IFNE 0
$WRITE Run-Time Error encountered
$CLEAR
$ENDC
$EXIT
```

You can save this command sequence in a command substitution system (CSS) file such as RUN1.CSS. (Avoid naming CSS files the same as the system CSS files. RUN.CSS is a system CSS.)

Loading and Starting the Task Image

To load the task image into memory, use the operating system LOAD command. It is not necessary to state the memory increment size. The Link OPTION WORK command allocates the necessary workspace for task execution. To begin execution, use the operating system START command.

Assigning Logical Units

All FORTRAN I/O statements require an logical unit (lu) assigned to the I/O device or file used by the program. FORTRAN VII has default lu assignments for I/O statements that either do not specify an lu or use * as the lu. These defaults are listed in Table 8-1.

FORTRAN I/O Statement	lu	Assignments
READ	1	CR:
WRITE	3	PR:
PRINT	3	PR:
ACCEPT	5	CON:
TYPE	5	CON:

Table 8-1. FORTRAN VII Default Logical Unit Assignments

In addition, lu6 is the default lu for TRACE and TEST output. This lu must be assigned by the user.

FORTRAN VII RTL also needs the run-time error file, F7RTL60.ERR. This file should be on the system volume on account 0. The file is opened, read, and closed on each I/O encountered by IOERR. The run-time library (RTL) assigns the error file dynamically to a free lu. If the error file does not exist or a free lu cannot be found, no details of the error message are given. The error file cannot be preassigned.

The program development procedure can assign the logical units as follows.

Example:

```
LOAD @1.TSK
ASSIGN 1,@3
ASSIGN 3,@4
ASSIGN 6,PR:
ASSIGN 5,CON:
START  @2
```

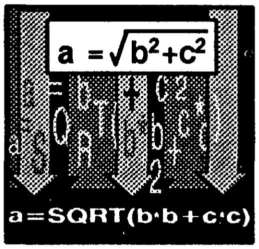
To assign a particular device to lu1 or lu3, the user specifies that device in the third and fourth parameter positions of the EXEC command as follows:

*** EXEC1 FORTPROG,TRACE TEST COMP,DATAFILE.IN,CON:**

Testing End of Task Codes

Routines within the RTL check for errors during program execution. Depending on the error, the RTL routines output an error message to the console or terminal and, in certain cases, conditionally pause execution so that the user can take the appropriate action to resolve the error.

Normal termination of a task yields end of task code 0. However, the user can change the end of task code by calling the RTL EXIT routine. Users can control the handling of execution exceptions or faults by writing task trap handling routines within the source program. See the *OS/32 System Support Run-Time Library (RTL) Reference Manual* for more information on using the real-time processing RTL routines.



9

Run-Time Debugging

In this chapter

We provide you with another look at the debug directives introduced in Chapter 3. These directives are designed to support run-time debugging. With additional aid from a set of run-time library (RTL) routines, these utilities provide you adequate tools in debugging your program.

Topics include:

- Conditional compilation
- Tracing variables and executable statements
- Checking array subscripts
- Using RTL version which performs argument checking

Basic Debugging Concepts

After you have successfully compiled your program, there is still no guarantee that it is free of errors. FORTRAN VII provides three directives and a set of RTL routines that are designed to support debugging. Chapter 3 describes the available debug directives. This chapter provides further details on how you can use them.

Essentially, debugging is testing the code during run-time. Debugging includes:

- Running the program with test variables and analyzing the results,
- Checking intermediate values of a variable as the program is executed,
- Tracing the flow of control throughout the program,
- Checking that array elements are within their declared bounds, and
- Analyzing run-time error messages.

Compiling Code Using $COMP/$NCOMP

The FORTRAN VII compilers provide a conditional compilation facility that allows debugging code to be incorporated into a program without having to delete them after you are done with the debugging process. An 'X' in column one of the initial line of a FORTRAN statement flags the statement as a conditionally compiled statement. When you compile the program with the $COMP directive specified, the debugging code becomes part of the program. When debugging is completed, you do not have to perform the time consuming and often error prone job of removing each line of test code individually. One simply replaces the $COMP directive with the $NCOMP directive before recompiling the program.

Recompiling the program with $NCOMP causes the debugging code to be compiled as comment lines with the # character placed in column one. The debugging code can be reactivated by replacing the $NCOMP directive with $COMP and recompiling the program.

NOTE The COMP/NCOMP start directive may be used instead of the instream directive $COMP/$NCOMP. By using the start directives, you do not have to modify your source code.

Example:

```
   SUBROUTINE SUM_SQUARES(S,X,I)
C  THIS SUBPROGRAM USES $COMP TO TEST
C  THE CODE FOR SPECIFIED VALUES
C  OF ARRAYS X AND I.
C
$COMP
      REAL S, X(3)
      INTEGER I(2)
X     X(1) = .2
X     X(2) = .3
X     X(3) = .4
X     I(1) = 0
X     I(2) = 1

        IF(I(1).EQ.1.AND.I(2).EQ.0) THEN
             S=X(1)**2 + (-X(2))**2 + (-X(3))**2
X            WRITE (*,40) S
        ELSE
             IF(I(1).EQ.1.AND.I(2).EQ.1) THEN
               S=X(1)**2 + (-X(2))**2 + X(3)**2
X              WRITE (*,40)S
             ELSE
               IF(I(1).EQ.0.AND.I(2).EQ.0) THEN
                  S=X(1)**2 + X(2)**2 + (-X(3))**2
X                 WRITE (*,40) S
               ELSE
                  S=X(1)**2 + X(2)**2 + X(3)**2
X                 WRITE (*,40) S
               END IF
             END IF
        END IF
X  40 FORMAT (1X,F15.8)
      RETURN
      END
```

In the previous example, the first five conditionally compiled statements assign values to each array element. Those values are used to evaluate S. The value of S is then output and the program terminated. You can then check the value of S to see if it is valid for the conditionally assigned values. Once satisfied that the subprogram works as intended, replace the $COMP directive with the $NCOMP directive and recompile the program. Note that this subprogram must be called by a main program to execute.

Checking Intermediate Values with $TRACE

$TRACE allows you to check the value of a variable which is redefined by an assignment statement without having to insert a write statement after each assignment.

Example:

```
   SUBROUTINE SUM_SQUARES(S,X,I)
C  THIS SUBPROGRAM USES $TRACE TO CHECK
C  THE VALUE OF S.

$COMP
$TRACE S
      REAL S, X(3)
      INTEGER I(2)
X     X(1) = .2
X     X(2) = .3
X     X(3) = .4
X     I(1) = 0
X     I(2) = 1

        IF(I(1).EQ.1.AND.I(2).EQ.0) THEN
             X=X(1)**2 + (-X(2))**2 + (-X(3))**2
        ELSE
             IF(I(1).EQ.1.AND.I(2).EQ.1) THEN
               S=X(1)**2 + (-X(2))**2 + X(3)**2
             ELSE
               IF(I(1).EQ.0.AND.I(2).EQ.0) THEN
                  S=X(1)**2 + X(2)**2 + (-X(3))**2
               ELSE
                  S=X(1)**2 + X(2)**2 + X(3)**2
               END IF
             END IF
        END IF
      RETURN
      END
```

When this subprogram is called by a main program, $TRACE automatically outputs the following message to logical unit 6 (lu6).

```
S = 0.29
```

When variables are traced, the format of the message is:

id=value

Where:

id is the name of the variable being traced.

value is the current value of *id*. Logical values are output as T or F. A complex value is output as two floating point numbers separated by a comma and enclosed in parentheses. Floating point values use the F, E, or D format depending on the magnitude of the data. Integers use the I format. A character is output as a quoted string.

Tracing Executable Statements

To find out when a particular statement is executed within a program, label that statement and insert $TRACE above its first occurrence in the source program. This causes all variables and labeled statements to be traced for that section of the program between the $TRACE <*label*> statement and the statement labeled <*label*>. For example, to know which statement within the BLOCK IF the program branches to, the appropriate statement would be labeled as shown in the following example:

Example:

```
   SUBROUTINE SUM_SQUARES(S,X,I)
C  THIS SUBPROGRAM USES $TRACE TO TRACE
C  THE FLOW OF CONTROL WITHIN THE BLOCK IF
$COMP
$TRACE 5
$TRACE S
      REAL S, X(3)
      INTEGER I(2)
X     X(1) = .2
X     X(2) = .3
X     X(3) = .4
X     I(1) = 0
X     I(2) = 1

        IF(I(1).EQ.1.AND.I(2).EQ.0) THEN
  1          S=X(1)**2 + (-X(2))**2 + (-X(3))**2
        ELSE
             IF (I(1).EQ.1.AND.I(2).EQ.1) THEN
  2            S=X(1)**2 + (-X(2))**2 + X(3)**2
             ELSE
               IF(I(1).EQ.0.AND.I(2).EQ.0) THEN
  3                 S=X(1)**2 + X(2)**2 + (-X(3))**2
               ELSE
  4                 S=X(1)**2 + X(2)**2 + X(3)**2
               END IF
             END IF
        END IF
  5   RETURN
      END
```

As this code is executed, $TRACE sends the following output to lu6.

```
STATEMENT LABEL 4
S = 0.29
STATEMENT LABEL 5
```

The format of a label trace message is:

```
STATEMENT LABEL n
```

In this format, *n* is the label of the statement being traced. This message is output before the statement is executed.

Checking Array Subscripts Using $TEST

The $TEST directive is used to check whether all array elements referenced within the program are within their declared bounds. $TEST checks the array subscript and substring bounds referenced in arithmetic, logical, and character expressions. However, it does not check boundary violations of arrays passed as arguments to subprograms or used as a buffer in ENCODE/DECODE statements.

Example:

```
   SUBROUTINE SUM_SQUARES (S,X,I)
C  THIS SUBPROGRAM USES $TEST TO CHECK
C  THE ARRAY ELEMENT SUBSCRIPTS AGAINST THEIR
C  DECLARED BOUNDS
$COMP
$TEST
      REAL S, X(3)
      INTEGER I(2)
X     X(1) = .2
X     X(2) = .3
X     X(3) = .4
X     I(1) = 0
X     I(2) = 1

      IF(I(1).EQ.1.AND.I(2).EQ.0) THEN
  1          S=X(1)**2 + (-X(2))**2 + (-X(3))**2
      ELSE
             IF (I(1).EQ.1.AND.I(2).EQ.1) THEN
  2            S=X(1)**2 + (-X(2))**2 + X(3)**2
             ELSE
               IF(I(1).EQ.0.AND.1(2).EQ.0) THEN
  3               S=X(1)**2 + X(2)**2 + (-X(3))**2
               ELSE
  4               S=X(1)**2 + X(2)**2 + X(4)**2
               END IF
             END IF
      END IF
  5   RETURN
      END
```

The FORTRAN VII compiler issues a warning for the statement labeled 4 since the constant subscript of X exceeds its bounds.

Since the statement labeled 4 is written as:

```
S=X(1)**2 + X(2)**2 + X(4)**2
```

and $TEST is specified, the following message is sent to lu6 during execution.

```
ERROR IN .MAIN AT LINE 28, ARRAY X:  DIM3 (1:3) SUBSCRIPT = 4
```

This message tells the user:

- the program unit in which the error occurred,
- the line number of the statement in which the error occurred,
- the name of the array,
- the original dimensions of the array, and
- the subscript that is out of bounds.

The format of the test message for out of bounds subscripts is:

```
ERROR IN subpro AT stmtno     aname:DIMd(lb:ub) SUBSCRIPT = S
```

In this format, *subpro* is the name of the program unit in which the error occurred, *stmtno* is the number of the source statement in which the error occurred, *aname* is the array identifier and *d* is the dimension number. The lower and upper bounds of the dimension are indicated by (*lb:ub*). *S* is the value of the subscript that is out of bounds.

The format of the test message for out of bounds substrings is:

```
ERROR IN ubpro AT stmtno     cname*len SUBSTRING=(b:e)
```

The identifier of the character string is *cname*. The declared length of *cname* is *len*. The beginning and ending substring values that caused the string to be out of bounds are indicated by (*b:e*).

Test messages are output before the statement is executed.

RTL Argument Checking

The Concurrent RTL is available with or without argument checking code. An argument checking RTL contains user-controllable code that automatically checks:

- the class of an RTL call; e.g., function or subroutine,
- the type of function; e.g., REAL or INTEGER,
- the class of arguments to an RTL call; e.g., array, array element, or scalar,
- the type of arguments; e.g., REAL or INTEGER, and
- the number of arguments.

The internal RTL routine .CHECK performs the actual checking of the type of argument. A user written RTL routine can use .CHECK if the routine has the correct interface. See Appendix A for more information on .CHECK.

This type of RTL argument checking code is user-controllable; that is, it can be turned off by calling ICHECK as follows:

```
CALL ICHECK (I)
```

If the argument *I* is an integer*4 zero and argument checking is in effect, a call to ICHECK turns off user-controllable argument checking for all subsequent RTL invocations. A call to ICHECK with a nonzero argument turns on the argument checking. The default for the argument checking RTL is to perform this checking.

While user-controllable argument checking code is available only with an argument checking RTL, many routines in both types of libraries automatically check the actual values passed to them. For example, EXP checks that its argument is in the range -180.0 to +174.0. The code to do this type of checking is present in both the argument checking and nonargument checking libraries. It cannot be turned off.

NOTE ▷ Once debugged, a task executes faster if the user-controllable argument checking is turned off. In addition, a program that uses the nonargument checking RTL requires less memory.

Analyzing Run-Time Error Messages

To aid in analyzing error messages, the RTL routine ERLU allows run-time error messages to be interposed with run-time output.

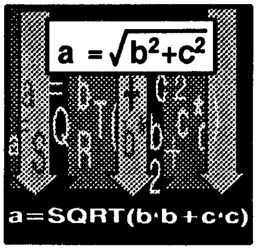
Run-time error messages can be logged to the system console or the lu designated ɔy the subroutine ERLU. ERLU is called as follows:

```
CALL ERLU (l)
```

The argument *l* is the INTEGER*4 lu number in the range 0 to the maximum lu for the task.

Removing the Debugging Aids

After debugging the program, delete all debug directives, remove all debugging codes, and recompile the program. $TRACE and $TEST can be deactivated by inserting $NTRACE and $NTEST directives. If modifying the program, it may be convenient to retain the debugging code as comments. The debugging code can also be superseded by specifying NTRACE, NTEST, and NCOMP as start directives.



10

Analyzing Program Maps and Listings

In this chapter

We illustrate the different types of information provided by the compiler listings and link maps. The different listings and maps produced by the FORTRAN VII compilers and OS/32 Link, respectively, provide a means for desk checking the source for bugs. This chapter uses the output that would result from compiling and linking the sample program presented in Figure 10-1. A different example is used to illustrate the extended listing generated by the F7Z compiler. Important items within the listings are identified by numbered balloons attached to arrows pointing to the items.

Topics include:

- Compiler source and cross-reference listing
- Compiler batch statistics
- Compiler optimization and register allocation summary
- F7Z extended source listing
- Compiler assembly language listing
- Link maps

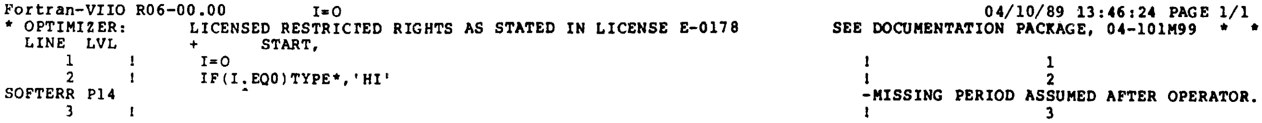
Source Listings

Desk checking is the first step in program debugging. In this step, the source code is checked for syntax and logic errors. Source listings are designed to aid the programmer in performing this step.

A source listing is comprised of all lines of source code as they are input to the compiler. Diagnostic messages are printed after each statement that is syntactically incorrect (does not adhere to rules of the FORTRAN language.) The caret (^) is used to indicate the position of syntactical errors which have generated the diagnostic messages.

There are three types of compiler run-time diagnostics generated by the FORTRAN VII compilers: WARNING, SOFTERR, and ERROR. A WARNING message indicates that no syntax error occurred, but the code as written could result in error upon execution. A SOFTERR message is produced when a syntax error is detected, but the compiler is able to take a corrective action and continues with compilation. The following shows an example of a SOFTERR.

Example:



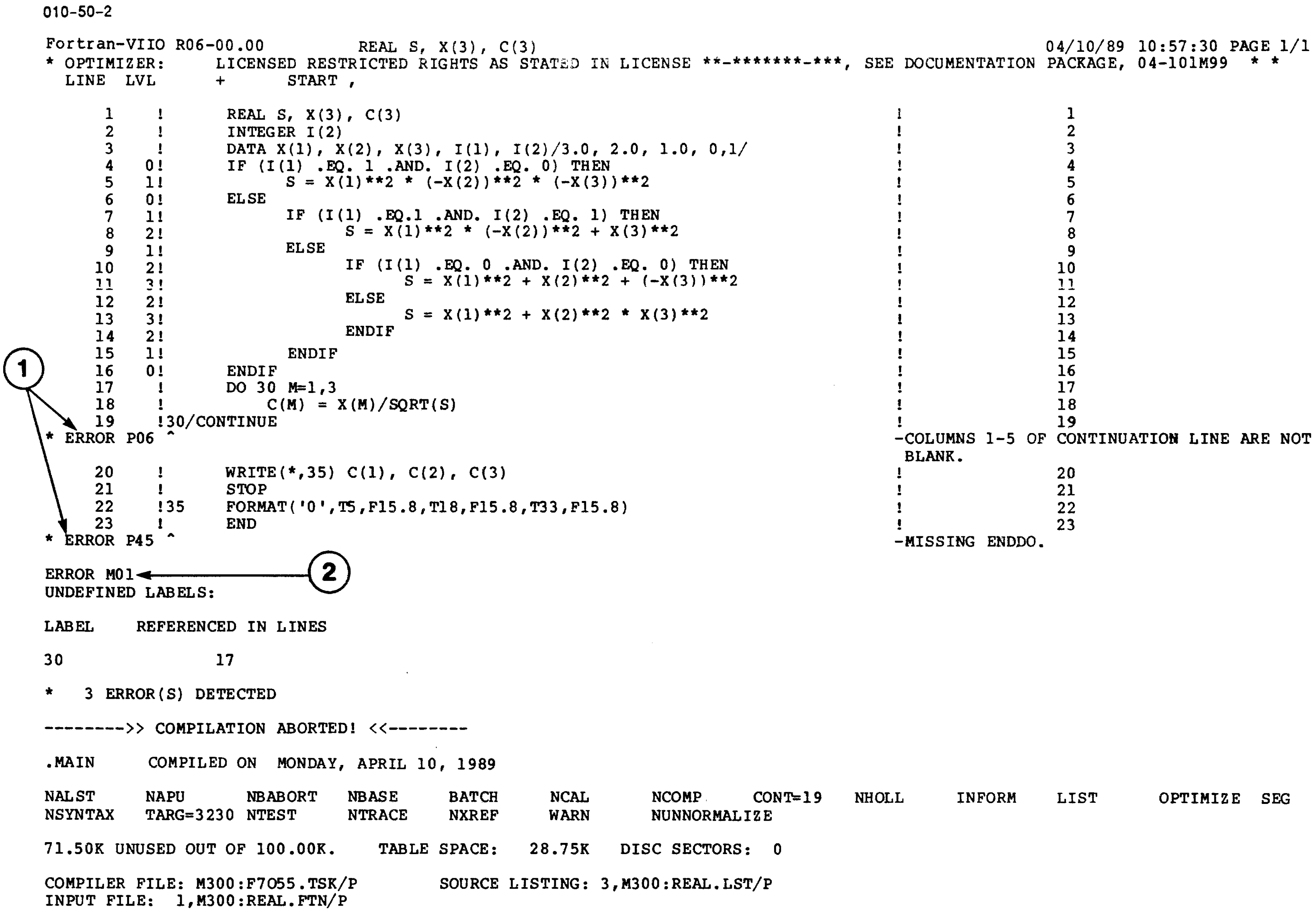
```
Fortran-VIIO R06-00.00        I=O                                                   04/10/89 13:46:24 PAGE 1/1
* OPTIMIZER:        LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE E-0178        SEE DOCUMENTATION PACKAGE, 04-101M99  *  *
  LINE  LVL         +      START,
     1     !         I=O                                                             !                 1
     2     !         IF(I.EQ0)TYPE*,'HI'                                             !                 2
SOFTERR P14             ^                                                            -MISSING PERIOD ASSUMED AFTER OPERATOR.
     3     !                                                                         !                 3
```

ERROR denotes the presence of a fatal error. A fatal error causes compilation to be aborted while WARNING and SOFTERR do not interrupt compilation. All fatal errors must be corrected before program processing can continue.

F7O Source Listing with Compilation Errors

Figure 10-1 is an example of an F7O source listing with compilation errors. Numbered items contained in this listing are identified as follows.

Number	List Item
1	These diagnostic messages indicate that the DO statement in line 17 has no ending statement because line 19 has a syntax error.
2	A diagnostic message indicating that the label referenced in line 17 was not defined.
3	A warning message is displayed indicating that a label is defined for a FORMAT statement, but no reference is made to that statement in the program.
4	The total number of errors detected by the compiler.



```
010-50-2

Fortran-VIIO R06-00.00          REAL S, X(3), C(3)                                              04/10/89 10:57:30 PAGE 1/1
* OPTIMIZER:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99   * *
  LINE  LVL      +       START ,

     1     !      REAL S, X(3), C(3)                                                   !              1
     2     !      INTEGER I(2)                                                         !              2
     3     !      DATA X(1), X(2), X(3), I(1), I(2)/3.0, 2.0, 1.0, 0,1/                !              3
     4    0!      IF (I(1) .EQ. 1 .AND. I(2) .EQ. 0) THEN                              !              4
     5    1!            S = X(1)**2 * (-X(2))**2 * (-X(3))**2                          !              5
     6    0!      ELSE                                                                 !              6
     7    1!            IF (I(1) .EQ.1 .AND. I(2) .EQ. 1) THEN                         !              7
     8    2!                  S = X(1)**2 * (-X(2))**2 + X(3)**2                       !              8
     9    1!            ELSE                                                           !              9
    10    2!                  IF (I(1) .EQ. 0 .AND. I(2) .EQ. 0) THEN                  !             10
    11    3!                        S = X(1)**2 + X(2)**2 + (-X(3))**2                 !             11
    12    2!                  ELSE                                                     !             12
    13    3!                        S = X(1)**2 + X(2)**2 * X(3)**2                    !             13
    14    2!                  ENDIF                                                    !             14
    15    1!            ENDIF                                                          !             15
    16    0!      ENDIF                                                                !             16
    17     !      DO 30 M=1,3                                                          !             17
    18     !          C(M) = X(M)/SQRT(S)                                              !             18
    19     !30/CONTINUE                                                                !             19
* ERROR P06 ^                                                                          -COLUMNS 1-5 OF CONTINUATION LINE ARE NOT
                                                                                        BLANK.
    20     !      WRITE(*,35) C(1), C(2), C(3)                                         !             20
    21     !      STOP                                                                 !             21
    22     !35    FORMAT('0',T5,F15.8,T18,F15.8,T33,F15.8)                             !             22
    23     !      END                                                                  !             23
* ERROR P45 ^                                                                          -MISSING ENDDO.

ERROR M01
UNDEFINED LABELS:

LABEL     REFERENCED IN LINES

30              17

*   3 ERROR(S) DETECTED

-------->> COMPILATION ABORTED! <<--------

.MAIN      COMPILED ON   MONDAY, APRIL 10, 1989

NALST     NAPU      NBABORT   NBASE     BATCH     NCAL      NCOMP     CONT=19   NHOLL     INFORM    LIST      OPTIMIZE  SEG
NSYNTAX   TARG=3230 NTEST     NTRACE    NXREF     WARN      NUNNORMALIZE

71.50K UNUSED OUT OF 100.00K.    TABLE SPACE:    28.75K   DISC SECTORS:  0

COMPILER FILE: M300:F7055.TSK/P        SOURCE LISTING: 3,M300:REAL.LST/P
INPUT FILE:  1,M300:REAL.FTN/P
```

Figure 10-1. Example of F7O Source Listing with Compilation Errors

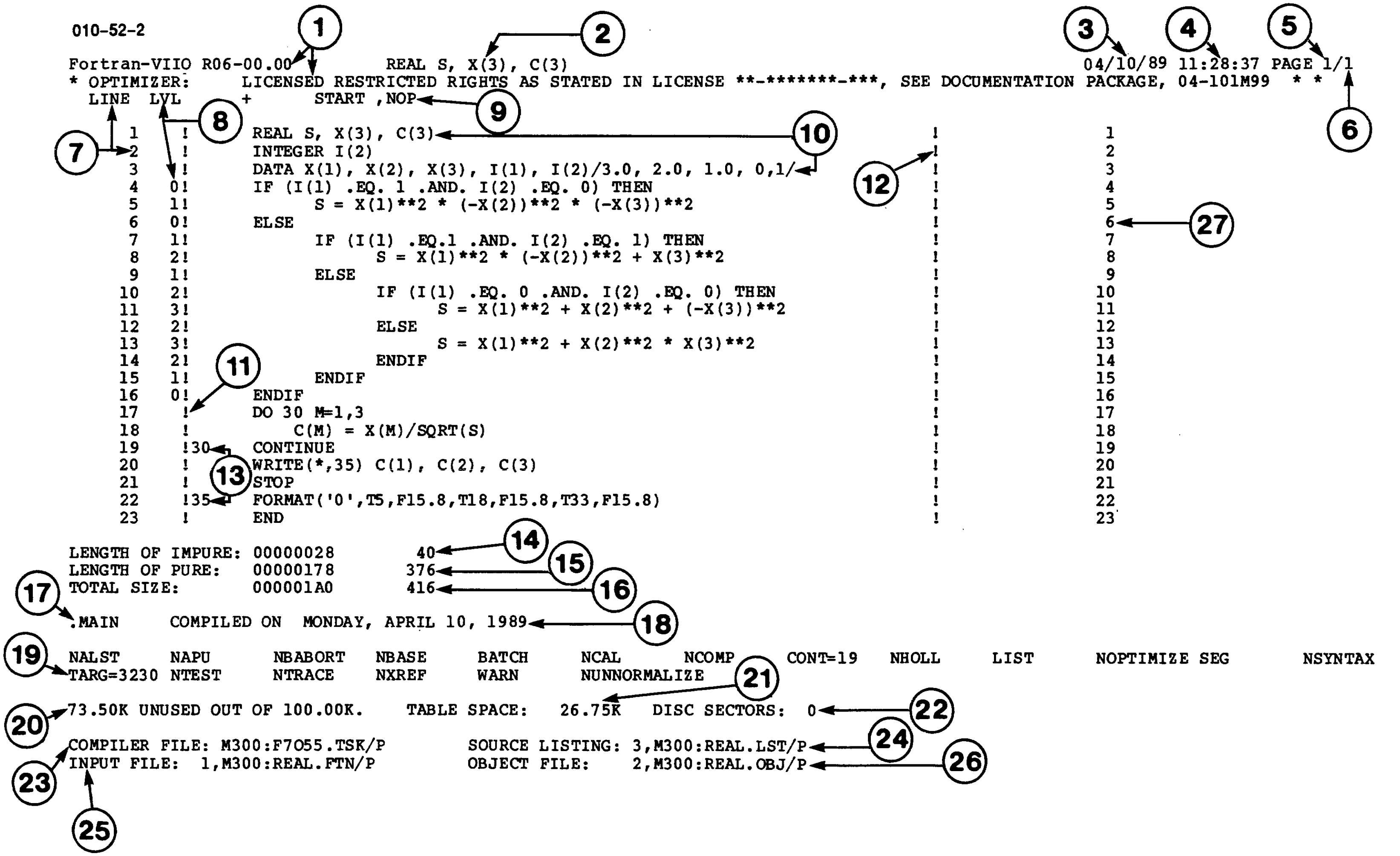
F7O and F7Z Source Listing Without Compilation Errors

An example of the source listing format for a program compiled without errors under F7O and F7Z is shown in Figure 10-2. Numbered items contained in this figure are identified as follows.

Number	List Item
1	FORTRAN compiler identification including compiler name, release and revision number, license number, and documentation package number.
2	Title of the program. If no title is specified through the TITLE directive, the compiler prints the first line of the source code.
3	Date compilation was performed.
4	Time compilation began.
5	Page number of listing for current program unit.
6	Page number of batch listing.
7	Line number assigned to each line of code by compiler.
8	Level number of nested IF statements.
9	Compiler directives specified by the user in the START command.
10	Actual lines of source code as input to the compiler.
11	Left-most source delimiter (column 0).
12	Right-most source delimiter (column 73).
13	Statement labels.
14	Length of impure code in bytes.
15	Length of pure code.

Number	List Item
16	Total size of the program object code in bytes.
17	Name of program unit compiled (compiler automatically assigns a name .MAIN, if PROG directive is not specified or if the PROGRAM statement is not used for a main program unit).
18	Day and date of compilation.
19	Compiler directives in effect during compilation.
20	Amount of workspace not used by compiler out of the total workspace allotted to it in the LOAD command.
21	Table space used by the compiler for processing.
22	Number of disk sectors used for compiler generated tables.
23	Name of the file that contains the compiler used to compile the source program.
24	Logical unit (lu) and device or file assigned for output of source listing.
25	lu and name of input device or file containing the source code.
26	lu and device or file assigned for output of object code.
27	Record number of the source line in the input source file. This number is not incremented for lines included by the $INCLUDE directive. For the included lines, the include nesting level is also produced to the right of the line number.

010-52-2



```
Fortran-VIIO R06-00.00          REAL S, X(3), C(3)                                       04/10/89 11:28:37 PAGE 1/1
* OPTIMIZER:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
  LINE  LVL      +      START ,NOP
     1     !      REAL S, X(3), C(3)                                          !                1
     2     !      INTEGER I(2)                                                !                2
     3     !      DATA X(1), X(2), X(3), I(1), I(2)/3.0, 2.0, 1.0, 0,1/       !                3
     4    0!      IF (I(1) .EQ. 1 .AND. I(2) .EQ. 0) THEN                     !                4
     5    1!            S = X(1)**2 * (-X(2))**2 * (-X(3))**2                 !                5
     6    0!      ELSE                                                        !                6
     7    1!            IF (I(1) .EQ.1 .AND. I(2) .EQ. 1) THEN                !                7
     8    2!                  S = X(1)**2 * (-X(2))**2 + X(3)**2              !                8
     9    1!            ELSE                                                  !                9
    10    2!                  IF (I(1) .EQ. 0 .AND. I(2) .EQ. 0) THEN         !               10
    11    3!                        S = X(1)**2 + X(2)**2 + (-X(3))**2        !               11
    12    2!                  ELSE                                            !               12
    13    3!                        S = X(1)**2 + X(2)**2 * X(3)**2           !               13
    14    2!                  ENDIF                                           !               14
    15    1!            ENDIF                                                 !               15
    16    0!      ENDIF                                                       !               16
    17     !      DO 30 M=1,3                                                 !               17
    18     !          C(M) = X(M)/SQRT(S)                                     !               18
    19     !30    CONTINUE                                                    !               19
    20     !      WRITE(*,35) C(1), C(2), C(3)                                !               20
    21     !      STOP                                                        !               21
    22     !35    FORMAT('0',T5,F15.8,T18,F15.8,T33,F15.8)                    !               22
    23     !      END                                                         !               23

LENGTH OF IMPURE: 00000028       40
LENGTH OF PURE:   00000178      376
TOTAL SIZE:       000001A0      416

.MAIN     COMPILED ON  MONDAY, APRIL 10, 1989

NALST     NAPU      NBABORT   NBASE     BATCH     NCAL      NCOMP     CONT=19   NHOLL     LIST      NOPTIMIZE SEG       NSYNTAX
TARG=3230 NTEST     NTRACE    NXREF     WARN      NUNNORMALIZE

73.50K UNUSED OUT OF 100.00K.    TABLE SPACE:   26.75K   DISC SECTORS:   0

COMPILER FILE: M300:F7055.TSK/P        SOURCE LISTING: 3,M300:REAL.LST/P
INPUT FILE:  1,M300:REAL.FTN/P         OBJECT FILE:    2,M300:REAL.OBJ/P
```

Figure 10-2. Example of F7O and F7Z Source Listing Without Compilation Errors

Cross-Reference Listings

If the XREF directive is specified, the FORTRAN compiler will generate a cross-reference listing of the source code. This listing can help determine:

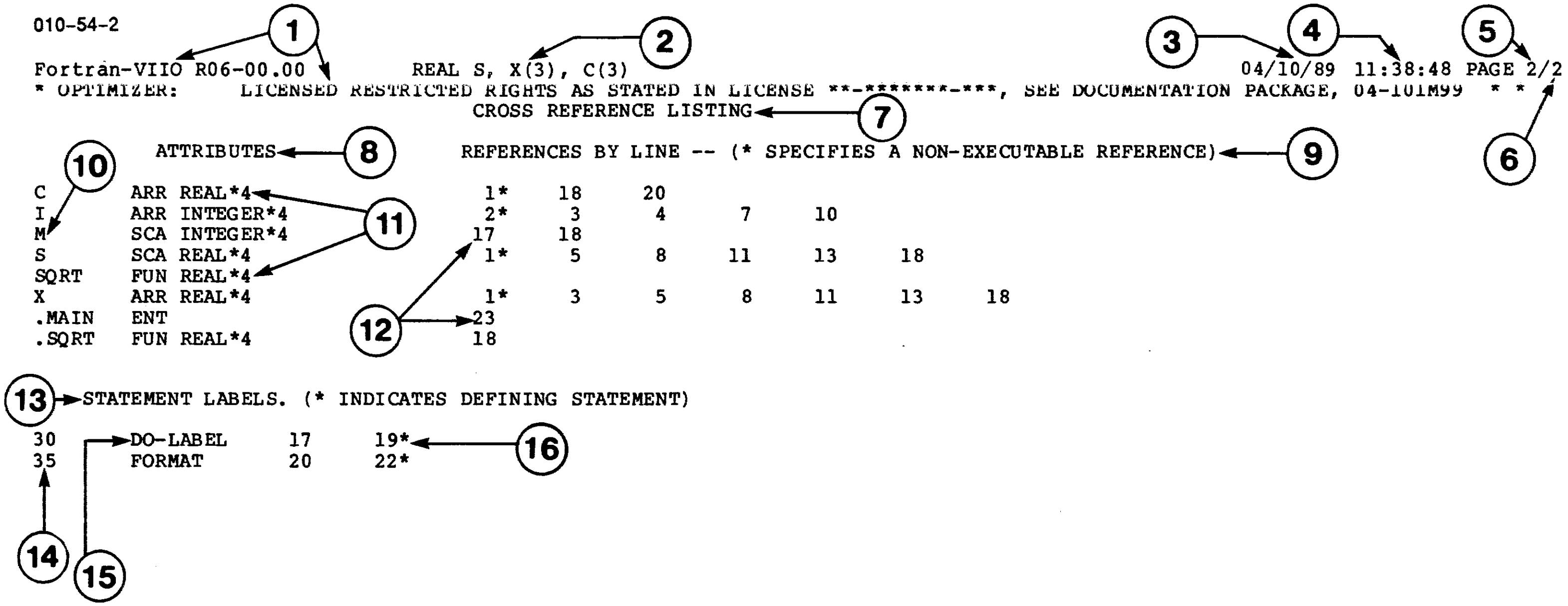
- If a variable or array was defined by a specification statement and not referred to after its definition.
- The line numbers where labels and subprogram entry points are defined and referenced. (If a label is not referenced, only its defining statement will appear in the cross-reference listing.)
- The line numbers where a variable or array are referenced.
- If a variable or array is used before its value is set.
- If a variable or array value is set and never used.

The last two features may not prove to be of significant value when desk checking a program whose flow of control is complex. In this instance, a more careful inspection of the source code is required.

An example of the cross-reference listing format for a program compiled without errors under F7O or F7Z is shown in Figure 10-3. Numbered items in this figure are identified as follows.

Number	List Item
1	FORTRAN compiler identification including compiler name, release and revision number, license, and documentation package numbers.
2	Title of the program. If no title is specified through the TITLE directive, the compiler prints the first line of source code.
3	Date compilation was performed.
4	Time compilation began.
5	Page number of listing.
6	Total number of pages for this listing.

Number	List Item
7	Title of listing.
8	Header for the attributes of program variables and procedures that follow.
9	Header for the statement line numbers of the referenced program variables and procedures contained in the program.
10	FORTRAN program variable or name of program unit.
11	Attributes of variable or procedure including data type, word size, and type of subprogram.
12	Line numbers of statements that reference the variables or procedures. Starred line numbers indicate that the statement declares the variable or procedure. See the source listing in Figure 10-3.
13	Header indicating that cross-reference information on statement labels follows.
14	Statement label.
15	Type of statement in which a statement label is used (e.g., DO, FORMAT, etc.).
16	Line numbers of statements that reference the statement label. Starred line numbers indicate that the statement defines the label. (See Figure 10-3.)



```
Fortran-VIIO R06-00.00        REAL S, X(3), C(3)                                              04/10/89 11:38:48 PAGE 2/2
* OPTIMIZER:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
                                    CROSS REFERENCE LISTING

         ATTRIBUTES               REFERENCES BY LINE -- (* SPECIFIES A NON-EXECUTABLE REFERENCE)

C        ARR REAL*4                 1*    18     20
I        ARR INTEGER*4              2*     3      4      7     10
M        SCA INTEGER*4             17     18
S        SCA REAL*4                 1*     5      8     11     13     18
SQRT     FUN REAL*4
X        ARR REAL*4                 1*     3      5      8     11     13     18
.MAIN    ENT                       23
.SQRT    FUN REAL*4                18

STATEMENT LABELS. (* INDICATES DEFINING STATEMENT)
30       DO-LABEL      17      19*
35       FORMAT        20      22*
```

Figure 10-3. Example of Cross-Reference Listing

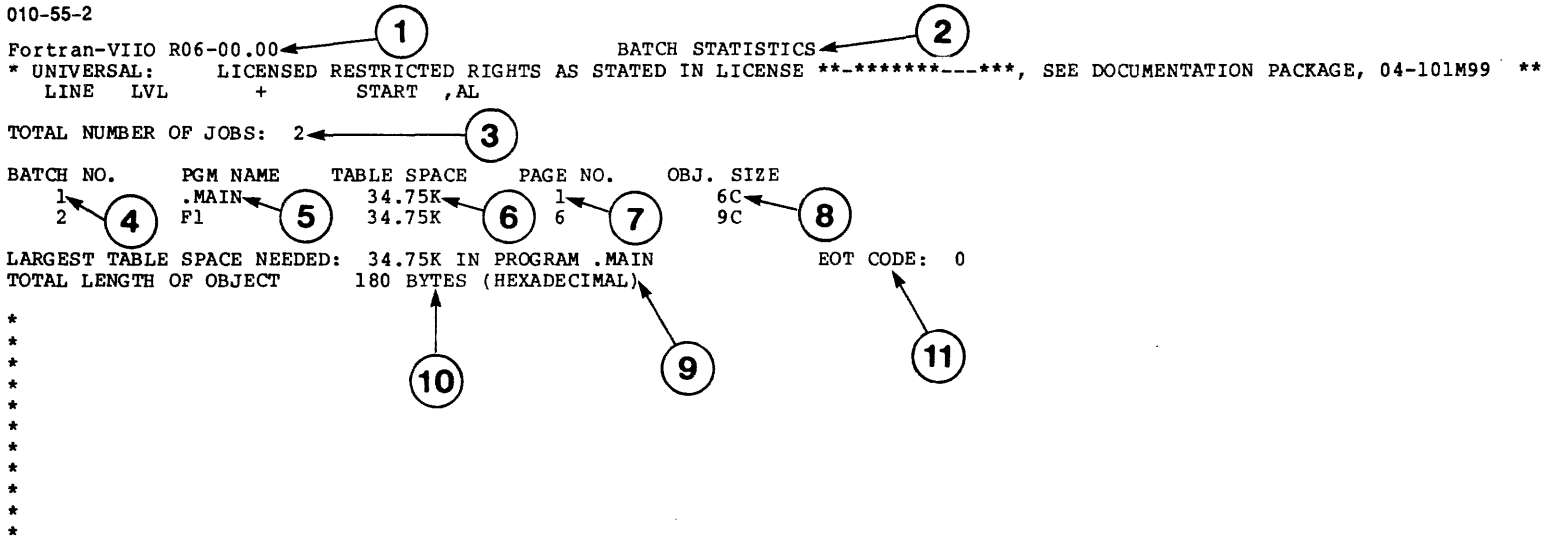
Batch Statistics

When more than one program unit is compiled in batch, a listing of the batch statistics is sent to the list device. The format of this listing is shown in Figure 10-4. Numbered items in Figure 10-4 are identified as follows.

Number	List Item
1	FORTRAN compiler identification.
2	Title of the listing.
3	Total number of program units compiled.
4	Sequence number of each program unit in the batch compilation.
5	Name of the program unit.
6	Memory required for internal tables to compile the program unit.
7	Number of the first page of the source listing for the specified program unit.
8	Hexadecimal number indicating the size in bytes of the object code generated.
9	Name of the program unit that required the largest memory space for internal tables.
10	Hexadecimal number indicating the total length of the object code generated for the entire batch compilation.
11	End of task code with which the batch compilation terminated.

In some instances, additional information may also be displayed on the listing of the batch statistics as follows:

• CAL	A Common Assembly Language (CAL) file was created instead of object code.
• ERRORS	Errors were detected during compilation.
• COMPILATION	Compilation of a program was skipped either due to a $SYNTAX or due to a $INSKIP directive.



```
010-55-2

Fortran-VIIO R06-00.00                          BATCH STATISTICS
* UNIVERSAL:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******---***, SEE DOCUMENTATION PACKAGE, 04-101M99   **
   LINE   LVL       +       START   ,AL

TOTAL NUMBER OF JOBS:  2

BATCH NO.     PGM NAME    TABLE SPACE    PAGE NO.    OBJ. SIZE
    1         .MAIN          34.75K         1           6C
    2         F1             34.75K         6           9C

LARGEST TABLE SPACE NEEDED:  34.75K IN PROGRAM .MAIN              EOT CODE:  0
TOTAL LENGTH OF OBJECT      180 BYTES (HEXADECIMAL)

*
*
*
*
*
*
*
*
*
*
*
```

Figure 10-4. Example of Batch Statistics

Link Maps

The Link map tells the programmer how the task is structured and where each subprogram or run-time library (RTL) routine is referenced by the program. The map can be used to determine whether a user-defined or Concurrent standard library routine was referenced incorrectly or redefined by the program.

Each Link map begins with a task establishment summary as shown in Figure 10-5. Numbered items contained in this summary are identified as follows.

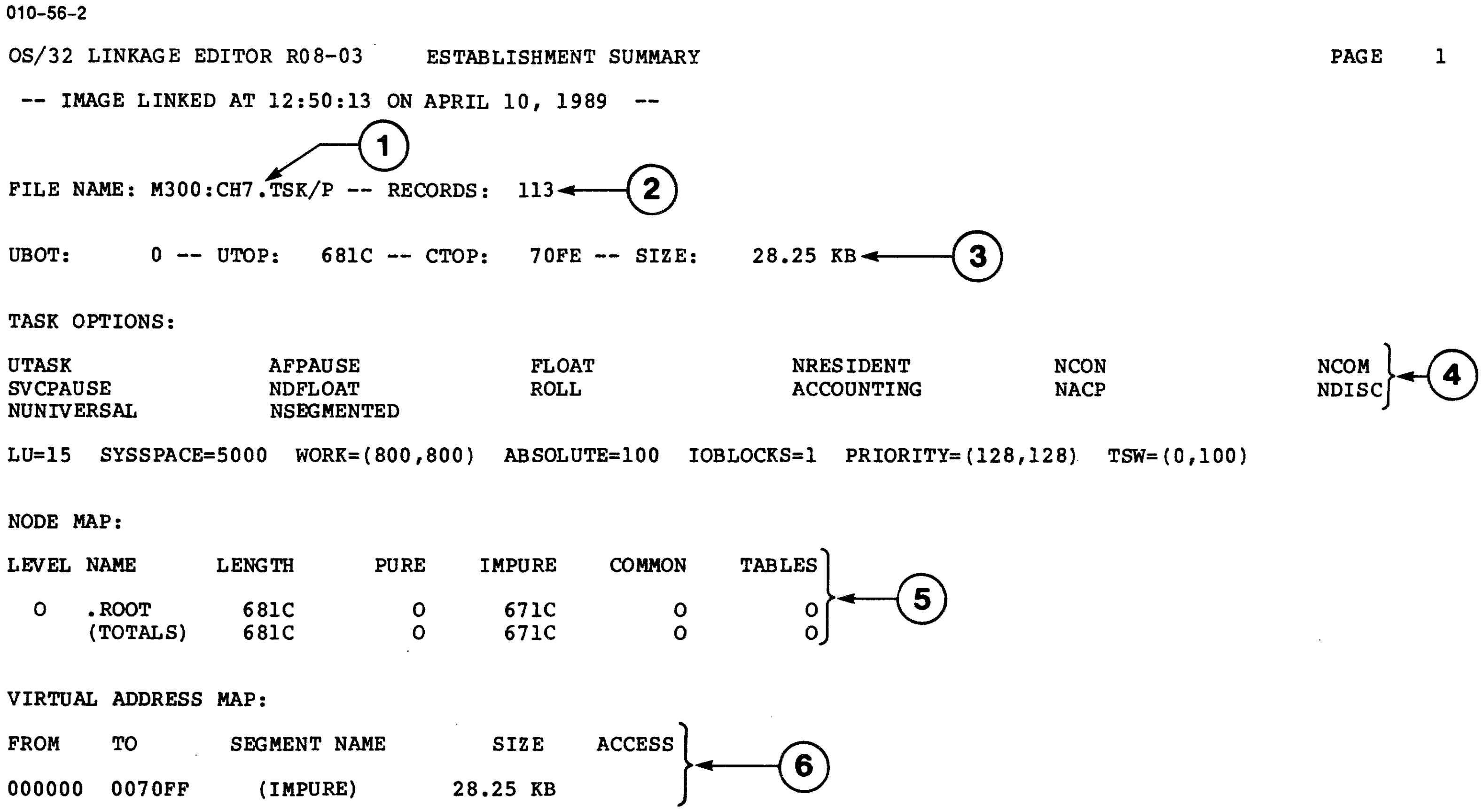
Number	List Item
1	File descriptor (fd) of task image file.
2	Number of records in task image file.
3	Task size and address space.
4	Task options set by Link OPTION command or by Link default.
5	Length of each segment within the address space.
6	Address and size of pure and impure task segments.

Following the task establishment summary are the symbol maps which list all subprograms and RTL routines called by the program. Symbols can be arranged alphabetically, in order of their entry point names, or according to how they are referenced by the program. How the map is arranged depends on the options specified in the Link MAP command.

Figure 10-6 is an address map which can be used to trace the use of any subprogram or RTL routine in the sample program. From this map it can be seen that the code for the standard SQRT routine starts at location 7FC. The program calling SQRT branches to the entry point 804.

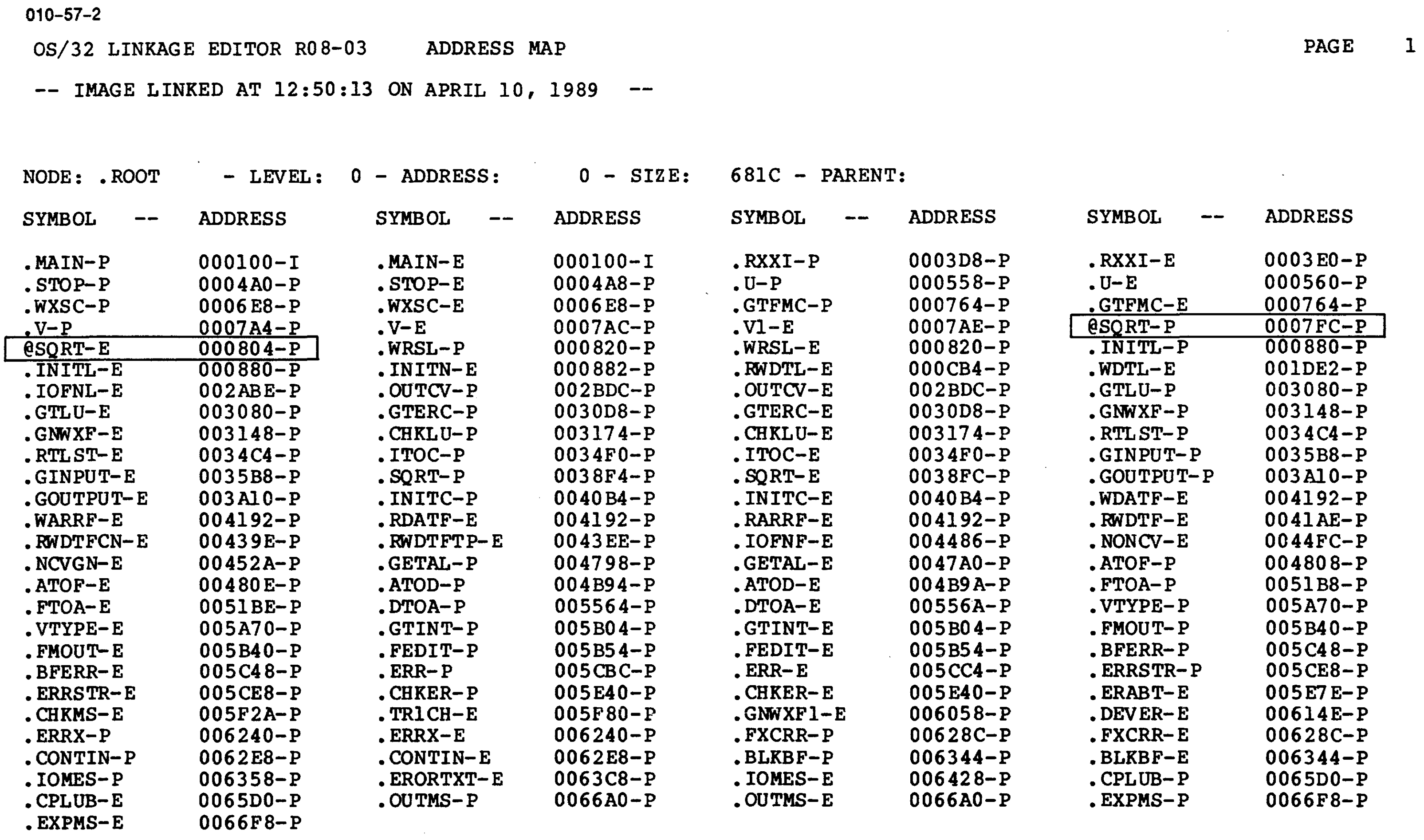
The alphabetic map in Figure 10-7 provides the same information as the address map except that all routines are arranged alphabetically rather than by their entry point addresses.

Reading the cross-reference map, the user can trace where a subprogram or an RTL routine is referenced. For example, Figure 10-8 shows that @SQRT is referenced in .MAIN and @SQRT references .SQRT which, in turn, references .ERR.



```
010-56-2

OS/32 LINKAGE EDITOR R08-03     ESTABLISHMENT SUMMARY                                          PAGE    1

 -- IMAGE LINKED AT 12:50:13 ON APRIL 10, 1989  --

FILE NAME: M300:CH7.TSK/P -- RECORDS:  113

UBOT:      0 -- UTOP:   681C -- CTOP:   70FE -- SIZE:    28.25 KB

TASK OPTIONS:

UTASK               AFPAUSE             FLOAT               NRESIDENT           NCON                NCOM
SVCPAUSE            NDFLOAT             ROLL                ACCOUNTING          NACP                NDISC
NUNIVERSAL          NSEGMENTED

LU=15  SYSSPACE=5000  WORK=(800,800)  ABSOLUTE=100  IOBLOCKS=1  PRIORITY=(128,128)  TSW=(0,100)

NODE MAP:

LEVEL NAME       LENGTH       PURE     IMPURE     COMMON     TABLES

  0   .ROOT        681C          0       671C          0          0
      (TOTALS)     681C          0       671C          0          0

VIRTUAL ADDRESS MAP:

FROM     TO        SEGMENT NAME        SIZE    ACCESS

000000   0070FF     (IMPURE)        28.25 KB
```

Figure 10-5. Link Establishment Summary

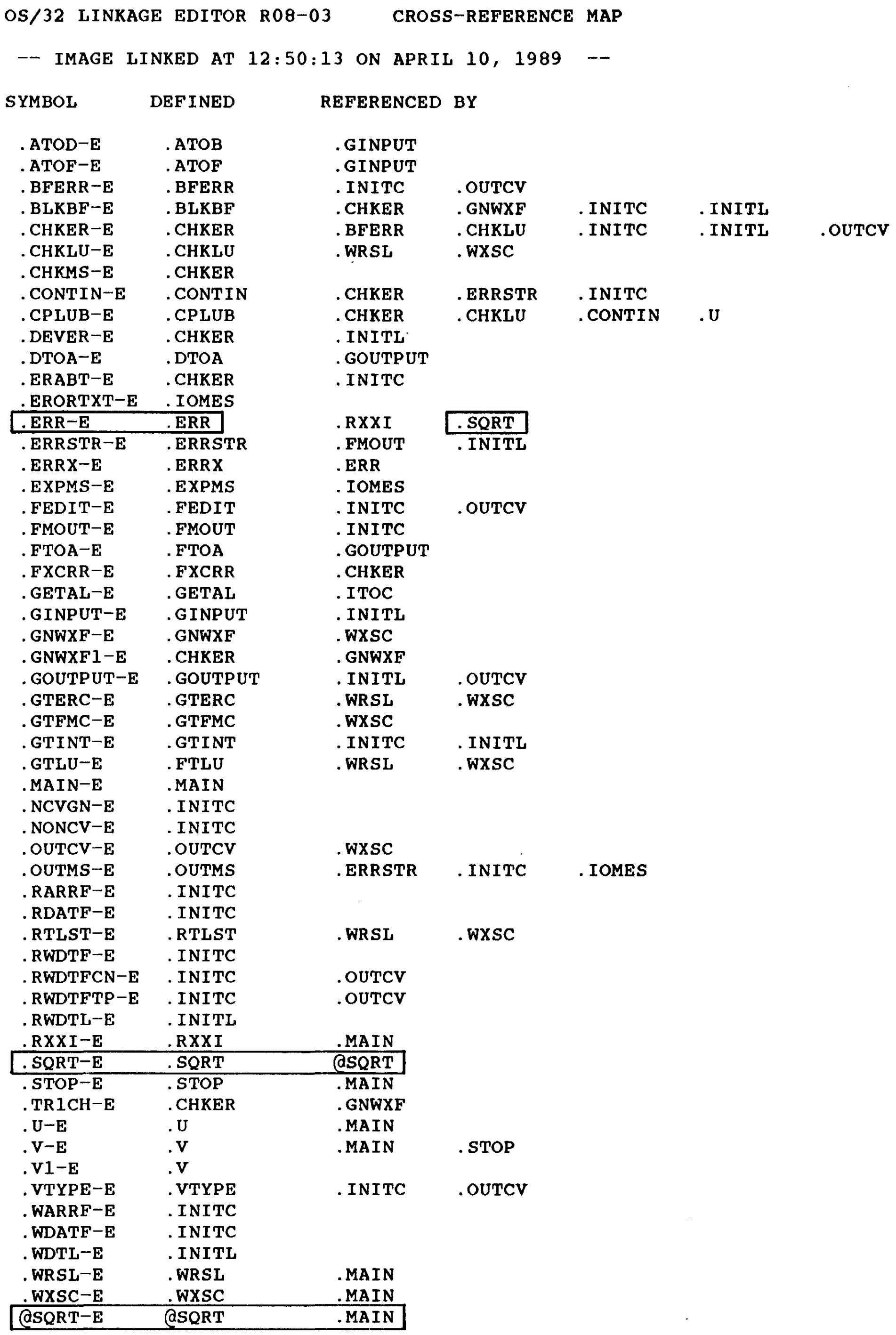


```
010-57-2
 OS/32 LINKAGE EDITOR R08-03     ADDRESS MAP                                                     PAGE    1
 -- IMAGE LINKED AT 12:50:13 ON APRIL 10, 1989   --

NODE: .ROOT      - LEVEL:  0 - ADDRESS:        0 - SIZE:    681C - PARENT:

SYMBOL    --     ADDRESS        SYMBOL    --     ADDRESS        SYMBOL    --     ADDRESS        SYMBOL    --     ADDRESS

.MAIN-P          000100-I       .MAIN-E          000100-I       .RXXI-P          0003D8-P       .RXXI-E          0003E0-P
.STOP-P          0004A0-P       .STOP-E          0004A8-P       .U-P             000558-P       .U-E             000560-P
.WXSC-P          0006E8-P       .WXSC-E          0006E8-P       .GTFMC-P         000764-P       .GTFMC-E         000764-P
.V-P             0007A4-P       .V-E             0007AC-P       .V1-E            0007AE-P       @SQRT-P          0007FC-P
@SQRT-E          000804-P       .WRSL-P          000820-P       .WRSL-E          000820-P       .INITL-P         000880-P
.INITL-E         000880-P       .INITN-E         000882-P       .RWDTL-E         000CB4-P       .WDTL-E          001DE2-P
.IOFNL-E         002ABE-P       .OUTCV-P         002BDC-P       .OUTCV-E         002BDC-P       .GTLU-P          003080-P
.GTLU-E          003080-P       .GTERC-P         0030D8-P       .GTERC-E         0030D8-P       .GNWXF-P         003148-P
.GNWXF-E         003148-P       .CHKLU-P         003174-P       .CHKLU-E         003174-P       .RTLST-P         0034C4-P
.RTLST-E         0034C4-P       .ITOC-P          0034F0-P       .ITOC-E          0034F0-P       .GINPUT-P        0035B8-P
.GINPUT-E        0035B8-P       .SQRT-P          0038F4-P       .SQRT-E          0038FC-P       .GOUTPUT-P       003A10-P
.GOUTPUT-E       003A10-P       .INITC-P         0040B4-P       .INITC-E         0040B4-P       .WDATF-E         004192-P
.WARRF-E         004192-P       .RDATF-E         004192-P       .RARRF-E         004192-P       .RWDTF-E         0041AE-P
.RWDTFCN-E       00439E-P       .RWDTFTP-E       0043EE-P       .IOFNF-E         004486-P       .NONCV-E         0044FC-P
.NCVGN-E         00452A-P       .GETAL-P         004798-P       .GETAL-E         0047A0-P       .ATOF-P          004808-P
.ATOF-E          00480E-P       .ATOD-P          004B94-P       .ATOD-E          004B9A-P       .FTOA-P          0051B8-P
.FTOA-E          0051BE-P       .DTOA-P          005564-P       .DTOA-E          00556A-P       .VTYPE-P         005A70-P
.VTYPE-E         005A70-P       .GTINT-P         005B04-P       .GTINT-E         005B04-P       .FMOUT-P         005B40-P
.FMOUT-E         005B40-P       .FEDIT-P         005B54-P       .FEDIT-E         005B54-P       .BFERR-P         005C48-P
.BFERR-E         005C48-P       .ERR-P           005CBC-P       .ERR-E           005CC4-P       .ERRSTR-P        005CE8-P
.ERRSTR-E        005CE8-P       .CHKER-P         005E40-P       .CHKER-E         005E40-P       .ERABT-E         005E7E-P
.CHKMS-E         005F2A-P       .TR1CH-E         005F80-P       .GNWXF1-E        006058-P       .DEVER-E         00614E-P
.ERRX-P          006240-P       .ERRX-E          006240-P       .FXCRR-P         00628C-P       .FXCRR-E         00628C-P
.CONTIN-P        0062E8-P       .CONTIN-E        0062E8-P       .BLKBF-P         006344-P       .BLKBF-E         006344-P
.IOMES-P         006358-P       .ERORTXT-E       0063C8-P       .IOMES-E         006428-P       .CPLUB-P         0065D0-P
.CPLUB-E         0065D0-P       .OUTMS-P         0066A0-P       .OUTMS-E         0066A0-P       .EXPMS-P         0066F8-P
.EXPMS-E         0066F8-P
```

Figure 10-6. Link Address Map

```
010-58-2

OS/32 LINKAGE EDITOR R08-03     ALPHABETIC MAP                                                      PAGE    1

 -- IMAGE LINKED AT 12:50:13 ON APRIL 10, 1989  --


SYMBOL      -- NODE      -- ADDRESS       SYMBOL      -- NODE      -- ADDRESS       SYMBOL      -- NODE      -- ADDRESS

.ATOD-E        .ROOT        004B9A-P      .ATOD-P        .ROOT        004B94-P      .ATOF-E        .ROOT        00480E-P
.ATOF-P        .ROOT        004808-P      .BFERR-E       .ROOT        005C48-P      .BFERR-P       .ROOT        005C48-P
.BLKBF-E       .ROOT        006344-P      .BLKBF-P       .ROOT        006344-P      .CHKER-E       .ROOT        005E40-P
.CHKER-P       .ROOT        005E40-P      .CHKLU-E       .ROOT        003174-P      .CHKLU-P       .ROOT        003174-P
.CHKMS-E       .ROOT        005F2A-P      .CONTIN-P      .ROOT        0062E8-P      .CONTIN-E      .ROOT        0062E8-P
.CPLUB-P       .ROOT        0065D0-P      .CPLUB-E       .ROOT        0065D0-P      .DEVER-E       .ROOT        00614E-P
.DTOA-E        .ROOT        00556A-P      .DTOA-P        .ROOT        005564-P      .ERABT-E       .ROOT        005E7E-P
.ERORTXT-E     .ROOT        0063C8-P      .ERR-E         .ROOT        005CC4-P      .ERR-P         .ROOT        005CBC-P
.ERRSTR-P      .ROOT        005CE8-P      .ERRSTR-E      .ROOT        005CE8-P      .ERRX-P        .ROOT        006240-P
.ERRX-E        .ROOT        006240-P      .EXPMS-E       .ROOT        0066F8-P      .EXPMS-P       .ROOT        0066F8-P
.FEDIT-E       .ROOT        005B54-P      .FEDIT-P       .ROOT        005B54-P      .FMOUT-E       .ROOT        005B40-P
.FMOUT-P       .ROOT        005B40-P      .FTOA-P        .ROOT        0051B8-P      .FTOA-E        .ROOT        0051BE-P
.FXCRR-E       .ROOT        00628C-P      .FXCRR-P       .ROOT        00628C-P      .GETAL-P       .ROOT        004798-P
.GETAL-E       .ROOT        0047A0-P      .GINPUT-P      .ROOT        0035B8-P      .GINPUT-E      .ROOT        0035B8-P
.GNWXF-P       .ROOT        003148-P      .GNWXF-E       .ROOT        003148-P      .GNWXF1-E      .ROOT        006058-P
.GOUTPUT-P     .ROOT        003A10-P      .GOUTPUT-E     .ROOT        003A10-P      .GTERC-P       .ROOT        0030D8-P
.GTERC-E       .ROOT        0030D8-P      .GTFMC-P       .ROOT        000764-P      .GTFMC-E       .ROOT        000764-P
.GTINT-P       .ROOT        005B04-P      .GTINT-E       .ROOT        005B04-P      .GTLU-P        .ROOT        003080-P
.GTLU-E        .ROOT        003080-P      .INITC-E       .ROOT        0040B4-P      .INITC-P       .ROOT        0040B4-P
.INITL-E       .ROOT        000880-P      .INITL-P       .ROOT        000880-P      .INITN-E       .ROOT        000882-P
.IOFNF-E       .ROOT        004486-P      .IOFNL-E       .ROOT        002ABE-P      .IOMES-P       .ROOT        006358-P
.IOMES-E       .ROOT        006428-P      .ITOC-P        .ROOT        0034F0-P      .ITOC-E        .ROOT        0034F0-P
.MAIN-P        .ROOT        000100-I      .MAIN-E        .ROOT        000100-I      .NCVGN-E       .ROOT        00452A-P
.NONCV-E       .ROOT        0044FC-P      .OUTCV-P       .ROOT        002BDC-P      .OUTCV-E       .ROOT        002BDC-P
.OUTMS-E       .ROOT        0066A0-P      .OUTMS-P       .ROOT        0066A0-P      .RARRF-E       .ROOT        004192-P
.RDATF-E       .ROOT        004192-P      .RTLST-P       .ROOT        0034C4-P      .RTLST-E       .ROOT        0034C4-P
.RWDTF-E       .ROOT        0041AE-P      .RWDTFCN-E     .ROOT        00439E-P      .RWDTFTP-E     .ROOT        0043EE-P
.RWDTL-E       .ROOT        000CB4-P      .RXXI-E        .ROOT        0003E0-P      .RXXI-P        .ROOT        0003D8-P
.SQRT-E        .ROOT        0038FC-P      .SQRT-P        .ROOT        0038F4-P      .STOP-E        .ROOT        0004A8-P
.STOP-P        .ROOT        0004A0-P      .TR1CH-E       .ROOT        005F80-P      .U-E           .ROOT        000560-P
.U-P           .ROOT        000558-P      .V-E           .ROOT        0007AC-P      .V-P           .ROOT        0007A4-P
.V1-E          .ROOT        0007AE-P      .VTYPE-P       .ROOT        005A70-P      .VTYPE-E       .ROOT        005A70-P
.WARRF-E       .ROOT        004192-P      .WDATF-E       .ROOT        004192-P      .WDTL-E        .ROOT        001DE2-P
.WRSL-E        .ROOT        000820-P      .WRSL-P        .ROOT        000820-P      .WXSC-E        .ROOT        0006E8-P
.WXSC-P        .ROOT        0006E8-P      @SQRT-E        .ROOT        000804-P      @SQRT-P        .ROOT        0007FC-P
```

Figure 10-7. Link Alphabetic Map



```
OS/32 LINKAGE EDITOR R08-03     CROSS-REFERENCE MAP

 -- IMAGE LINKED AT 12:50:13 ON APRIL 10, 1989  --

SYMBOL       DEFINED       REFERENCED BY

 .ATOD-E      .ATOB         .GINPUT
 .ATOF-E      .ATOF         .GINPUT
 .BFERR-E     .BFERR        .INITC     .OUTCV
 .BLKBF-E     .BLKBF        .CHKER     .GNWXF     .INITC     .INITL
 .CHKER-E     .CHKER        .BFERR     .CHKLU     .INITC     .INITL     .OUTCV
 .CHKLU-E     .CHKLU        .WRSL      .WXSC
 .CHKMS-E     .CHKER
 .CONTIN-E    .CONTIN       .CHKER     .ERRSTR    .INITC
 .CPLUB-E     .CPLUB        .CHKER     .CHKLU     .CONTIN    .U
 .DEVER-E     .CHKER        .INITL
 .DTOA-E      .DTOA         .GOUTPUT
 .ERABT-E     .CHKER        .INITC
 .ERORTXT-E   .IOMES
 .ERR-E       .ERR          .RXXI      .SQRT
 .ERRSTR-E    .ERRSTR       .FMOUT     .INITL
 .ERRX-E      .ERRX         .ERR
 .EXPMS-E     .EXPMS        .IOMES
 .FEDIT-E     .FEDIT        .INITC     .OUTCV
 .FMOUT-E     .FMOUT        .INITC
 .FTOA-E      .FTOA         .GOUTPUT
 .FXCRR-E     .FXCRR        .CHKER
 .GETAL-E     .GETAL        .ITOC
 .GINPUT-E    .GINPUT       .INITL
 .GNWXF-E     .GNWXF        .WXSC
 .GNWXF1-E    .CHKER        .GNWXF
 .GOUTPUT-E   .GOUTPUT      .INITL     .OUTCV
 .GTERC-E     .GTERC        .WRSL      .WXSC
 .GTFMC-E     .GTFMC        .WXSC
 .GTINT-E     .GTINT        .INITC     .INITL
 .GTLU-E      .FTLU         .WRSL      .WXSC
 .MAIN-E      .MAIN
 .NCVGN-E     .INITC
 .NONCV-E     .INITC
 .OUTCV-E     .OUTCV        .WXSC
 .OUTMS-E     .OUTMS        .ERRSTR    .INITC     .IOMES
 .RARRF-E     .INITC
 .RDATF-E     .INITC
 .RTLST-E     .RTLST        .WRSL      .WXSC
 .RWDTF-E     .INITC
 .RWDTFCN-E   .INITC        .OUTCV
 .RWDTFTP-E   .INITC        .OUTCV
 .RWDTL-E     .INITL
 .RXXI-E      .RXXI         .MAIN
 .SQRT-E      .SQRT         @SQRT
 .STOP-E      .STOP         .MAIN
 .TR1CH-E     .CHKER        .GNWXF
 .U-E         .U            .MAIN
 .V-E         .V            .MAIN      .STOP
 .V1-E        .V
 .VTYPE-E     .VTYPE        .INITC     .OUTCV
 .WARRF-E     .INITC
 .WDATF-E     .INITC
 .WDTL-E      .INITL
 .WRSL-E      .WRSL         .MAIN
 .WXSC-E      .WXSC         .MAIN
 @SQRT-E      @SQRT         .MAIN
```

Figure 10-8. Link Cross-Reference Map

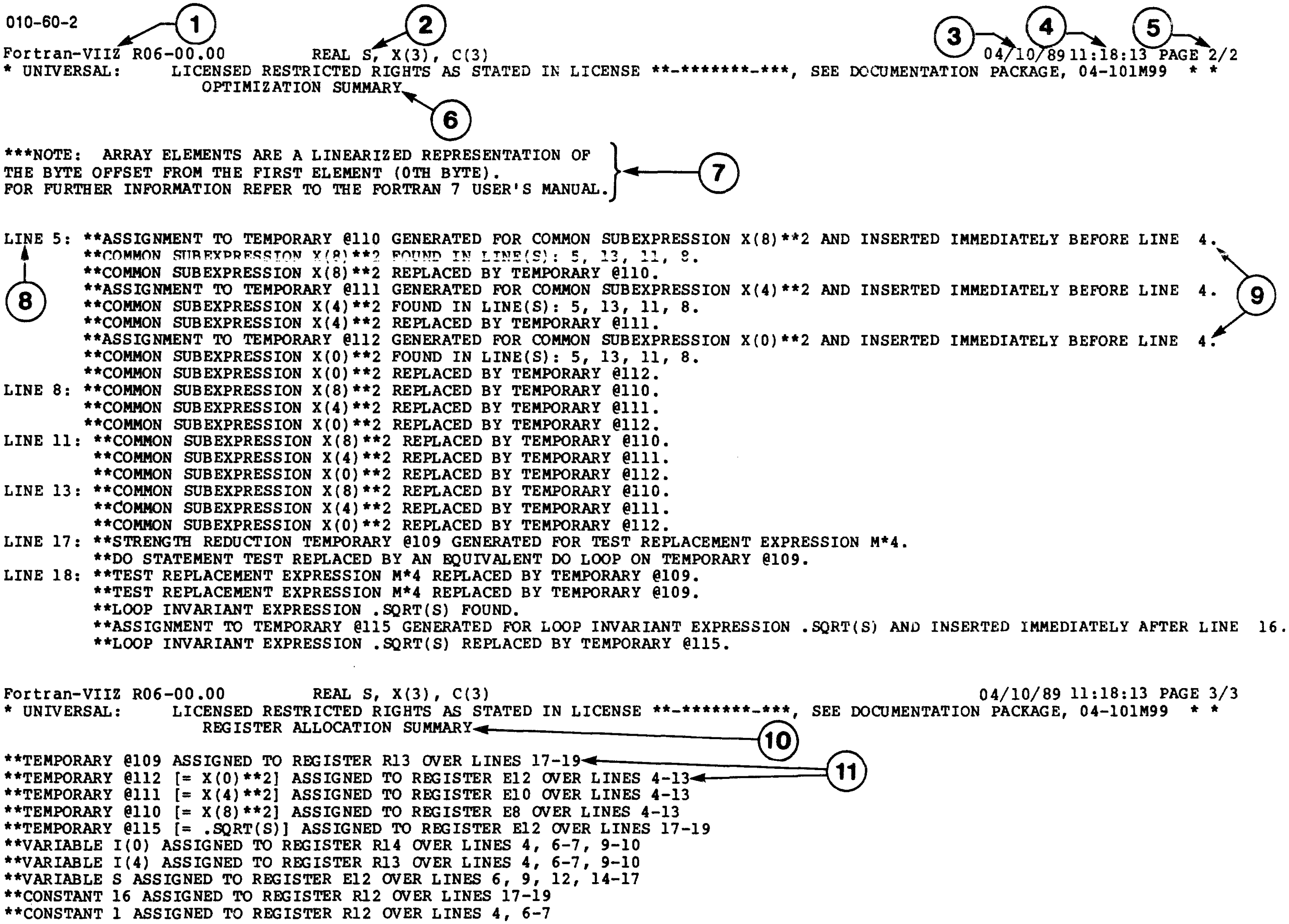
Optimization Summaries

The source and cross-reference listings are important aids in determining how the compiler translated the source code. However, more information is needed when debugging optimized code. Compiler optimization summaries provide this information.

The optimizing compilers automatically follow each source listing with an optimization and register allocation summary. This summary tells the programmer what optimizations were performed on each line of source code. From this information, the programmer can determine what code was eliminated, moved, or replaced. For example, the optimization summary in Figure 10-9 for the sample program introduced in Figure 10-2 shows that common subexpression replacement (i.e., replacement of expressions with compiler-generated temporary variables was performed on lines 5, 8, 11, and 13.) The register allocation summary shows the allocation of registers for specific entities within the program. For example, the register allocation summary in Figure 10-11 shows that I(O) was assigned to register 14 in lines 4, 6, 7, 9, and 10. Numbered items in Figure 10-9 are identified as follows.

Number	**List Item**
1	FORTRAN compiler identification including compiler name, release and revision number, license number and documentation package number.
2	Title of the program. If no title is specified through the TITLE directive, the compiler prints the first line of source code.
3	Date compilation was performed.
4	Time compilation began.
5	Page number of listing.
6	Header for optimization summary.
7	Array linearization message. This message informs the user that all array subscripts were replaced by the actual byte address of the array element. (See Chapter 3)
8	Line number of an optimized source statement.

Number	List Item
9	Optimizations performed on the specified line.
10	Header for register allocation summary.
11	Allocated registers and the line numbers of the statements on which register allocation was performed.



```
010-60-2
Fortran-VIIZ R06-00.00        REAL S, X(3), C(3)                                         04/10/89 11:18:13 PAGE 2/2
* UNIVERSAL:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
                    OPTIMIZATION SUMMARY

***NOTE:  ARRAY ELEMENTS ARE A LINEARIZED REPRESENTATION OF
THE BYTE OFFSET FROM THE FIRST ELEMENT (0TH BYTE).
FOR FURTHER INFORMATION REFER TO THE FORTRAN 7 USER'S MANUAL.

LINE 5: **ASSIGNMENT TO TEMPORARY @110 GENERATED FOR COMMON SUBEXPRESSION X(8)**2 AND INSERTED IMMEDIATELY BEFORE LINE  4.
        **COMMON SUBEXPRESSION X(8)**2 FOUND IN LINE(S): 5, 13, 11, 8.
        **COMMON SUBEXPRESSION X(8)**2 REPLACED BY TEMPORARY @110.
        **ASSIGNMENT TO TEMPORARY @111 GENERATED FOR COMMON SUBEXPRESSION X(4)**2 AND INSERTED IMMEDIATELY BEFORE LINE  4.
        **COMMON SUBEXPRESSION X(4)**2 FOUND IN LINE(S): 5, 13, 11, 8.
        **COMMON SUBEXPRESSION X(4)**2 REPLACED BY TEMPORARY @111.
        **ASSIGNMENT TO TEMPORARY @112 GENERATED FOR COMMON SUBEXPRESSION X(0)**2 AND INSERTED IMMEDIATELY BEFORE LINE  4.
        **COMMON SUBEXPRESSION X(0)**2 FOUND IN LINE(S): 5, 13, 11, 8.
        **COMMON SUBEXPRESSION X(0)**2 REPLACED BY TEMPORARY @112.
LINE 8: **COMMON SUBEXPRESSION X(8)**2 REPLACED BY TEMPORARY @110.
        **COMMON SUBEXPRESSION X(4)**2 REPLACED BY TEMPORARY @111.
        **COMMON SUBEXPRESSION X(0)**2 REPLACED BY TEMPORARY @112.
LINE 11: **COMMON SUBEXPRESSION X(8)**2 REPLACED BY TEMPORARY @110.
         **COMMON SUBEXPRESSION X(4)**2 REPLACED BY TEMPORARY @111.
         **COMMON SUBEXPRESSION X(0)**2 REPLACED BY TEMPORARY @112.
LINE 13: **COMMON SUBEXPRESSION X(8)**2 REPLACED BY TEMPORARY @110.
         **COMMON SUBEXPRESSION X(4)**2 REPLACED BY TEMPORARY @111.
         **COMMON SUBEXPRESSION X(0)**2 REPLACED BY TEMPORARY @112.
LINE 17: **STRENGTH REDUCTION TEMPORARY @109 GENERATED FOR TEST REPLACEMENT EXPRESSION M*4.
         **DO STATEMENT TEST REPLACED BY AN EQUIVALENT DO LOOP ON TEMPORARY @109.
LINE 18: **TEST REPLACEMENT EXPRESSION M*4 REPLACED BY TEMPORARY @109.
         **TEST REPLACEMENT EXPRESSION M*4 REPLACED BY TEMPORARY @109.
         **LOOP INVARIANT EXPRESSION .SQRT(S) FOUND.
         **ASSIGNMENT TO TEMPORARY @115 GENERATED FOR LOOP INVARIANT EXPRESSION .SQRT(S) AND INSERTED IMMEDIATELY AFTER LINE  16.
         **LOOP INVARIANT EXPRESSION .SQRT(S) REPLACED BY TEMPORARY @115.

Fortran-VIIZ R06-00.00        REAL S, X(3), C(3)                                         04/10/89 11:18:13 PAGE 3/3
* UNIVERSAL:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
                    REGISTER ALLOCATION SUMMARY

**TEMPORARY @109 ASSIGNED TO REGISTER R13 OVER LINES 17-19
**TEMPORARY @112 [= X(0)**2] ASSIGNED TO REGISTER E12 OVER LINES 4-13
**TEMPORARY @111 [= X(4)**2] ASSIGNED TO REGISTER E10 OVER LINES 4-13
**TEMPORARY @110 [= X(8)**2] ASSIGNED TO REGISTER E8 OVER LINES 4-13
**TEMPORARY @115 [= .SQRT(S)] ASSIGNED TO REGISTER E12 OVER LINES 17-19
**VARIABLE I(0) ASSIGNED TO REGISTER R14 OVER LINES 4, 6-7, 9-10
**VARIABLE I(4) ASSIGNED TO REGISTER R13 OVER LINES 4, 6-7, 9-10
**VARIABLE S ASSIGNED TO REGISTER E12 OVER LINES 6, 9, 12, 14-17
**CONSTANT 16 ASSIGNED TO REGISTER R12 OVER LINES 17-19
**CONSTANT 1 ASSIGNED TO REGISTER R12 OVER LINES 4, 6-7
```

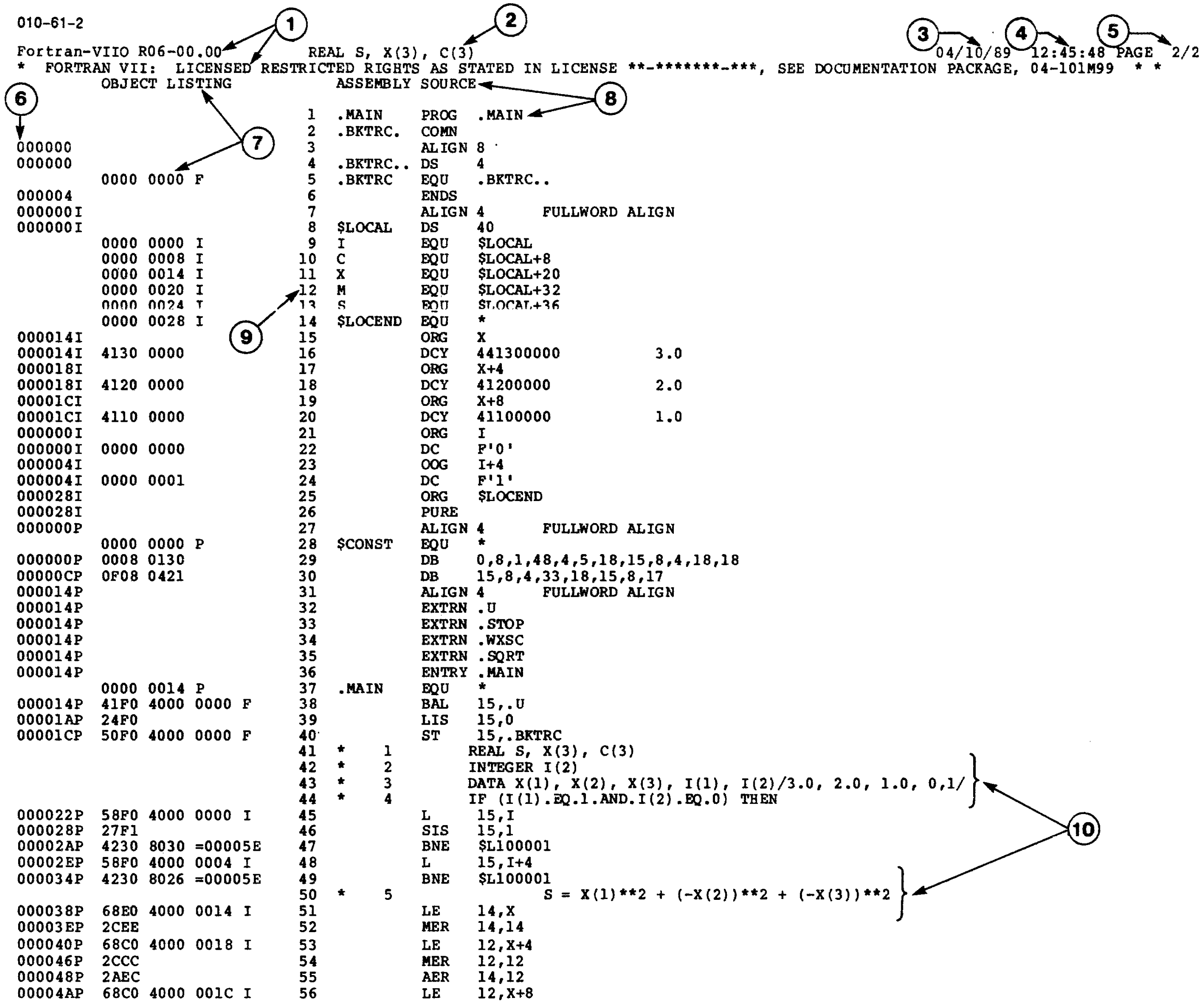
Figure 10-9. Example of Optimization Summary

Assembly Listings

If the ALST directive is specified, the F7O and F7Z compilers will output an assembly listing to the list device after a successful compilation. Using the assembly listing and the Link map described in the section entitled "Link Maps," the user can debug the program with OS/32 Aids.

A portion of the assembly listing for the sample program in Figure 10-2 is shown in Figure 10-10. To aid debugging, the actual lines of FORTRAN source code are inserted before their equivalent lines of assembly code. Numbered items in Figure 10-10 are identified as follows.

Number	**List Item**
1	FORTRAN compiler identification including compiler name, release and revision number, license number, and documentation number.
2	Title of the program. If no title is specified through the TITLE directive, the compiler prints the first line of source code.
3	Date compilation was performed.
4	Time compilation began.
5	Page number of listing.
6	Relative memory address of each line of assembly code.
7	Object code compiled from the source code.
8	Assembly source code of program unit .MAIN.
9	Line number.
10	Actual lines of FORTRAN source code as input to the compiler.



```
010-61-2
Fortran-VIIO R06-00.00            REAL S, X(3), C(3)                                                   04/10/89  12:45:48 PAGE  2/2
*  FORTRAN VII:  LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
         OBJECT LISTING          ASSEMBLY SOURCE

                                  1  .MAIN     PROG  .MAIN
                                  2  .BKTRC.   COMN
000000                            3            ALIGN 8
000000                            4  .BKTRC..  DS    4
         0000 0000 F              5  .BKTRC    EQU   .BKTRC..
000004                            6            ENDS
000000I                           7            ALIGN 4       FULLWORD ALIGN
000000I                           8  $LOCAL    DS    40
         0000 0000 I              9  I         EQU   $LOCAL
         0000 0008 I             10  C         EQU   $LOCAL+8
         0000 0014 I             11  X         EQU   $LOCAL+20
         0000 0020 I             12  M         EQU   $LOCAL+32
         0000 0024 I             13  S         EQU   $LOCAL+36
         0000 0028 I             14  $LOCEND   EQU   *
000014I                          15            ORG   X
000014I  4130 0000               16            DCY   441300000           3.0
000018I                          17            ORG   X+4
000018I  4120 0000               18            DCY   41200000            2.0
00001CI                          19            ORG   X+8
00001CI  4110 0000               20            DCY   41100000            1.0
000000I                          21            ORG   I
000000I  0000 0000               22            DC    F'0'
000004I                          23            OOG   I+4
000004I  0000 0001               24            DC    F'1'
000028I                          25            ORG   $LOCEND
000028I                          26            PURE
000000P                          27            ALIGN 4       FULLWORD ALIGN
         0000 0000 P             28  $CONST    EQU   *
000000P  0008 0130               29            DB    0,8,1,48,4,5,18,15,8,4,18,18
00000CP  0F08 0421               30            DB    15,8,4,33,18,15,8,17
000014P                          31            ALIGN 4       FULLWORD ALIGN
000014P                          32            EXTRN .U
000014P                          33            EXTRN .STOP
000014P                          34            EXTRN .WXSC
000014P                          35            EXTRN .SQRT
000014P                          36            ENTRY .MAIN
         0000 0014 P             37  .MAIN     EQU   *
000014P  41F0 4000 0000 F        38            BAL   15,.U
00001AP  24F0                    39            LIS   15,0
00001CP  50F0 4000 0000 F        40            ST    15,.BKTRC
                                 41  *     1         REAL S, X(3), C(3)
                                 42  *     2         INTEGER I(2)
                                 43  *     3         DATA X(1), X(2), X(3), I(1), I(2)/3.0, 2.0, 1.0, 0,1/
                                 44  *     4         IF (I(1).EQ.1.AND.I(2).EQ.0) THEN
000022P  58F0 4000 0000 I        45            L     15,I
000028P  27F1                    46            SIS   15,1
00002AP  4230 8030 =00005E       47            BNE   $L100001
00002EP  58F0 4000 0004 I        48            L     15,I+4
000034P  4230 8026 =00005E       49            BNE   $L100001
                                 50  *     5                S = X(1)**2 + (-X(2))**2 + (-X(3))**2
000038P  68E0 4000 0014 I        51            LE    14,X
00003EP  2CEE                    52            MER   14,14
000040P  68C0 4000 0018 I        53            LE    12,X+4
000046P  2CCC                    54            MER   12,12
000048P  2AEC                    55            AER   14,12
00004AP  68C0 4000 001C I        56            LE    12,X+8
```

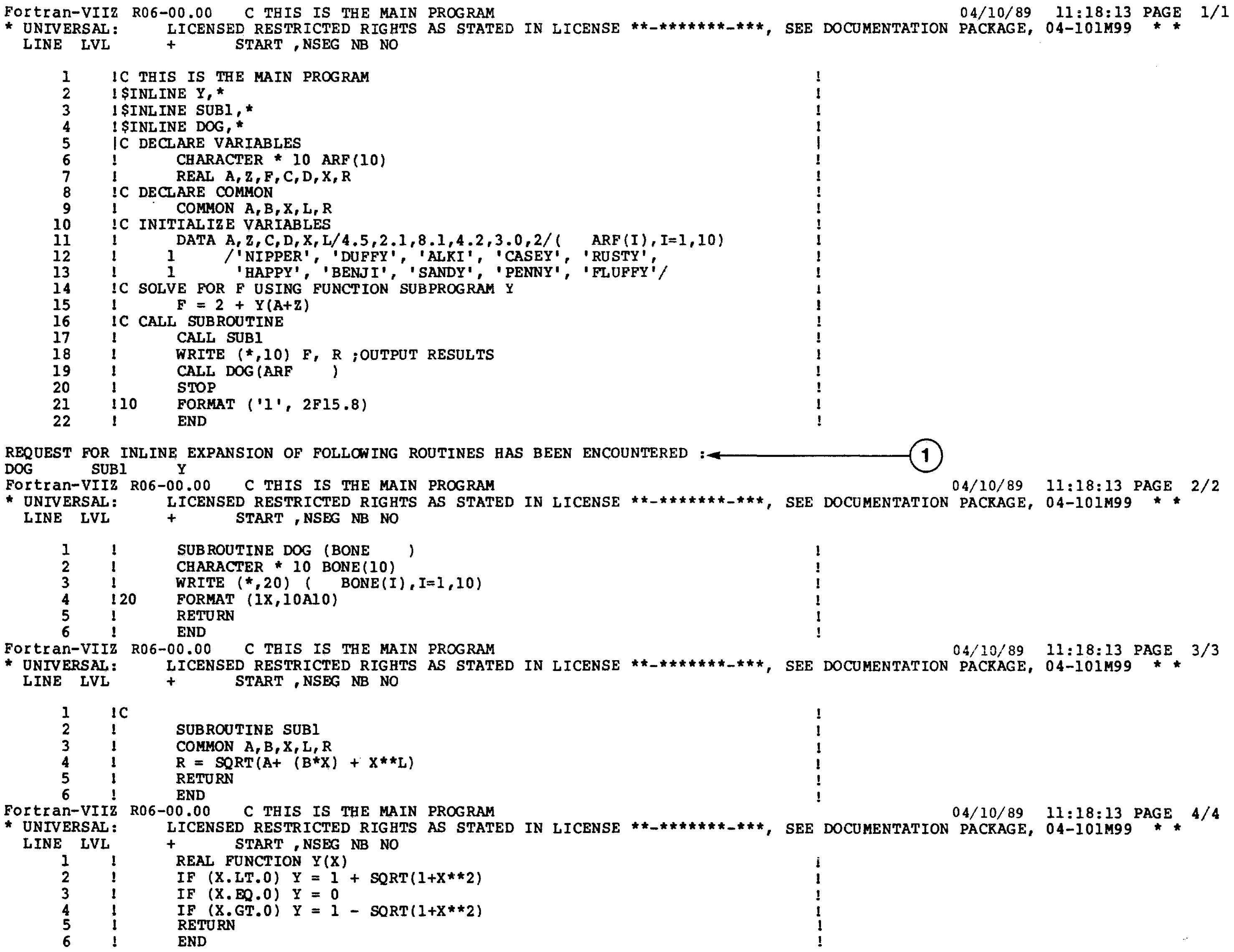
Figure 10-10. Example of Assembly Listing

F7Z Extended Listing

When a program requesting in-line expansion is compiled, the compiler automatically generates an extended listing in addition to the original source listing. The extended listing is the back translated form of the compiler's intermediate representation of the user program before it is optimized by the global optimization routines. At this intermediate stage, subprograms were expanded in-line as requested. All listings which are produced by the compiler after the extended listing (i.e., optimization summaries and assembly listing) refer to this listing of the source, rather than the user's source program.

To read the extended listing, you must understand how the original code is transformed at this stage. See Chapter 15 for more information on F7Z intermediate code translation. The program whose source listing is shown in Figure 10-11 will be used to illustrate the F7Z extended listing.

010-62-2



```
Fortran-VIIZ R06-00.00    C THIS IS THE MAIN PROGRAM                                          04/10/89  11:18:13 PAGE  1/1
* UNIVERSAL:      LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99   * *
  LINE  LVL       +      START ,NSEG NB NO

      1     !C THIS IS THE MAIN PROGRAM                                               !
      2     !$INLINE Y,*                                                              !
      3     !$INLINE SUB1,*                                                           !
      4     !$INLINE DOG,*                                                            !
      5     |C DECLARE VARIABLES                                                      !
      6     !       CHARACTER * 10 ARF(10)                                            !
      7     !       REAL A,Z,F,C,D,X,R                                                !
      8     !C DECLARE COMMON                                                         !
      9     !       COMMON A,B,X,L,R                                                  !
     10     !C INITIALIZE VARIABLES                                                   !
     11     !       DATA A,Z,C,D,X,L/4.5,2.1,8.1,4.2,3.0,2/(   ARF(I),I=1,10)         !
     12     !      1     /'NIPPER', 'DUFFY', 'ALKI', 'CASEY', 'RUSTY',                !
     13     !      1      'HAPPY', 'BENJI', 'SANDY', 'PENNY', 'FLUFFY'/               !
     14     !C SOLVE FOR F USING FUNCTION SUBPROGRAM Y                                !
     15     !       F = 2 + Y(A+Z)                                                    !
     16     !C CALL SUBROUTINE                                                        !
     17     !       CALL SUB1                                                         !
     18     !       WRITE (*,10) F, R ;OUTPUT RESULTS                                 !
     19     !       CALL DOG(ARF    )                                                 !
     20     !       STOP                                                              !
     21     !10     FORMAT ('1', 2F15.8)                                              !
     22     !       END                                                               !

REQUEST FOR INLINE EXPANSION OF FOLLOWING ROUTINES HAS BEEN ENCOUNTERED :◄─────────────(1)
DOG      SUB1      Y
Fortran-VIIZ R06-00.00    C THIS IS THE MAIN PROGRAM                                          04/10/89  11:18:13 PAGE  2/2
* UNIVERSAL:      LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99   * *
  LINE  LVL       +      START ,NSEG NB NO

      1     !       SUBROUTINE DOG (BONE     )                                        !
      2     !       CHARACTER * 10 BONE(10)                                           !
      3     !       WRITE (*,20) (   BONE(I),I=1,10)                                  !
      4     !20     FORMAT (1X,10A10)                                                 !
      5     !       RETURN                                                            !
      6     !       END                                                               !
Fortran-VIIZ R06-00.00    C THIS IS THE MAIN PROGRAM                                          04/10/89  11:18:13 PAGE  3/3
* UNIVERSAL:      LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99   * *
  LINE  LVL       +      START ,NSEG NB NO

      1     !C                                                                        !
      2     !       SUBROUTINE SUB1                                                   !
      3     !       COMMON A,B,X,L,R                                                  !
      4     !       R = SQRT(A+ (B*X) + X**L)                                         !
      5     !       RETURN                                                            !
      6     !       END                                                               !
Fortran-VIIZ R06-00.00    C THIS IS THE MAIN PROGRAM                                          04/10/89  11:18:13 PAGE  4/4
* UNIVERSAL:      LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99   * *
  LINE  LVL       +      START ,NSEG NB NO
      1     !       REAL FUNCTION Y(X)                                                !
      2     !       IF (X.LT.0) Y = 1 + SQRT(1+X**2)                                  !
      3     !       IF (X.EQ.0) Y = 0                                                 !
      4     !       IF (X.GT.0) Y = 1 - SQRT(1+X**2)                                  !
      5     !       RETURN                                                            !
      6     !       END                                                               !
```

Figure 10-11. Source Listing for In-line Expansion Program

Except for item number 1, the original source listing shown in Figure 10-11 has virtually the same format as the F7O and F7Z source listings. Item number 1 is only present on source listings of programs that request in-line expansion. This message informs the user which subprograms will be expanded in-line.

Figure 10-12 is the extended listing for this compilation. Because the extended listing comes from the compiler's internal representation of the user's program, it differs from the original source listing in a number of ways.

Some of these differences are shown by the numbered items in Figure 10-12 as follows.

Number **List Item**

1 Name of the main program unit. If no name is assigned by the user, the compiler generated name, .MAIN, is used.

2 Explicit type declaration of all variables and functions used in the main program and subprograms. To distinguish subprogram variable names from variable names used in the main program, the compiler precedes the name of each subprogram variable with the name of the subprogram followed by a period; i.e., *subprogram name.variable name.*

Notice that if the program uses a Concurrent RTL function, the compiler outputs it in its internal form; i.e., .SQRT.

NOTE All format statements are listed before the first executable statement in the program.

3 EQUIVALENT COMMON statement generated by compiler. This statement indicates the variables in the subprogram that occupy the same positions as the variables specified in the COMMON statement.

Number **List Item**

4 Explicit declaration of intrinsic or external functions used in the program.

5 EQUIVALENCE dummy argument statement. This statement indicates which arguments in the CALL statement are equivalent to the dummy arguments in the SUBROUTINE or FUNCTION statements.

6 Hollerith code replacement. The compiler replaces all quoted strings in FORMAT statements with equivalent Hollerith code.

7 Compiler-generated labels.

8 Expanded function Y as follows:

- Line 16 indicates the compiler generated variable (@100) assigned to the variable expression (A+Z) whose value is passed from the calling program to the subprogram through Y.X.
- Line 17 indicates that the argument Y.X for the statement function Y is to be assigned the value of the compiler generated variable @100.
- Lines 18 through 20 indicate the body of the expanded function Y which is represented by the compiler as follows:

  — All variable names of the subprogram Y are preceded by the subprogram name followed by a period.

  — .LT. is replaced by its symbolic equivalent <.

  — SQRT is replaced by its RTL internal name .SQRT.

  — .GT. is replaced by its symbolic equivalent >.

  — All integer values assigned to real variables are represented by real numbers; i.e., 0 replaced by 0.0.

- Line 21 indicates that the result passed from the subprogram Y to the calling program is to be assigned to the compiler generated variable @100.

Number **List Item**

9 Expanded subroutine SUB1 as follows:

- Line 24 indicates the body of the expanded subroutine SUB1 which is represented by the compiler as follows:

  — each variable of the subprogram SUB1 is preceded by its subprogram name followed by a period.

  — SQRT is replaced by its RTL internal name.

10 Elimination of trailing comments.

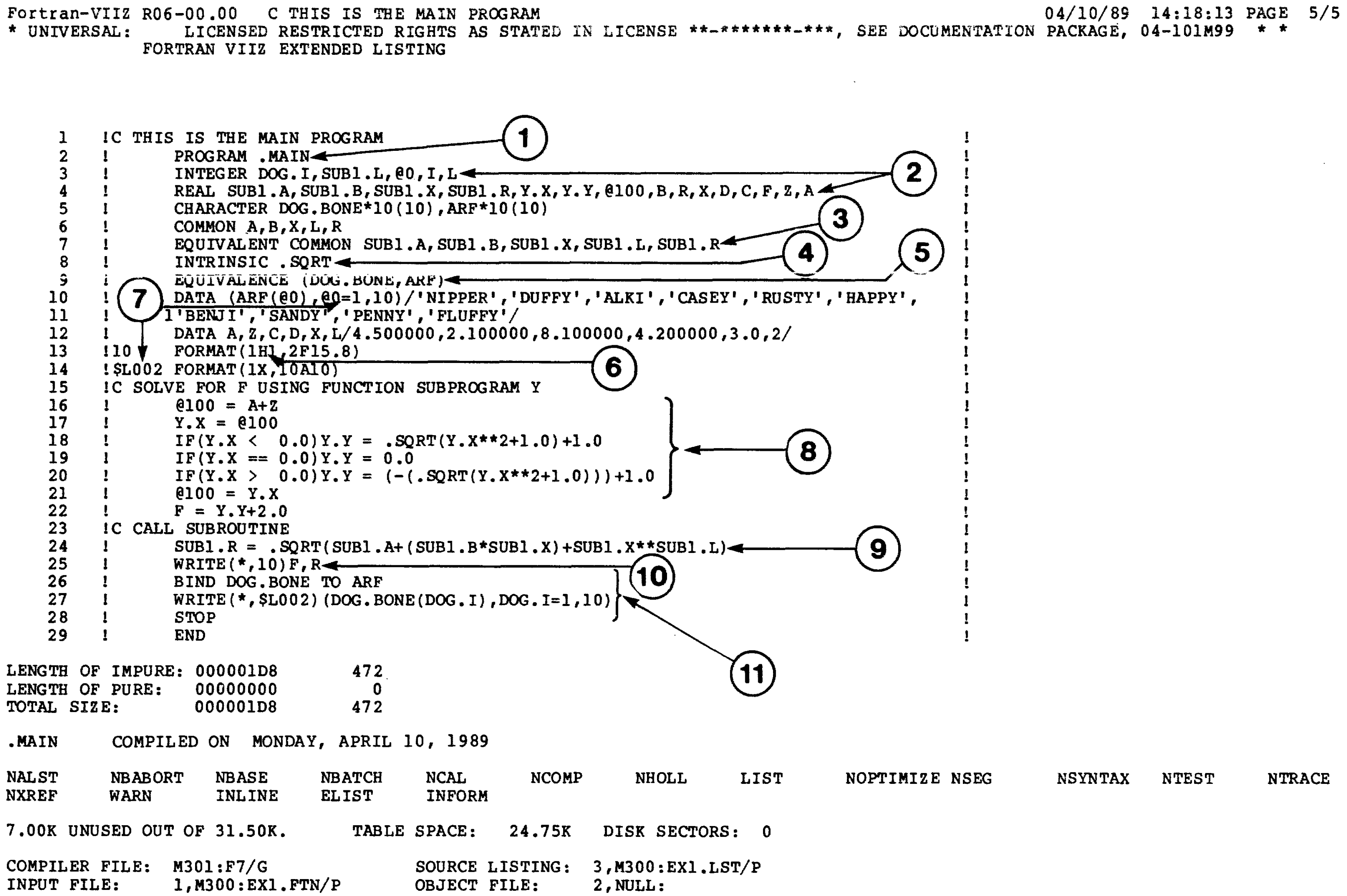
11 Expanded subroutine DOG as follows:

- Line 26 indicates the compiler generated BIND statement that binds the start of the dummy character array BONE to the location of the actual argument array ARF.
- Line 27 indicates the body of the subprogram DOG, represented by the compiler as follows:

  — each variable is preceded by the subprogram name DOG followed by a period.

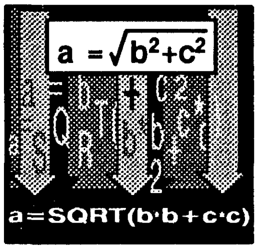
  — WRITE statement label is replaced by the compiler generated label $L002 which is unique to this expansion of the subprogram.

010-63-2



```
Fortran-VIIZ R06-00.00   C THIS IS THE MAIN PROGRAM                                        04/10/89  14:18:13 PAGE  5/5
* UNIVERSAL:     LICENSED RESTRICTED RIGHTS AS STATED IN LICENSE **-*******-***, SEE DOCUMENTATION PACKAGE, 04-101M99  * *
             FORTRAN VIIZ EXTENDED LISTING


    !C THIS IS THE MAIN PROGRAM                                                          !
    !       PROGRAM .MAIN                                                                !
    !       INTEGER DOG.I,SUB1.L,@0,I,L                                                  !
    !       REAL SUB1.A,SUB1.B,SUB1.X,SUB1.R,Y.X,Y.Y,@100,B,R,X,D,C,F,Z,A                !
    !       CHARACTER DOG.BONE*10(10),ARF*10(10)                                         !
    !       COMMON A,B,X,L,R                                                             !
    !       EQUIVALENT COMMON SUB1.A,SUB1.B,SUB1.X,SUB1.L,SUB1.R                         !
    !       INTRINSIC .SQRT                                                              !
    !       EQUIVALENCE (DOG.BONE,ARF)                                                   !
    !       DATA (ARF(@0),@0=1,10)/'NIPPER','DUFFY','ALKI','CASEY','RUSTY','HAPPY',      !
    !      1'BENJI','SANDY','PENNY','FLUFFY'/                                            !
    !       DATA A,Z,C,D,X,L/4.500000,2.100000,8.100000,4.200000,3.0,2/                  !
    !10     FORMAT(1H1,2F15.8)                                                           !
    !$L002 FORMAT(1X,10A10)                                                             !
    !C SOLVE FOR F USING FUNCTION SUBPROGRAM Y                                           !
    !       @100 = A+Z                                                                   !
    !       Y.X = @100                                                                   !
    !       IF(Y.X <  0.0)Y.Y = .SQRT(Y.X**2+1.0)+1.0                                    !
    !       IF(Y.X == 0.0)Y.Y = 0.0                                                      !
    !       IF(Y.X >  0.0)Y.Y = (-(.SQRT(Y.X**2+1.0)))+1.0                               !
    !       @100 = Y.X                                                                   !
    !       F = Y.Y+2.0                                                                  !
    !C CALL SUBROUTINE                                                                   !
    !       SUB1.R = .SQRT(SUB1.A+(SUB1.B*SUB1.X)+SUB1.X**SUB1.L)                        !
    !       WRITE(*,10)F,R                                                               !
    !       BIND DOG.BONE TO ARF                                                         !
    !       WRITE(*,$L002)(DOG.BONE(DOG.I),DOG.I=1,10)                                   !
    !       STOP                                                                         !
    !       END                                                                          !

LENGTH OF IMPURE: 000001D8       472
LENGTH OF PURE:   00000000         0
TOTAL SIZE:       000001D8       472

.MAIN     COMPILED ON  MONDAY, APRIL 10, 1989

NALST     NBABORT   NBASE     NBATCH    NCAL      NCOMP     NHOLL     LIST      NOPTIMIZE NSEG      NSYNTAX   NTEST     NTRACE
NXREF     WARN      INLINE    ELIST     INFORM

7.00K UNUSED OUT OF 31.50K.      TABLE SPACE:   24.75K   DISK SECTORS:  0

COMPILER FILE:  M301:F7/G           SOURCE LISTING:  3,M300:EX1.LST/P
INPUT FILE:     1,M300:EX1.FTN/P    OBJECT FILE:     2,NULL:
```

Figure 10-12. Extended Source Listing for In-line Expansion Program



11

FORTRAN VII RTL Routines

In this chapter

We introduce you to the general routines provided by the FORTRAN VII RTL. These routines allow you to terminate a program's execution, output informational messages, access the operating system time and date, control access to shared data, and access a program's run-time start options. The FORTRAN VII RTL also provides routines to control processing within a real-time system. These routines provide a variety of task manipulations such as performing system level input/output (I/O), I/O error handling, and controlling a 3200MPS processor. These types of RTL routines are described in the *OS/32 System Support Run-Time Library (RTL) Reference Manual.*

Topics include:

- Terminating execution using EXIT and EXITRE
- Accessing the system time and date
- Sending messages
- Controlling access to shared data
- Accessing run-time start options

Terminating Execution Using EXIT and EXITRE

The EXIT subroutine allows the calling task to terminate its own execution. If the calling task is nonresident, it is removed from memory. The EXITRE subroutine is the same as EXIT except that, if the task is resident, any assigned logical units are not closed at the end of task. The format for the subroutine is as follows:

CALL EXIT (*arg*)

or

CALL EXITRE (*arg*)

Where:

arg is an optional INTEGER*4 argument used to specify an end of task code. This code must be greater than 0, but less than 255.

Functional Details:

When *arg* is specified, it is truncated to an 8-bit value, thereby limiting *arg* to 255. If *arg* is not specified, EXIT will set the following end of task codes.

- 0 - Normal termination
- 255 - Task cancelled by operator or via an ABORT call

If the task is executing in the background, the return code can be used by a command substitution system (CSS) procedure to affect system control. For information concerning the writing of CSS procedures, see the *OS/32 Multi-Terminal Monitor (MTM) Reference Manual.*

Accessing the System Time and Date

The following RTL routines allow access to the system calendar, time of day clock, and interval clock.

DATE obtains the current calendar date from the operating system.

TIME obtains the current time of day from the operating system.

ICLOCK obtains the current time of day from the operating system in one of the following formats:

- hours: minutes: seconds
- ASCII
- number of seconds since midnight

NOTE The argument checking RTL checks all calls to real-time subroutines. All arguments to these routines must be stated as indicated in the following sections. If the number of arguments is incorrect, the following message is logged and the request is ignored:

progname: `INCORRECT NUMBER OF ARGUMENTS`

DATE Subroutine

The DATE subroutine obtains the current calendar date. Call DATE as follows:

CALL DATE (*ARRAY*)

Where:

ARRAY is an INTEGER*4 array of at least 3 elements whose first 3 elements receive:

First element - year 0 to 99
Second element - month 1 to 12
Third element - day 1 to 31

Example:

```
          INTEGER*4 ARRAY(3)
          CALL DATE(ARRAY)
          WRITE(5,10)ARRAY
    10    FORMAT(X,3I)
          END
```

Output:

```
    87   4    1
```

Functional Details:

The content of elements 1, 2, and 3 of ARRAY depend on the DATE option chosen at system generation (sysgen) time. The above explanation assumes that the mm/dd/yy DATE option was chosen. If the dd/mm/yy DATE option is chosen, the day (1 to 31) occupies element 2 and the month (1 to 12) occupies element 3.

TIME Subroutine

The TIME subroutine obtains the current time of day. Call TIME as follows:

CALL TIME (*ARRAY*)

Where:

ARRAY is an INTEGER*4 array of at least 3 elements whose first 3 elements receive:

First element - hour 0 to 23
Second element - minutes 0 to 59
Third element - seconds 0 to 59

Example:

```
          INTEGER*4 ARRAY(3)
          CALL TIME(ARRAY)
          WRITE(5,10)ARRAY
   10     FORMAT(X,3I)
          END
```

Output:

```
16 00 00
```

ICLOCK Subroutine

The ICLOCK subroutine obtains the current time of day in one of three formats. Call ICLOCK as follows:

CALL ICLOCK (*IFUNC,DEST*)

Where:

IFUNC is an INTEGER*4 argument that specifies the required format of the time of day as follows:

0 - three integers: hours, minutes, seconds
1 - 8-byte formatted ASCII string
2 - number of seconds since midnight

DEST is a variable where the time of day is to be stored. There are different requirements for DEST according to the IFUNC specified.

If IFUNC=0, DEST must be an INTEGER*4 array of at least 3 elements, into the first 3 of which the time of day is placed as follows:

DEST (1) - hour 0 to 23
DEST (2) - minute 0 to 59
DEST (3) - second 0 to 59

If IFUNC=1, DEST must be an INTEGER*4 array with at least 2 elements or a double precision variable into which the time of day is placed as follows:

DEST = *hh:mm:ss*

If IFUNC=2, DEST must be an INTEGER*4 variable into which ICLOCK places the time of day as follows:

DEST = *number of seconds since midnight*

Example:

```
        INTEGER*4 ARRAY(3),IFUNC
        IFUNC=0
        CALL ICLOCK(IFUNC,ARRAY)
        WRITE(5,10)ARRAY
10      FORMAT(X,3I)
        END
```

Output:

```
16 00 00
```

Functional Details:

If the value of IFUNC is other than 0, 1, or 2, an ILLEGAL FUNCTION CODE error message is output, and the request is ignored.

Sending Messages Using CONMSG

The CONMSG subroutine outputs messages to the user's terminal. This routine allows a task to output a message to the user's terminal without the caller having to know the device mnemonic of the terminal. Call CONMSG as follows:

CALL CONMSG (*NCHAR,MSG*)

Where:

NCHAR is an INTEGER*4 argument that specifies the number of characters in MSG that is to be displayed. If this is not specified, the whole string MSG is displayed.

MSG is a character constant, a Hollerith constant, a variable or an array containing the message.

Functional Details:

NCHAR characters, starting at *MSG*, are output to the user's terminal. If *NCHAR* is not specified, the whole string denoted by *MSG* is displayed.

If *NCHAR* has a value less than one, the following error message is logged and the request is ignored:

```
CONMSG:  ARGUMENTS (ZERO, NEGATIVE)
```

Example:

```
INTEGER*4 NCHAR
CHARACTER*8 MSG
DATA MSG /'HELLO'/
NCHAR = 5
CALL CONMSG(NCHAR,MSG)
END
```

Output:

```
HELLO
```

Controlling Access to Shared Data

FORTRAN VII provides for the sharing of data between tasks via shared segments or task common areas. The optimizing compilers' perception of access to these areas can have serious implications for optimization. The following function and subroutine are provided to signal current usage of shared data. For more information about shared data area usage and optimization, see the sections " Linking Shared Data Areas" and "Guidelines for Preparing Source Code for Optimization."

LOKON Function

The logical function LOKON sets the high order bit of its argument. If this bit is already set, LOKON returns ".TRUE.". If the high order bit of its argument was not set, LOKON sets it and returns ".FALSE.". This function alerts the F7 optimizing compilers that subsequent references to members of common are under exclusive control of the calling task. This control is granted solely by user-stipulated convention with all tasks which have access to this common area. Succeeding references to these variables will not be moved across this call. See Chapter 4 for more information on shared data usage.

Invoke LOKON as follows:

LOKON(*I*)

Where:

I	is any halfword aligned variable; i.e., not character, Holerith or LOGICAL*1.

Functional Details:

This function is implemented via the machine instruction test and set. Use the FORTRAN function TESET when control of data areas is not to be implied.

LOKOFF Subroutine

The LOKOFF subroutine resets the high order bit of its argument. This is the companion routine to LOKON. It must be invoked by a task which has gained exclusive access rights to shareable segments or task commons by calling LOKON to signal that it is releasing access rights.

Failure to observe this convention may result in indeterminate run-time data in COMMON. Call LOKOFF as follows:

CALL LOKOFF(*I*)

Where:

I is any halfword aligned variable; i.e., not character, Holerith or LOGICAL*1.

Functional Details:

This subroutine is implemented via the machine instruction RBT. The FORTRAN subroutine BCLR can be used when control of data areas is not to be implied.

Example:

```
      COMMON /FLAG/ FLAGA,FLAGB
      CALL P(FLAGA)
         .
         .
         .
      CALL V(FLAGA)
         .
      STOP
      END
      SUBROUTINE P(/FLAG/)
    5 IF(FLAG .LT. 0) GOTO 5
      IF(LOKON(FLAG)) GOTO 5
      RETURN
      END
      SUBROUTINE V(/FLAG/)
      CALL LOKOFF(FLAG)
      RETURN
      END
```

Accessing Run-Time Start Options Through GETOPTS

The GETOPTS routine allows users to access their run-time start options from a FORTRAN program. Call GETOPTS as follows:

CALL GETOPTS(*opstring,length*)

Where:

opstring is a character variable or array element in which GETOPTS returns the string of start options.

length is the length of the string returned in opstring.

Example:

```
      CHARACTER OPSTRING*80
      INTEGER*4 LENGTH
      CALL GETOPTS(OPSTRING,LENGTH)
      TYPE *,'Length = ',LENGTH
      TYPE *,'String = ',OPSTRING
      END
```

```
START,BEGIN=10 END=200
```

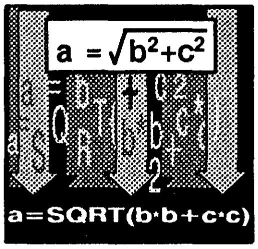
(This assumes that the code was compiled, linked, and the task loaded.)

Output:

```
Length = 16
String = BEGIN=10 END=200
```

Functional Details:

Each system start option must be preceded by the % character and all system start options must come before any user start options. The delimiter for start options is either a comma or a space. The first option, from left to right, which does not start with the % character marks the beginning of the options. See Chapter 7 for the available run-time system start options.



12

FORTRAN VII XPA System

In this chapter

We discuss the execution profile analysis (XPA) system. XPA is a tool which monitors your program during execution and helps you determine the optimal candidates for in-line expansion or fine tuning.

Topics include:

- How to include the XPA in your program
- Analyzing the results using the XPA commands

Introducing the XPA System

To aid in deciding which subprograms are likely candidates for expansion under the optimizing compilers, Concurrent provides the XPA. XPA in its basic form does not require program recompilation. It is a two part system. The first part, which must be included at Link time, creates an address trace during program execution. An address trace is a statistical sampling of run-time locations trapped at specified regular time intervals.

The second part of XPA is a separate task which analyzes the address trace and produces 'execution frequency' tables by either:

- modules, or
- address ranges.

This task is run after completion of the user task (u-task).

NOTE When using XPA, non-FORTRAN subroutines should not use general register 1 for any purpose other than FORTRAN's work space pointer (even if general register 1 is saved at the entry and restored at the exit of the subroutine). If general register 1 is used for any other purpose, run-time faults occur when using XPA.

Timing Your Program

XPA provides the ability to trace addresses at specified time intervals during program execution.

Including the Timer in Your Program

You can easily include the XPA system in your task without recompilation of any program module. However, if you need more flexibility, you can utilize the user interface routine, XPA_SET, supplied with the XPA system. This interface is described later in this chapter.

In its basic form, XPA traps the current address every 10ms. Immediately after the FORTRAN VII run-time environment is initialized, XPA issues its initial timer trap. Thereafter, every 10ms the u-task is interrupted by the operating system, and the trap is passed to XPA. The task's current address is stored in an internal buffer and another timer trap is issued. When the 1,024 byte internal buffer is full, it is written to MAXLU. This process continues until either the timer is turned off by a call to XPA_SET or the task goes to end of task. If the task completes normally, XPA dumps the last buffer to MAXLU. If the task is cancelled for any reason, the last buffer is lost. (Note the buffer holds up to 256 entries.)

If MAXLU is preassigned, it must be an indexed file of record size 1,024. It should be blocked at a factor of 4 or greater to ensure a reasonable input/output (I/O) speed. If MAXLU is unassigned, XPA creates an indexed file of record size 1,024/4 and assigns it to MAXLU. The name of this file is the program's task filename appended with .XPA; e.g., *usertask*.XPA. If this file already exists, it is deleted and reallocated.

To append XPA data to an already existing XPA file, reprotect it with keys 0100. This prevents it from being reallocated.

The maximum logical unit (lu) number is used to perform all I/O operations of the XPA timer. MAXLU cannot be used during program execution for any purpose other than for XPA output. This lu may be changed by means of the OPTION LU=*nn* command at link time. MAXLU may then be calculated as *nn*-1. The default for *nn* is 15, giving a default MAXLU of 14. See the *OS/32 Link Reference Manual* for more information.

The timer interrupt module of XPA exists in the FORTRAN VII run-time library (RTL). To begin timing immediately after the program is started, whether or not XPA_SET is present in your source, include .XPATIMR from the RTL at Link time. To delay the start of the timer, call XPA_SET from the appropriate spot in your source program and do not include .XPATIMR at link time.

Additional XPA Link requirements are:

- 1.5kB workspace above normal task requirements.
- An ADDRESS Link map if a frequency profile by module is desired.

A typical link command sequence for including XPA is as follows:

```
>OP FL,DFL,WORK=X1600
>LIB F7PFUT/S              (LIB against F7PFUT.OBJ)
>LIB F7RTL/S               (LIB against F7RTL)
>IN user                   (includes user's program)
>IN F7PFUT/S,.XPATIMR      (includes the XPA timer from
                            the F7PFUT.OBJ)
>MAP user.MAP,ADDR,XREF (gets map on the user.MAP file)
>BUI user                  (builds user task with XPA)
>END
```

For Link R00-01 or lower, this same link command applies except that the first line of the command should be replaced with the following.

```
>OP FL,DFL,WORK=1600
```

In order to Link with an RTL shared segment, add the following information after the first command line.

```
>RES FORTLIB.IMG
```

See the section entitled "Linking Shared Segments" found in Chapter 7 for more information on RTL shared segments.

XPA_SET Routine

The RTL subroutine XPA_SET performs multiple functions. It can turn the timer on or off, change the interval between ticks, or mark all subsequent ticks with a tag character. The call to XPA_SET consists of 0, 1, or 2 arguments in any order. If no arguments are specified, it starts the timer with either the default values (the first time) or the previously specified values (otherwise). If any argument is missing, it keeps the value from the previous call.

One of the two arguments specifies the timing interval in milliseconds. It must be an INTEGER*4 value. If it is <=0, the timer is turned off and the buffer is written. If it is >0, the timer is restarted with this new interval. The default interval value is 10ms.

The other argument specifies an address tag. It must be a CHARACTER*1 value between 'A' and 'Z' or a '0' (zero). Any other value is taken as '0'. The address tag is used to identify the execution of separate parts of a program with respect to either address space or time. For instance, an address tag can be used to identify ticks from different overlays. In this case, the addresses cover the same space, but are really from different parts of the program. In another case, the address tag can be used to distinguish ticks from the same routine which was called at different times. The second phase of the XPA system, the XPA analyzer, allows you to use this tag when you display timing results. The default value of the tag is '0'.

Some examples of calls to XPA_SET follow:

```
CALL XPA_SET (15,'A')      ; starts the timer with a 15ms interval.
                           ; Addresses are tagged with 'A'.

CALL XPA_SET ('B')         ; Addresses are now tagged with 'B'.  The
                           ; interval is still 15ms.

CALL XPA_SET ('0',0)       ; Stops the timer and sets the tag
                           ; back to default '0'.

CALL XPA_SET               ; Restarts the timer with previous interval,
                           ; i.e. 15ms and with tag '0'.
```

Influences Upon the Trace Profile

XPA uses supervisor call 2 (SVC2) code 23 option X'00' to set the timer trap. Option X'00' causes the timer interval to run concurrently with program execution. When the interval elapses, a service request is added to the program's task queue. When the program is in CURRENT state, the request is serviced by XPA. XPA's central processing unit (CPU) and I/O time to service a trap is excluded from the user's trace.

If an interval ends when the program is in READY, WAIT, or PAUSED state, the tick gets charged to that part of the program which ran last or which issued the wait or pause. For instance, if the program is in I/O wait and if the interval is smaller than the time it takes to service the I/O request, the tick occurs in the I/O routine. That is, when the program is reading or writing data, the tick occurs in the FORTRAN RTL I/O routines. This should not unduly bias the sample as there will be only one tick per read or write. In short, program composition and system load can affect the address trace sample. Some of these factors are:

- amount of I/O by the program,
- amount of system wide I/O, and
- number of CPU bound tasks in the system.

These factors affect the number of times the task is in READY, WAIT, or PAUSE state when the trap is due. Therefore it is possible to get an execution profile on a stand alone machine that is somewhat different from one on a shared system. In general, the results obtained from a multi-terminal monitor (MTM) environment are sufficiently accurate, especially with respect to heavily frequented parts of the program.

Interfacing with INIT/ENABLE and Error Conditions

The XPA system does not preclude the use of any of OS/32 trap handling facilities available in FORTRAN VII's real-time routines INIT and ENABLE. These routines are aware of XPA and do not interfere with each other.

There are three errors which may occur during the execution of a task in which XPA was included.

- Insufficient user memory.
- Insufficient amount of system space available to store the timer trap.
- Overflow of user's task queue.

Cases 1 and 2 cause the XPA to log an error message and cancel the program with end of task code 240 and 241, respectively. For case 1, reload the program with a load size of 1.5kB more than previously. For case 2, increase the system space using the system console and rerun the program. Case 3 causes OS/32 to cancel the program with an end of task code 1000. This error is only possible if the program, itself, is using FORTRAN VII's real-time facility. The size of the queue is fixed at 50 entries. The queue size can be changed by a call to INIT. See the *OS/32 System Support Run-Time Library (RTL) Reference Manual* for details on the INIT routine.

How to Analyze the Results

The second part of the XPA system is a task, XPATAB.TSK, which formats the result from .XPATIMR in a ledger like fashion. The ledger contains a list of either module names, if a Link map is available, or a range of addresses. Both lists are sorted by the frequency of execution. Then, by concentrating on those parts of the program which consume the most time, you can determine the optimal candidates for inline expansion or for fine tuning an algorithm.

XPATAB assigns all the necessary files after it has been started. However, if you wish to preassign, perform the following:

- assign lu1 to the program trace file,
- assign lu3 to the output device or file the analysis is to be printed,
- assign lu5 to the command input file, defaults to CON:, and
- assign lu6 to the logging device for messages, defaults to CON:.

There are two kinds of table formats available to the user:

- by module name, or
- by address.

To output the table by module name, a standard Link map with the ADDRESS option must be available. XPATAB uses this map to find out the module name in which each tick occurred. It then prints a table associating module names with their frequencies. If an ADDRESS map is not available, or if you want to dissect a module, the address format is available. The address format only associates frequencies with ranges of addresses. You must then associate the addresses to your program.

Basic XPA Commands

The preceding commands allow you to manipulate the XPA. You must have loaded and started XPATAB.TSK prior to using these commands.

PROGRAM Command

To enable the module format and to clear the XPATAB environment i.e., start afresh, use the PROGRAM command as follows:

```
PROGRAM [pgmname]
```

Where:

pgmname is a program name of a traced task. XPATAB appends .MAP to *pgmname* i.e., *pgmname*.MAP to find the program's map file. The map file is searched for on the user's private, group, and system accounts. If *pgmname* is absent, XPATAB logs the current program name to lu6.

INPUT Command

The INPUT command specifies which trace file to analyze. The format is:

```
INPUT [fdi]
```

Where:

fdi is the file descriptor (fd) of the trace file generated by .XPATIMR. If this command is not specified, *pgmname*.XPA is assumed, if possible. If the argument is absent, XPATAB logs the current trace fd.

OUTPUT Command

The OUTPUT command specifies to which file the analysis is to be output. The format is:

```
OUTPUT [fdo]
```

Where:

fdo is the file to which the analysis ledger is to be printed. If *fdo* is nonexistent, it is allocated as indexed 80/1. If this command is not specified, CON: is assumed. If *fdo* is absent, XPATAB logs the current *fdo*.

TAG Command

If the character tag facility of XPA is used in a FORTRAN program, XPATAB provides the TAG command to restrict the analysis to a specific set of tagged trace entries. If a Link map is available, a tag may be associated to an overlay node. The format is:

```
TAG [t [=node] | -]
```

Where:

t is a character tag in your program (A through Z and 0 (zero)).

node is the overlay name from an OVERLAY command in your Link command sequence.

— means to accept all trace entries regardless of tag.

If this command is not specified, a tag of 0, the default from XPATIMR, is assumed. Essentially, the TAG command specifies which tag is current and causes analysis to be restricted to a particular set of trace entries carrying that tag. Furthermore, for those who have an overlayed program, =*node* is provided to specify a path from the root up to and including that node. The Link address map must be accessible to XPATAB via the PROGRAM command for this feature to be available. XPATAB ascertains the path of the overlay tree for this node and only consider those module names occurring along that path for analysis.

If there are no arguments present in the command, XPATAB logs a list of only those tags and their associated nodes, if any, which were entered by TAG. The current tag is highlighted.

TABLE Command

The TABLE command causes the analysis ledger to be produced on *fdo*. Its format is:

TABLE [*ADDRESS*]

The optional argument, *ADDRESS*, specifies that the address range format is desired; it is only needed if a *pgmname* was specified and it is to be temporarily ignored. In general, TABLE defaults to a module format if a *pgmname* was specified; otherwise an address range format is given. The following is a sample of a module ledger:

```
RUTABAGA:-,.ROOT    ;title goes here
TOTAL TICKS = 5120; # OF MODULES = 61; MAX # AVAILABLE = 8057
5120 TICKS IN RANGE, 100.0%;  0 TICKS OUTSIDE RANGE, 0.0%
MODULE: POTATO               3717 TICKS,   72.6%
MODULE: DOORMAT               576 TICKS,   11.3%
MODULE: PRETZEL               242 TICKS,    4.7%
MODULE: PHYSIQUE              119 TICKS,    2.3%
     TOTAL SHOWING:          4654 TICKS,   90.9%
```

Where:

`TOTAL TICKS` is the total number of ticks in the sample.

`# OF MODULES` is the total number of modules including RTL routines specific to an analysis.

`MAX # AVAILABLE` is the maximum of modules that can be analyzed given the load size of XPATAB.

`TICKS IN RANGE` is the number of ticks in the analysis.

`TICKS OUTSIDE RANGE` is the number of ticks outside the range of an analysis.

NOTE All percentages are with respect to the total number of ticks in the sample.

`TOTAL SHOWING` is the number of ticks shown on the ledger. This number is not always 100% of the total number of ticks in the sample; it is based on the number of entries that the ledger is limited to. See the LIMIT command described later on in this chapter for more information.

A sample address ledger is:

```
TAG = -;title goes here
SPECIFIED RANGE:0-1AC54; STEP SIZE: 100; BIAS:0
TOTAL TICKS = 5120; # OF BUCKETS = 424; MAX # AVAILABLE = 8057
5120 TICKS IN RANGE, 100.0%;  0 TICKS OUTSIDE RANGE, 0.0%
RANGE:   DF00-  DFFF        991 TICKS,   19.4%
RANGE:   D800-  D8FF        791 TICKS,   15.4%
RANGE:   DA00-  DAFF        352 TICKS,    6.9%
RANGE:   DE00-  DEFF        272 TICKS,    5.3%
     TOTAL SHOWING:        2406 TICKS,   47.0%
```

Where:

SPECIFIED RANGE	is the absolute address range for an address format.
STEP SIZE	is the step size for each address interval.
BIAS	is the address bias for an analysis.

A further explanation of range, step size, and bias can be found under their respective commands RANGE, STEP, and BIAS. Both the modules and the address intervals are sorted according to their frequencies in descending order.

Commands for Analyzing Modules

The preceding commands are sufficient to analyze both overlayed and nonoverlayed programs by module.

A typical XPATAB command sequence is as follows:

```
*L XPATAB
*ST
 EXECUTION PROFILE ANALYZER Rxx-xx.xx
>P user          (Program command)
>T               (Table command)
 .               (Module ledger appears here)
>END
```

The remainder of the commands add address range capability and additional format control.

BIAS Command

The BIAS command sets up an address bias specified either by address or by module. This bias is then applied upon the address range intervals in the address range format. This is useful when conducting a micro-inspection of a module's algorithm and the FORTRAN VII assembly listing of that module is close at hand. The format is:

```
BIAS ['adr' | modname[:node] | *]
```

Where:

'*adr*' represents a hexadecimal address (up to six digits) enclosed in apostrophes.

modname is a module name in the program.

node is the overlay name in which *modname* may be found.

* means set the bias to the start of the current RANGE.

The address to which the bias is set if the modname argument is specified is the pure starting address. The impure address of the module is used only if there is no pure code for that module.

If this command is not specified, a bias of 0 is assumed. If the argument is absent, XPATAB logs the bias. The bias has no effect upon a module ledger format.

RANGE Command

The RANGE command restricts the range in which the analysis is to be performed and thus the information printed in the ledger. It can restrict the range either by address or by module name. RANGE has no effect upon a module ledger format. The format is:

```
RANGE [from [ - to ]] | -
```

Where:

from and *to* — are either '*adr*'[*] or *modname*[:*node*]. '*adr*', *modname*, and *node* are as defined from BIAS.

* — means the address is relative to the bias; otherwise the address is absolute.

Some examples of range values are:

'*adr*'-'*adr*' — means to span the address range from '*adr*' to '*adr*'.

'*adr*'- — means from '*adr*' to 'FFFFFF' or Utop.

- — means from '0' to 'FFFFFF' or Utop (the default).

'0' * - — means from the bias to 'FFFFFF' or Utop.

modname -modname — ranges the address from *modname* to *modname*, inclusive.

modname — sets the range to include only that module.

modname :node -modname — ranges from *modname* in overlay *node* to *modname* in *node*.

modname :node -modname :node — ranges from *modname* in overlay *node* to *modname* in *node* where *node* is accessible from *node*.

If *modname* is specified without a node, and that *modname* occurs in more than one node of a program, the first occurrence in the map is used. If *modname* is specified without a node, then the node containing *modname* is assumed. If RANGE is not specified, the range is from '0' to 'FFFFFF' or Utop if a map is available. If no arguments are specified, XPATAB logs the current range.

STEP Command

To set the length of the address interval for an address ledger, use the STEP command. Its format is:

```
STEP [size]
```

Where:

size is a hexadecimal number (up to six digits) specifying the address interval. The default step size is 100. If no argument is specified, XPATAB logs the current step size.

LIMIT Command

The LIMIT command limits the number of frequency entries i.e., modules or address intervals output to the ledger. The format is:

LIMIT [*lim*]

Where:

lim is an integer decimal number between 1 and 100. The default for *lim* is 18 entries. If no argument is specified, XPATAB logs the current frequency-entry limit.

TITLE Command

The TITLE command places its argument on the top line of the analysis ledger. Its format is:

TITLE [*title*]

Where:

title is the first 76 characters after the first nonblank character from the keyword TITLE, inclusive.

If no argument is specified, XPATAB logs the current title.

LOCATE Command

To find out where a particular module occurs in your program, use the LOCATE command. Its format is:

LOCATE *modname*

It finds and logs all occurrences of *modname*, their beginning and ending addresses of their pure code and their overlay nodes, if any. The log arrangement is:

modname$_1$ *saddr*$_1$ - *eaddr*$_1$ NODE:*overlay*
modname$_2$ *saddr*$_2$ - *eaddr*$_1$ *NODE:overlay*

NODE Command

To find out what overlay nodes appear in the Link map, use the NODE command. It accepts no arguments and logs a list of all the overlay nodes appearing in the Link map.

MAP Command

To ascertain the members of a particular overlay node or to dump the entire program map containing all overlays and their members, use the MAP command. Its format is:

MAP [*node*]

When *node* is specified, only that overlay node is detailed; otherwise, all overlay nodes are detailed. Note that nonoverlayed programs always contain a .ROOT node. MAP's logging format is:

UBOT: *xxxx*   UTOP: *xxxx*
NODE: *node*

modname$_1$*:saddr*!*modname*$_2$*:saddr*!*modname*$_3$*:saddr*!

modname$_4$*:saddr*!*modname*$_5$*:saddr*!*modname*$_6$*:saddr*!

HELP Command

The HELP command logs to lu6 a list of all available commands of XPATAB and their syntax. For this revision it does not accept a command argument.

Merging XPA Files

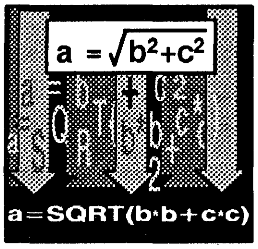
Two or more XPA files generated by different runs of the same program may be merged in order to summarize their contents by using the COPY/32 Utility as follows:

```
*L COPY32
*AL SUM.XPA,IN,1024/4
*ST
>OP BIN
>COPY in1.XPA,SUM.XPA
>COPY in2.XPA,*
>END
```

XPATAB may now be run on this summary file by specifying 'IN SUM.XPA'.

Ending the Session

To end the analysis session use the END command. To begin a fresh session use either the PROGRAM or NEW command. Both commands clear the XPATAB environment; NEW command, however, does not specify a *pgmname.*



13

FORTRAN VII CRA System

In this chapter

We discuss the call recording analysis (CRA) System. This utility is similar to the execution profile analysis (XPA) (described in Chapter 12). The difference is that the CRA creates and analyzes the call graph of an executing program and produces tables of call counts identifying the caller and the callee.

Topics include:

- How to use the CRA
- Analyzing results
- Limitations of the CRA

Introducing the CRA

To expedite the process of selecting which subprograms are likely candidates for in-line expansion under the FORTRAN VII optimizing compiler, use the CRA system. This utility is easy to use and does not require program recompilation. It consists of two parts:

- The first part, which must be included at link time, creates a dynamic call graph of an executing program in memory. This graph is then written to a disk file.
- The second part, the call recorder analyzer, is a separate task which analyzes the dynamic call graph and produces tables of all counts identifying the caller and the callee.

The space requirements of the CRA are as follows:

- The size of CRA is 5.5kB, CRAXPA is 8.0kB, including the XPA routines.
- The size of run-time library (RTL) routines used by CRA/CRAXPA is 28.5kB.
- The workspace requirement of CRA/CRAXPA is approximately 8kB (depends on size of dynamic call graph).

Analysis of Your Program

The call recorder is easy to include in your task and does not require recompilation of any program module. Link it into your task and it automatically begins to operate.

The call recorder logs a message to the console when it is initiated. It opens the link address map (*user*.MAP) which must be the task filename with the extension '.MAP' appended. The call recorder installs a breakpoint supervisor call 14 (SVC14) on each entry point in the program with the exception of those routines which begin with a period (.). For each entry point, a table entry is built in memory which serves as a node of the dynamic call graph, as well as holding the instruction replaced by the breakpoint. The map file is then closed and the call recorder exits, allowing the task being monitored to execute.

When a breakpoint occurs, control is passed to the call recorder's trap handler. The call is charged to the callee and is marked as being from the caller, possibly adding a new arc to the dynamic call graph. This process continues until the user task (u-task) is complete.

Immediately before the u-task terminates, the call recorder's end of task handler is invoked. An indexed file of record size 1024/4 is created and given the name of the task with extension '.CRA'. If this file exists, it will be deleted and reallocated. The dynamic call graph is then written to this file. To append CRA data to an already existing CRA file, reprotect it with keys 0100. This will prevent it from being reallocated.

There are two versions of the call recorder: .CRA and .CRAXPA. .CRA is the call recorder, as previously described. .CRAXPA is the call recorder integrated with XPA in such a way that both run simultaneously. In order to enable the CRA system, either version is included into the user program at link time.

The CRA link requirements are as follows:

- Approximately 8kB (2000) work space above normal task requirements depending upon the size of the dynamic call graph. This workspace includes all XPA requirements.
- An address link map on file *user*.MAP. This map file must be on the same volume as the task and must be deleted and reallocated each time the task is linked.

A typical link command sequence for including CRA is as follows:

```
>OP FL,DFL,WORK=X3000
>LIB F7PFUT/S              (LIB against F7PFUT.OBJ)
>LIB F7RTL/S               (LIB against F7RTL)
>IN user                   (includes user's object)
>IN F7PFUT/S,.CRA  (or) IN F7PFUT/s,.CRAXPA
                           (includes object from F7PFUT.OBJ)
>MAP user.MAP,ADDR,XREF    (gets map)
>BUI user                  (builds task)
>END
```

For Link R00-01 and lower, this same link command sequence applies except that the first line will be replaced by:

```
>OP FL,DFL,WORK=3000
```

In order to Link with an RTL shared segment, add the following information after the first command line:

```
>RES FORTLIB.IMG
```

See the section on "Linking Shared Segments" found in Chapter 7 for more information on RTL shared segments.

To append CRA data to an already existing CRA file, reprotect it with keys 0100. This will prevent it from being reallocated.

The CRA system does not preclude the use of any of the OS/32 trap handling facilities available in FORTRAN VII's real-time routines INIT and ENABLE.

The maximum logical unit (lu) number is used for all call recorder input/output (I/O). This lu must not be preassigned and may not be used by the user program. This lu may be changed by relinking the task with a larger limit on the OPTION *lu=nn* link command.

CRA imposes an overhead of approximately 310μs per call (on a Model 3250). The overhead for CRAXPA is approximately 380μs per call.

CRA Limitations

The limitations of CRA are as follows:

- Because breakpoints are installed at run-time, the task must not be linked segmented (pure and impure), although the call recorder is itself sharable.
- Overlayed tasks are not supported for the same reason.
- Recording call information over a portion of the program is not supported. Recording is always enabled for the entire program.
- Programs that abort abnormally will lose their call recorder information. This information is only written by normal program termination (STOP, END of main program or CALL EXIT). Note that normal program termination can be simulated by continuing a paused program at entry point '.V'.
- Modules whose names begin with '.' are not included in the call graph. This keeps the call recorder from significantly slowing down the program.
- CRAXPA automatically starts XPA at program initialization as if '.XPATIMR' were linked into the task. There is no provision for a delayed start of XPA. The default XPA interval (10ms) is used, but may be changed by calling XPA_SET immediately in the main program.
- XPA_SET must not be called if the CRA version is selected instead of CRAXPA.
- A subroutine call which is immediately followed by a branch (i.e., GOTO) will be recorded as occurring at the target of the branch. This is due to an optimization performed by the FORTRAN VII O and Z compilers.

Error Conditions

There are four errors which may occur during the execution of a task in which CRA was included.

- Insufficient user memory.
- MAP file nonexistant or ADDRESS map not found.
- I/O error reading the MAP file.
- I/O error writing the CRA file.

Cases 1 and 2 cause CRA to log an error message and cancel the program with end of task code 240 and 241, respectively. Reload the program with a larger load size. Case 3 causes CRA to pause (uses standard FORTRAN I/O). Case 4 causes CRA to log an error message and cancel the program with end of task code 242.

How to Analyze the Results

The second part of the call recording analysis system is a task, CRATAB.TSK, that formats the result from the call recorder into two tables. See Figures 13-1 and 13-2.

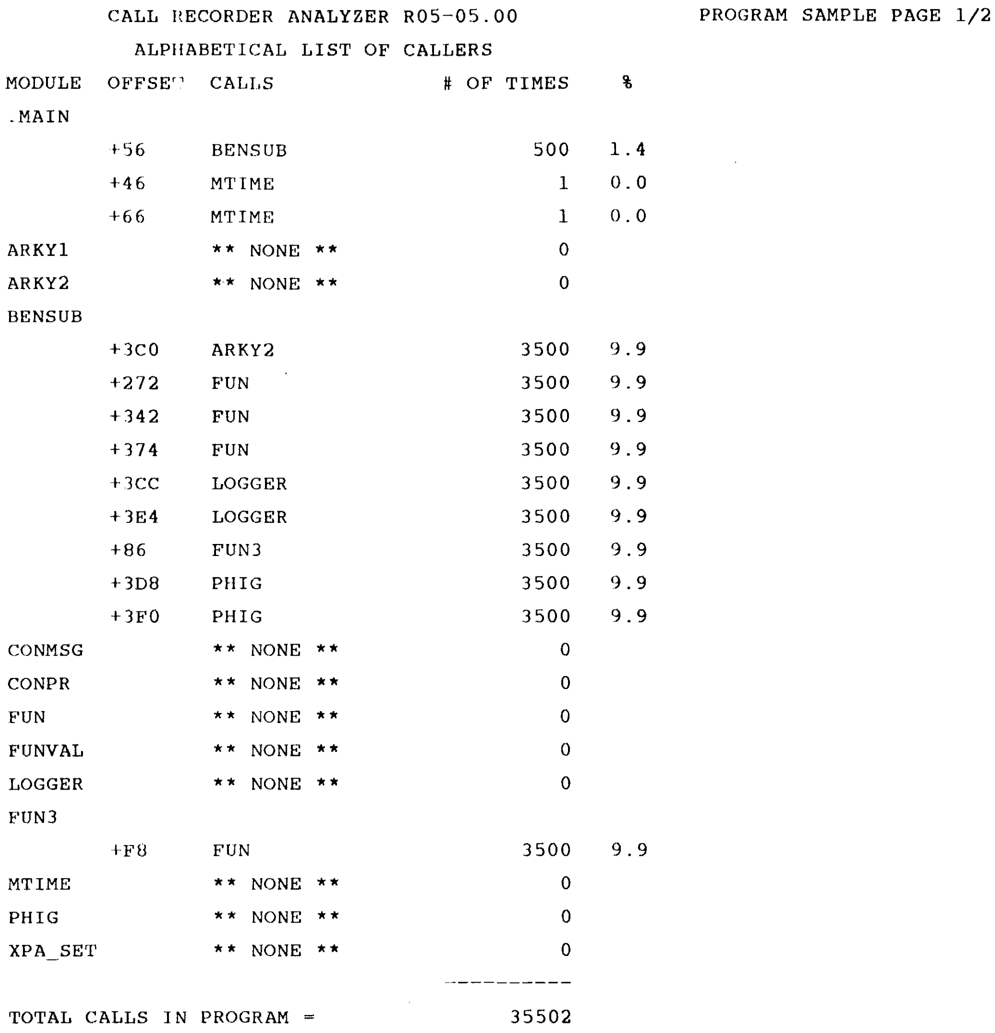
Figure 13-1 lists each entry point in the user program (1) sorted in order of total calls (2). With each entry point is a list of all the call sites given as a subprogram name (3) with an offset in hex (4). The offset is from the start of the pure section of the subprogram, or if the program has no pure section (NSEG), the offset is from the start of the subprogram. The number of times called from each call site is given (5) as well as the percentage in relation to the total number of calls in the entire user program (6). The total number of calls from each caller subprogram is given (7) as well as the percentage with respect to the rest of the program (8). The percentage of total calls with respect to the rest of the program is given (9). The total calls of all entries is given (10).

Figure 13-2 lists each subprogram in the user program sorted in alphabetical order (11). With each subprogram name is a list of all the entry points called by this subprogram (12), the offset of the call site (13), the number of calls (14), and the percentage with respect to the rest of the program (15). Finally, the total calls of all entries is given (16).

By concentrating on those subprograms which are called numerously, you can find the optimal candidates for in-line expansion. The offset given will help you locate which call is to be selectively expanded in a subprogram which has many different calls to a candidate module. Routines that are called only a few times, but consume considerable run-time from XPA, are potential candidates for optimization through replacement of a poor algorithm with a better one. Routines that are not called at all are candidates for elimination or are indicators that the test data is incomplete.

```
                CALL RECORDER ANALYZER R06-00.00                               PROGRAM SAMPLE PAGE 1
                   LIST OF CALLED ENTRIES SORTED BY TIMES CALLED
                   CALLED FROM                     FROM THIS MODULE
ENTRY NAME        MODULE  OFFSET  # OF TIMES    %  # OF TIMES    % TOTAL CALLS
FUN               BENSUB  +272          3500   9.9!                !
FUN               BENSUB  +342          3500   9.9!                !
FUN               BENSUB  +374          3500   9.9!    10500  29.6!
FUN               FUN3    +F8           3500   9.9!     3500   9.9!     14000   39.4
                                                  !                !
LOGGER            BENSUB  +3CC          3500   9.9!                !
LOGGER            BENSUB  +3E4          3500   9.9!     7000  19.7!      7000   19.7
                                                  !                !
PHIG              BENSUB  +3D8          3500   9.9!                !
PHIG              BENSUB  +3F0          3500   9.9!     7000  19.7!      7000   19.7
                                                  !                !
FUN3              BENSUB  +86           3500   9.9!     3500   9.9!      3500    9.9
                                                  !                !
ARKY2             BENSUB  +3C0          3500   9.9!     3500   9.9!      3500    9.9
                                                  !                !
BENSUB            .MAIN   +56            500   1.4!      500   1.4!       500    1.4
                                                  !                !
MTIME             .MAIN   +46              1   0.0!                !
MTIME             .MAIN   +66              1   0.0!        2   0.0!         2    0.0
                                                  !                !
CONPR             *** NOT INVOKED ***             !                !         0
                                                  !                !
ARKY1             *** NOT INVOKED ***             !                !         0
                                                  !                !
FUNVAL            *** NOT INVOKED ***             !                !         0
                                                  !                !
CONMSG            *** NOT INVOKED ***             !                !         0
                                                  !                !
XPA_SET           *** NOT INVOKED ***             !                !         0
                                                                    ----------
                                           TOTAL CALLS IN PROGRAM =      35502
```

Figure 13-1. Call Recorder Analyzer (CRA)



```
          CALL RECORDER ANALYZER R05-05.00           PROGRAM SAMPLE PAGE 1/2
            ALPHABETICAL LIST OF CALLERS
MODULE    OFFSET  CALLS                 # OF TIMES    %
.MAIN
          +56     BENSUB                       500   1.4
          +46     MTIME                          1   0.0
          +66     MTIME                          1   0.0
ARKY1             ** NONE **                     0
ARKY2             ** NONE **                     0
BENSUB
          +3C0    ARKY2                       3500   9.9
          +272    FUN                         3500   9.9
          +342    FUN                         3500   9.9
          +374    FUN                         3500   9.9
          +3CC    LOGGER                      3500   9.9
          +3E4    LOGGER                      3500   9.9
          +86     FUN3                        3500   9.9
          +3D8    PHIG                        3500   9.9
          +3F0    PHIG                        3500   9.9
CONMSG            ** NONE **                     0
CONPR             ** NONE **                     0
FUN               ** NONE **                     0
FUNVAL            ** NONE **                     0
LOGGER            ** NONE **                     0
FUN3
          +F8     FUN                         3500   9.9
MTIME             ** NONE **                     0
PHIG              ** NONE **                     0
XPA_SET           ** NONE **                     0
                                        ----------
TOTAL CALLS IN PROGRAM =                     35502
```

Figure 13-2. CRA

The CRATAB command substitution system (CSS) assigns all the necessary files for CRATAB. The format of the CSS call is:

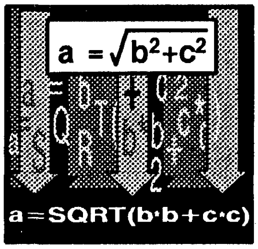
CRATAB <*taskname*>

Where:

<*taskname*>	is the name of the task being analyzed. The output is generated on file <*taskname*>.CTB which is also printed by the spooler PRINT command. CRATAB may be used without the CSS by assigning:

lu1 - <*taskname*>.CRA
lu3 - output
lu4 - <*taskname*>.MAP

The map file must exist on the same disk volume as the .TSK file. The map file must be the identical ADDRESS map that was used when the call recorder was run.



14

Floating Point Calculations

In this chapter

We discuss the limitations inherent in floating point calculations that produce inaccuracies in expected results. These inaccuracies may be due to a number of factors such as simple round-off error or the effects of optimization on floating point arithmetic.

Topics include:

- Floating point representation
- Different rounding techniques
- Floating point hardware
- Accuracy issues in the FORTRAN VII environment
- Effects of the run-time libraries (RTL)

Floating Point Representation

Floating point is a means of representing a quantity in any numbering system. For example, the decimal number 123 can be represented in the following forms:

$$123.0*10^0$$
$$12.3*10^1$$
$$1.23*10^2$$
$$.123*10^3$$

Notice how the decimal point moved in relation to the power of the factor 10. This is called the floating point. In actual floating point representation, the significant digits are always fractional and are collectively referred to as the mantissa. The factor with which the mantissa is multiplied by is called the base. And the power to which this base number is raised is called the exponent. For example, in the last representation of the decimal number 123, 123 is the mantissa, 10 is the base and 3 is the exponent. Both the mantissa and the exponent can be positive or negative. A floating point number is represented mathematically by the equation:

$$fpn=sgn*mn*(b^{ex})$$

Where:

sgn	plus or minus sign.
mn	mantissa
b	base
ex	exponent

Floating point representation in a computer is similar to the previous representation. The difference is that the hexadecimal numbering system is used instead of the decimal system.

The mantissa is a fraction whose value is less than 1 and greater than or equal to $1/b$. For b=16, this is the range $1/16<=$ *mantissa* < 1. The value of the mantissa is given by the formula:

$$mn=d1/b^1+d2/b^2+....+dn/b^n$$

Where:

dn represents hexadecimal digits and *n* is the total number of hexadecimal digits. For single precision, *n* is equal to 6 and for double precision, *n* is equal to 14.

In a computer, floating point numbers are represented by a string of bits divided into fields representing the sign, exponent, and mantissa. The machine representation is as follows:

sign	*exponent*	*mantissa*		
0	1-7	8	...	*n*

Where:

sign is the sign bit. '1' indicates a negative number and '0' indicates a positive value.

exponent is a 7-bit quantity representing the exponent in excess-64 notation. The number in this field contains the true value of the exponent plus X'40'.

mantissa is a string of 6 hexadecimal digits for single precision or 14 hexadecimal for double precision whose first digit is non-zero. Thus, *n* is equal to 31 for single precision and 63 for double precision.

Not all floating point numbers have exact hexadecimal machine representation. While real numbers are continuous, there are gaps between consecutive floating point numbers. The smallest step between two consecutive floating point numbers having the same exponent value is referred to as a 'granule'. The size of this gap is given by (b**(ex-X'40'))/(b**n), where *n* is 6, if single precision, or 14, if double precision. This limitation exists for all representational forms on binary computers due to the finite number of bits available. This explains the possible loss of precision when converting from decimal to hexadecimal values. To illustrate:

Decimal	**Hexadecimal**
.125	.20000000
.010	.028F5C28...

The previous example shows that the decimal number .125 has an exact hexadecimal representation whereas the decimal number .01 does not. Thus, if .01 is converted into a hexadecimal number having single precision floating point (SPFP), it loses precision from the seventh fractional position and up. In contrast, the decimal number .125 loses only zeroes.

Rounding Techniques

The manipulation of floating point numbers may require exponent equalization and normalization. In exponent equalization, the exponent of the operand with the smaller exponent value is incremented by one and the mantissa shifted four bits to the right until its exponent equals that of the other operand. Equalization is done before an addition or subtraction operation can be performed.

Normalization is performed when the most significant digit of the mantissa of a floating point number is zero. The number is normalized by repeatedly shifting the mantissa four bits to the left and decrementing the exponent by one until the most significant digit is nonzero. Normalization is done on the result of an arithmetic operation.

There are basically three approaches to rounding the results of a floating point calculation. These are truncation, jamming, and r-star rounding. Given two floating point numbers to be added, 41444444 and 40888888, the expected result is:

```
    41   444444
+   41   0888888    (after exponent equalization)
-------------------
    41   4CCCCC8
```

This is an exact result and rounds to 414CCCCD.

The following sections illustrate the above addition using each of the rounding techniques mentioned.

Truncation

Truncation is the simplest way to deal with the rounding problem. One just removes the low-order bits to get the proper number of digits and does the calculation with the remaining digits. The greatest drawback is that the absolute value of the average error is half a granule and this granule is always off in the same direction. The preceding example is performed using truncation as follows:

```
    41   444444
+   41   0888888   (after equalization and truncation)
------------------
    41   4CCCCC
```

Thus, accuracy is lost through the last hexadecimal digit. This is not a significant problem by itself, but accumulates with each additional calculation and can result in significant error.

Multiplication and division require no exponent equalization before the operation takes place, but the result must be six hexadecimal digits (single precision). Thus, any information in the least significant 6 digits of the 12-digit result are truncated and lost. A similar loss of accuracy occurs in double precision multiplication and division.

Jamming

Another method to deal with rounding is the jamming technique. This method requires that the least significant bit (LSB) of the result should always be a '1'. Jamming is less biased than truncation since there is an equal probability of the error being positive or negative, but the absolute value of the error still averages half a granule.

The previous addition is performed using the jamming technique.

```
    41   444444
+   41   0888888   (after equalization and truncation)
------------------
    41   4CCCCC    (C = '1100' - the last bit is a 0)
```

In this example, the last bit is set to a '1' giving the result 4CCCCD.

R-Star Rounding

The third approach is called r-star rounding. R-star rounding uses extra digits, called guard digits, obtained from the intermediate result of the arithmetic operation. These guard digits are used to decide which number is closest to the result. After the operation is performed and the intermediate result normalized, the remaining guard digits are inspected and the final result is chosen to be the number closest to the intermediate result. If the intermediate answer is exactly halfway between the representable floating point numbers, the larger number is generally chosen. The absolute value of the error is reduced to one quarter of a granule. No bias is introduced unless the value of the intermediate result is exactly halfway, in which case the result is biased towards the larger number.

This bias can be eliminated by using a technique called r-star rounding. This technique works the same as regular rounding except for the handling of results which are exactly halfway between the representable floating point numbers. The rules for r-star rounding, used by floating point hardware, are:

- If the most significant guard digit is X'7' or less, no rounding is performed.
- If the most significant guard digit is X'8' and all other guard digits are zero, the LSB of the mantissa is set to '1'.
- In all other cases, a '1' is added to the mantissa of the result.

The addition of 41444444 and 40888888 using the r-star method is as follows:

```
    41   444444
+   41   0888888        (after exponent equalization)
-------------------
    41   4CCCCC800000   (shifted byte is retained and padded
                        zeroes to serve as guard digits)
```

Since the most significant guard digit is 8 and the rest are all zeroes, from rule 2 above, the LSB of the mantissa (C = '1100') is set to 1. This gives a final result of 414CCCCD for the addition.

The following examples illustrate each of the rules for r-star rounding.

Examples:

Intermediate Result Guard Digit	Final Result (SP)	Comments
42B317E6 53010000	42B317E6	No change since 5 < 8
42B317E6 80000000	42B317E7	Force lowest bit to 1 since most significant guard digit is '8' and the rest are zeros.
42B317E9 80000000	42B317E9	No change. Lowest bit is already 1.
42B317E6 A4201310	42B317E7	Add 1 to mantissa since it is neither of the first two rules.

The error is still the same as for regular rounding, but the bias is eliminated using the guard digits.

Floating Point Hardware

There are two sets of floating point hardware, that offered with Series 3200 Processors and 8/32 Processors.

The 8/32 Processors perform truncation. Floating point calculations are truncated to the appropriate precision.

Series 3200 Processors' floating point hardware performs r-star rounding. This floating point hardware has two versions, one for the 3203/3205 processor and one for other Series 3200 Processors (e.g., 3230, 3250, ...). They differ only in how they keep track of the information necessary to perform the rounding. The 3203/3205 and the 3280 processors are functionally equivalent and perform the same r-star rounding technique.

The 3203/3205/3280 hardware carries out the calculations to full precision and uses this full precision to round. For example the addition of Y'46445566' and Y'41886644' becomes:

```
     46   445566
+    46   00000886644
----------------------
     46   44556D86644    which rounds to CW46 44556E
```

with the 'D' being rounded up to an 'E'.

Rather than carrying all the extra digits, the other Series 3200 Processors use one guard digit and one sticky-bit to perform the rounding. When the mantissa is shifted downward to equalize the exponents, the shifted digits go through the guard digit, to the sticky-bit, and are then lost. The sticky-bit keeps track of the effect of these lost digits. This bit, initially zero, is set to one and stays one if any of the lost bits were '1' and remains zero if only '0's were shifted out. Keeping track of the lost bits through the sticky-bit allows r-star rounding to be performed without carrying along the extra bits associated with the operation. The addition of Y'41886644' and Y'46445566' becomes:

```
     46   445566
+    46   000008 8 '1'
-------------------
     46   44556E
```

In this case, the least significant digit of the second addend is an '8', the guard digit is an '8', and the sticky-bit is set to '1', indicating that the lost bits were not all zeroes'. Thus, a '1' is added to the least significant digit of the result changing the 'D' to an 'E'.

Lost Precision in Floating Point Arithmetic

As an example of a round-off error associated with floating point calculations, consider the following code.

```
      REAL YADD,YMULT,EX
      EX = 0.1
      YADD = 0.0
      YMULT = 0.0
      DO 17 I = 1,1024
   17 YADD = YADD + EX
      YMULT = EX * 1024.0
      WRITE(5,'(X,Z,X,Z)')YADD,YMULT
      END
```

This code adds EX repeatedly, 1024 times, and then computes the product EX * 1024.0, both of which may be expected to yield the same result. The results, however, show YADD = Y'426667C4' and YMULT = Y'42666668', which shows a discrepancy of X'15C' in the least significant places.

This discrepancy is a result of accumulated round-off error. It is acquired as follows. The single precision hexadecimal equivalence of the decimal value 0.1 is Y'4019999A'. The result of the first nine additions of this series is precisely Y'40E6666A' as no rounding yet occurred. The addition of the tenth item yields:

```
     40   E6666A
+    40   19999A 8 '1'
----------------------
     46   44556E
```

After the tenth addition, there is no loss of precision. However, the next addition results in the following:

```
     41   100000
+    41   019999A
-----------------
     41   119999A   which is 41 11999A after rounding
```

By rounding the last digit of the mantissa, this intermediate result is larger by 6/16 granules in the least significant hexadecimal place. These rounding errors continue until the fiftieth sum. At this point, the partial result is Y'41500010' rather than the infinite precision result of Y'41500000'. After the 160th addition, the result is Y'42100004', which is 4 granules larger than the infinite precision result of Y'42100000'. In the remaining 864 additions, the smaller addend has to be shifted down two places. In each addition, the result is rounded up and the effect is that each step is larger by 102/256 granules in the least significant hexadecimal place. This leads to an additional 158 granules of error in the last 864 additions. Together with the 4 granules from the first 160 additions, the total error is X'15C' as noted previously.

Accuracy Issues in the FORTRAN Code

Aside from the inaccuracies that arise from round-off errors, other factors can affect the results of calculations such as the processor on which the compiler and task run, Series 3200 Processors versus the 8/32 Processor. The following sections describe the general behavior of floating point arithmetic and type conversions in FORTRAN. The effects of the order of evaluation of expressions are discussed in the section entitled "Optimization and Order of Evaluation Effects."

FORTRAN performs arithmetic in three different modes: integer, SPFP and double precision floating point (DPFP). The last two follow the same rules and differ only in the number of digits involved in the calculations.

Integer Arithmetic

Integer arithmetic is always exact as integers are exactly representable and the results of all integer operations are forced to integers. The only limitation is the range of representable integers, -2147483648 to +2147483647, allowing for integers of up to eight hexadecimal digits. An overflow condition occurs if the result of an integer operation falls outside this range. This is not flagged at compile-time and does not result in run-time error.

FORTRAN allows an integer to be assigned to a floating point variable. However, not all integer values are exactly representable as floating point numbers. This is true for SPFP representations since large integers can contain up to eight hexadecimal digits while the mantissa portion of a single precision number contains only six hexadecimal digits. In cases where an integer value having more than six hexadecimal representation is assigned to a SPFP variable, the assignment involves rounding of the integer value to six hexadecimal places. For example, if the decimal number 72788941 (X'456ABCD') is assigned to a SPFP variable, the internal representation of the floating point number is X'47456ABD'. Notice that the value is forced to a six hexadecimal number after rounding the least significant digit.

DPFP numbers contain fourteen hexadecimal digits and can thus represent all integers exactly.

Integers may also be assigned the result of a floating point expression. For example, INTA = 7.0/4.0 results in the integer value '1' being assigned to INTA. This assignment is a FORTRAN language definition and is not an inaccuracy.

Floating Point Arithmetic

The following discussion on floating point arithmetic is broken into two parts. The first section illustrates straight assignment between floating point types as performed by the optimizing compiler. The second section examines the differences that may occur depending on whether the evaluation of an expression occurs at compile-time or at run-time and whether the target processor is a Series 3200 Processor or 8/32 Processor.

Simple Assignment

Some care must be taken when mixing types in floating point expressions. A REAL*8 variable can be assigned a REAL*4 value and vice-versa. In addition, the intrinsic subprograms SNGL or DBLE may be called to perform type conversions. FORTRAN defines these type conversions as illustrated by the following examples.

The following programs perform type conversions between SPFP and DPFP types and print the results in hexadecimal and decimal notation. In all four examples, compilation is performed using the optimizing compiler and the variables EX and Y are declared as REAL*4 and REAL*8, respectively.

Example 1:

```
EX = 0.05E0
Y = DBLE(EX)
WRITE(5,'(X,2Z,X,F10.8,X,D20.14,X)') EX,Y,EX,Y
```

Output:

```
3FCCCCCD 3FCCCCCD00000000 0.05000000 0.50000000745058D-01
```

In Example 1, a single precision constant is assigned to a single precision variable, EX, and the function DBLE is used to convert EX from single to double precision to be assigned to Y. The decimal number 0.05E0 is represented as the repeating number 0.0CCCCCCCC... in hexadecimal. The assignment of 0.05E0 to EX results in the hexadecimal number 3FCCCCCD being assigned to it. Note that the assigned value is rounded to obtain the closest possible hexadecimal number to the exact value. The intrinsic function DBLE converts EX to double precision and this value is assigned to Y. The result is 3FCCCCCD00000000. The function DBLE loads the most significant fullword of the double precision value from the single precision value and pads the least significant fullword of the result with zeros. This assignment does not regain the loss of precision from the assignment EX = 0.05E0. Even though it is apparent to the compiler that 0.05 is being assigned to Y, neither the compiler nor the function DBLE can assume that the value is 0.05 in double precision. Also, the double precision decimal output is not exactly 0.05. It is 0.05 to single precision accuracy, but since DBLE pads the lower fullword with zeros, the resulting value is not exactly 0.05 but a slightly larger number.

Example 2:

```
EX = 0.05D0
Y = DBLE(EX)
WRITE(5,'(X,2Z,X,F10.8,X,D20.14,X)') EX,Y,EX,Y
EX = SNGL(0.05D0)
Y = DBLE(EX)
WRITE(5,'(X,2Z,X,F10.8,X,D20.14,X)') EX,Y,EX,Y
```

Output:

```
3FCCCCCC 3FCCCCCC00000000 0.05000000 0.49999997019768D-01
3FCCCCCC 3FCCCCCC00000000 0.05000000 0.49999997019768D-01
```

In Example 2, a double precision value, 0.05D0, is assigned to EX and the function DBLE is invoked to convert this value to double precision before assignment to Y. In line one, the double precision value is directly assigned to EX. In line four, the intrinsic function SNGL is used to convert the double precision value to single precision before assigning the result to Y. The results show that the assignment of a double precision value to a single precision variable may lose some precision and that reassigning the resultant value to a double precision variable does not regain this lost precision.

Example 3:

```
Y = 0.05E0
WRITE(5,'(X,Z,X,F10.8,X,D20.14,X)') Y,Y,Y
```

Output:

```
3FCCCCCD00000000 0.05000000 0.50000000745058D-01
```

Example 3 produces the same results for Y as does Example 1. 0.05E0 is a single precision value whose hexadecimal equivalent, 3FCCCCCD, is loaded into the most significant fullword of Y while the least significant fullword is padded with zeros. Again, the decimal output shows that the result is exactly 0.05 to single precision accuracy and a little larger than 0.05 to double precision accuracy.

Example 4:

```
Y = 0.05D0
WRITE(5,'(X,Z,F10.8,X,D20.14,X)') Y,Y,Y
```

Output:

```
3FCCCCCCCCCCCCCD 0.05000000  0.50000000000000D-01
```

Example 4 results in 0.05 being assigned to Y to double precision accuracy. The decimal output shows that the result is exactly 0.05 to both single precision and double precision accuracy.

Compile-Time and Run-Time Evaluation

As mentioned before, there are basically two sets of floating point hardware, those associated with Series 3200 Processors and those associated with the 8/32 Processors. The FORTRAN compilers evaluate all compile-time constants in 8/32 compatible mode regardless of where the compiler is running.

Using the floating point instruction set associated with the Non-Series 3200 processors to compute compile-time constants allows the compilers to run on all processor equally well. The larger instruction set of the Series 3200 Processors support floating point instructions which perform r-star rounding. Thus, differences between run-time and compile-time evaluations may occur as illustrated in the following example.

```
REAL*8 PFIVE
X1=SNGL(0.05D0)         ;1
PFIVE=SOMEHOW(0.05D0) ;
X2=SNGL(PFIVE)          ;2
X3=ESNGL(PFIVE,SNGL)    ;3
END

REAL FUNCTION ESNGL(D,F)
REAL*8 D
ESNGL=F(D)              ;3
END

REAL*8 FUNCTION SOMEHOW(D)
REAL*8 D
SOMEHOW=D
END
```

1. computed at compile time
2. compiler generates inline code for this intrinsic function
3. RTL routine is called for this case

The results as a function of host and target processor are given in Table 14-1.

Host	Target	X1	X2	X3
8/32	8/32	3FCCCCCC	3FCCCCCC	3FCCCCCC
8/32	3200	3FCCCCCC	3FCCCCCD	3FCCCCCD
3200	8/32	3FCCCCCC	3FCCCCCC	3FCCCCCC
3200	3200	3FCCCCCC	3FCCCCCD	3FCCCCCD

Table 14-1. Values of X1, X2, and X3 as a Function of Host and Target Processors

The FORTRAN statement X1 = SNGL(0.05D0) results in assigning 3FCCCCCC to X1 since the 8/32 instruction LE is performed rather that the LED/LEDR instruction available with Series 3200 Processors. If SNGL is declared as an external or invoked as in case 3, then the RTL is used and a process running on an 8/32 uses the LE and one running on a 3200 uses an LED instruction. If the routine SNGL is performed inline (case 2) rather than at compile time then the LE instruction is generated for an 8/32 target and the faster LED instruction is generated for a Series 3200 Processors target. If an assignment like X1 = 0.05D0 is performed, then, at all times, the LE instruction is used.

When converting a floating point constant to an internal representation the FORTRAN compilers assume all such constants are double precision. A SPFP constant is thus treated as double precision when converted to an internal representation and is rounded to single precision as the final step of the conversion process.

Optimization and Order of Evaluation Effects

The FORTRAN VII optimizing compilers provide two classes of optimizations, built-in and optional. The built-in optimizations are performed on input code and cannot be disabled. The optional optimizations are controlled by the OPTIMIZE/NOPTIMIZE compiler directives. Chapter 2 discusses these optimizations in detail.

While these optimizations produce object code that executes faster, the presence or absence of these optimizations may produce slight variations on the results under certain conditions. The possible influence on floating point calculations by these two classes of optimizations are discussed below.

The optimizing compilers attempt to evaluate as much of the code as possible at compile time rather than at run-time to save execution time and memory space. The optimizations performed include constant computation, type conversion, symbolic arithmetic, machine instruction choice, and strength reduction. In general, these types of optimizations have no effect on the results; but if the order of evaluation of an expression greatly affects the results, then the expression should be parenthesized to force a particular order of evaluation. Thus, the source code T=16.*S/2.+(8.0-3), which is optimized to T=8.*S+5, should be written as T=(16.*S)/2.+(8.0-3) if it is absolutely necessary to multiply S by 16 before performing the division. Such use of parentheses inhibits optimization and its overuse can degrade run-time speed.

Other built-in optimizations which may affect the results of floating point calculations are the evaluation of symbolic arithmetic expressions, expression reordering, and strength reduction. Examples of optimizations which may be performed are shown in the Table 14-2.

Code Before Compilation	Optimized Code
J*I+K*I	(J+K)*I
I*K1+J*K2 WHERE K=GCF(K1,K2	([K1/K]*I+[K2/K]*J)*K
I**K*J**K	(I*J)**K
I**J*I**K	I**(J+K)
K1**I*K2**J WHERE K1=2***K*K2	2**(K*I)*K2**(I+J)
A*A	A**2
A**X*A	A**(X+1)
A*A**X	A**(X+1)
A**X/A	A**(X-1)
A/A**X	A**(1-X)
A/C+B/C	(A+B)/C
1/B*A	A/B
X**2+X*((X*2)+X)	(X+K)**2
(A*B)/(A*C)	B/C
(A/B)/(C/D)	(A*D)/(B*C)
(A/B)*(C/D)	(A*C)/(B*D)
SQRT(A)*SQRT(B)	SQRT(A*B)
A**.5	SQRT(A)
A**.25	SQRT(SQRT(A))
X/RK	X*(1/RK) RK IS POWER OF 2
A*X**K+B*X**(K-N)	(A*X**N+B)*X**(K-N)
A*X**K+B*X	(A*X**(K-1)+B)*X
A*X**K+X**(K-N)	(A*X**N+1)*X**(K-N)
X**K+B*X**(K-N)	(X**N+B)*X**(K-N)
X**K+B*X	(X**(K-1)+B)*X
X**K+X**(K-N)	(X**N+1)*X**(K-N)
A/A	1
A*X+A	A*(X+1)
X**RK WHERE INT(RK) == RK	X**INT(RK)

Table 14-2. Examples of Symbolic Arithmetic Performed By the Optimizing Compilers

The optional optimizations include other strength reductions, invariant code motion, constant propagation and computation, scalar propagation, and common subexpression elimination. These techniques are discussed in detail in Chapter 3.

For discussion purposes, consider the following example using the common subexpression elimination.

```
A = C + D + B
C = B + C
```

The optimized version of the code is:

```
@100 = B + C
A = @100 + D
C = @100
```

The compiler recognizes the common subexpression B + C and computes this value which is then used in the assignment statements. @100 is a temporary variable generated by the compiler.

If the NOPTIMIZE compiler option is specified, common subexpression optimization is turned off. This may result in a different order of evaluation for the expression assigned to A. The FORTRAN development compiler may evaluate the operands in a third order which may lead to a third slightly different answer. Thus, while C + D + B mathematically evaluates to one particular result, under some conditions, each different compilation method may result in a slightly different answer. If an application is very sensitive to small variations in the intermediate results, as small as one granule, then parentheses should be used in the expression to enforce the desired order of evaluation. In addition, parentheses should be used if it is necessary to get very close results from the use of both the development and optimizing compilers.

Possible Effects of the FORTRAN RTLs

The FORTRAN language system contains a number of RTLs with which the compiled source code may be linked. The RTLs may be either writeable control store (WCS), where the routines are written in microcode, or non-WCS, with the routines generally written in assembly code, and may be either argument-checking or nonargument checking.

For a particular RTL, either WCS or non-WCS, there is no computational difference between the argument checking and nonargument checking versions. The argument checking RTLs check to see if the arguments to the RTL routines are of the correct type and class and that the correct number of arguments are passed. All other computations are the same. Additionally, the functionality of both the assembly RTL and the WCS RTL is exactly the same.

The WCS has a different instruction set than the non-WCS and the floating point instructions allow greater accuracy than available in assembly code. To take advantage of this, some of the WCS routines may use slightly different algorithms than their non-WCS counterpart. In general, all single precision routines return values with a relative accuracy better than 1.0E-7 and all double precision routines return values with a relative accuracy better than 1.0E-16. A statement of the relative accuracy of a computation is based on the assumption that the numerical data, such as the argument of a function, is perfectly accurate. See the *System Mathematical Run-Time Library (RTL) Reference Manual* for a discussion on relative accuracy. It should be noted that the WCS RTLs are limited by the size of the WCS available on a particular machine. Therefore, not the entire RTL is written in microcode, but only some commonly used transcendental and involution routines. A list of the microcoded routines can be found in the the *Series 3200 Processors FORTRAN Enhancement Package (FEP) Reference Manual.*

Given these inherent differences, it may be expected that the WCS and non-WCS versions of the RTL may give a slightly different answer for a specific operation. For example, given two specific floating point numbers *fp*1 and *fp*2, then the result of *fp*1***fp*2 obtained using the WCS RTL may be different from that obtained using the non-WCS RTL. This is generally not the case, for the vast majority of input values the WCS and non-WCS RTLs provide exactly the same answer for the given operation. Even in those cases where the computed value differs from the mathematically correct value by a granule or two, the WCS and non-WCS routines usually return the same slightly inaccurate answer. In cases where the WCS and non-WCS RTLs provide different answers, the WCS RTL is generally more accurate.

It is possible to set up conditions where the differences are noticed. For example, if two tasks are communicating via task common with one task using the non-WCS RTL and the other task using the WCS RTL then occasional small differences may occur. Should comparisons for exact equality, rarely a good idea for floating-point numbers, be made between numbers from the differing tasks, then discrepancies might again be noticed.

There are two WCS RTLs, one for the 3230 and one for the 3250. The storage allotted to WCS is larger on the 3230 than on the 3250. This leads to the difference in the WCS RTLs for these machine. The 3230 WCS RTL contains a routine, RXXR, which specifically performs the exponentiation of a single precision base to a single precision power. The 3250 WCS RTL does not have the RXXR but instead uses the DXXD, which performs the exponentiation of a double precision base to a double precision power. There is a loss of precision on the 3230 WCS RTL since it makes use of single precision quantities when computing exponentiation.

As a result of differences between the 3230 WCS and the 3250 WCS, the discrepancies between WCS and non-WCS RTLs and the differences between compilers, several situations exist where normal use can lead to inconsistencies:

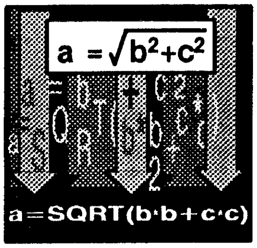
1. Results from repeated runs of a single task on a 3260MPS under the load leveling executive (LLE). Different processors may be used for different parts of a program.
2. Results of runs of the same tasks on a 3260MPS where one run is central processing unit (CPU) directed and the other is auxiliary processing unit (APU) directed. Results on the same processor will be consistent but may differ between processors.
3. Results of tasks compiled under different compilers.
4. Tasks communicating via task common where either 2 or 3 above come into play or one task uses WCS and the other does not.

Summary

This chapter addressed the limitations inherent in floating point calculations. It presented the various factors which may affect the accuracy of floating point arithmetic. These factors are:

- round-off errors,
- conversion between types (i.e., integer to real),
- optimizations performed on the source code,
- the order in which calculations are performed, and
- conversion between decimal and hexadecimal representations.

These factors may in turn be affected by the user's code, the user's data, the FORTRAN compiler and RTL used, and even the hardware on which the task is run. These considerations were discussed in the hope of helping the users understand unexpected results when using the FORTRAN language system.



15

Universal Optimization

In this chapter

We present the different phases of optimization performed by the FORTRAN VII compilers. The process involves the gathering and use of information for optimizing code across program unit boundaries. Optimization under the F7Z compiler provides the FORTRAN programmer with a semiautomatic method for improving program performance, while retaining the benefits of program modularity; i.e., ease of development, debugging, and maintenance.

Topics include:

- Comparison of the optimization methods
- FORTRAN VII compilation phases
- F7Z in-line expansion feature

Comparing the Optimization Methods

The use of high-level languages in place of assembly language for implementing time critical applications depends on the quality of code generated by the compiler. Because a statement-by-statement translation of the user's source program often produces inefficient object code, other compilation methods must be used to generate code suitable for time critical applications. These methods involve gathering information from the user's program that can be used to rearrange and modify the original source into an optimized version that executes more efficiently. Different compilation methods provide different levels of optimization dependent upon the information gathered and used by the compiler.

According to the level of optimization provided, compilers can be classified as follows:

- Statement optimizers
- Block optimizers
- Global optimizers
- Universal optimizers

In general, each level of optimization includes the optimization capabilities of the more primitive levels.

Statement Optimizers

Statement optimizing compilers scan each statement to determine whether or not it can be rearranged to minimize the use of temporary storage, decrease code size, and increase execution speed. This type of code rearrangement is more commonly known as smart code generation. Most high-level language compilers provide this level of optimization.

Block Optimizers

Block optimizing compilers scan blocks of code within a program to determine where code can be rearranged to eliminate redundant computation of expressions and minimize references to memory. A block of code consists of a number of statements that are executed sequentially until a branch statement (IF, GOTO, computed GOTO, etc.) or the target of a branch statement (labeled statement) is reached. Some block optimizing compilers also optimize special constructs such as DO loops.

Global Optimizers

Global optimizing compilers perform a complete analysis of the data flow within each separately compiled program unit. The compiler uses the information obtained from this analysis to optimize the entire program unit. Even though some block optimizers that optimize special constructs are occasionally marketed as global optimizers, only compilers which perform a complete data flow analysis of each program unit are considered to be true global optimizers. The FORTRAN VII O compiler is a global optimizing compiler.

Universal Optimizers

A universal optimizing compiler gathers information for optimization across program unit boundaries. This information can be used to enhance global optimization in the following ways.

- Program units can be merged; i.e., calls to subprograms can be expanded within the calling program and the resulting code can be optimized as a single unit. The F7Z compiler provides this level of optimization.
- Subprogram interface information, gathered by the compiler on the effects of a particular call on the arguments of the CALL and/or COMMON statements, can be used by the global optimization routines to further optimize the code in the calling program. This allows the compiler to overcome the limits imposed on optimizing separately compiled units without in-line expansion.

Phases of the F7O and F7Z Compilation

The operation of the F7O and F7Z compilers are divided into several phases. Each phase makes at least one complete pass through the user program (source or internal tables).

When the F7Z compiler is being used, Phases 1 and 2 are repeated for each source subprogram for which in-line expansion is requested.

Some of the phases are optional. Phases 5 and 6 are invoked only if optimization is desired. Phase 3 is invoked only when the F7Z compiler is being used and in-line expansion is requested.

Phase	Description
0	Table initialization.
1	Creates the program table (P-table) and all other parse related tables; outputs the source listing indicating any errors that were detected.
2	Detects errors on labels, equivalence statements, and DO loops; generates an optional cross-reference listing of variables and labels.
3	Performs in-line expansion of subroutines creating a compound internal table; produces an extended listing, if requested. (Phase 3 is an optional phase on the F7Z compiler if in-line expansion was requested.)
4	Completes semantic structure for some statements (i.e., logical IF), array linearization and statement label transformation; optionally creates a cross-reference for the extended listing on the F7Z compiler.

Table 15-1. Phases of the F7O and F7Z Compilation

Phase	Description
5	Optional; globally optimizes the user's program by building a thorough flowchart of the program; examples of optimization are: common subexpression elimination, invariant code motion, constant propagation, etc. Various optimizer messages are produced for user information.
6	Always follows Phase 5; transforms machine independent optimizations to machine dependent operations; this improves the quality of the object code. Examples are: strength reductions of integer multiply, exponentiation operations, and global register assignment.
7	Interprets the intermediate language and decides which machine instructions to use; it must correctly interpret this P-graph whether or not optimization is chosen; that is, whether Phase 5 and Phase 6 are executed or skipped.
8	Transforms the intermediate language into one or two forms: • An assembly-form suitable for the Common Assembly Language (CAL) assembler, or • A squeezed object-form suitable for input into OS/32 Library Loader or Link. An assembly listing can be optionally produced.

Table 15-1. Phases of the F7O and F7Z Compilation (Continued)

Figure 15-1 presents the operational phases of the F7O and F7Z compilers in flowchart form.

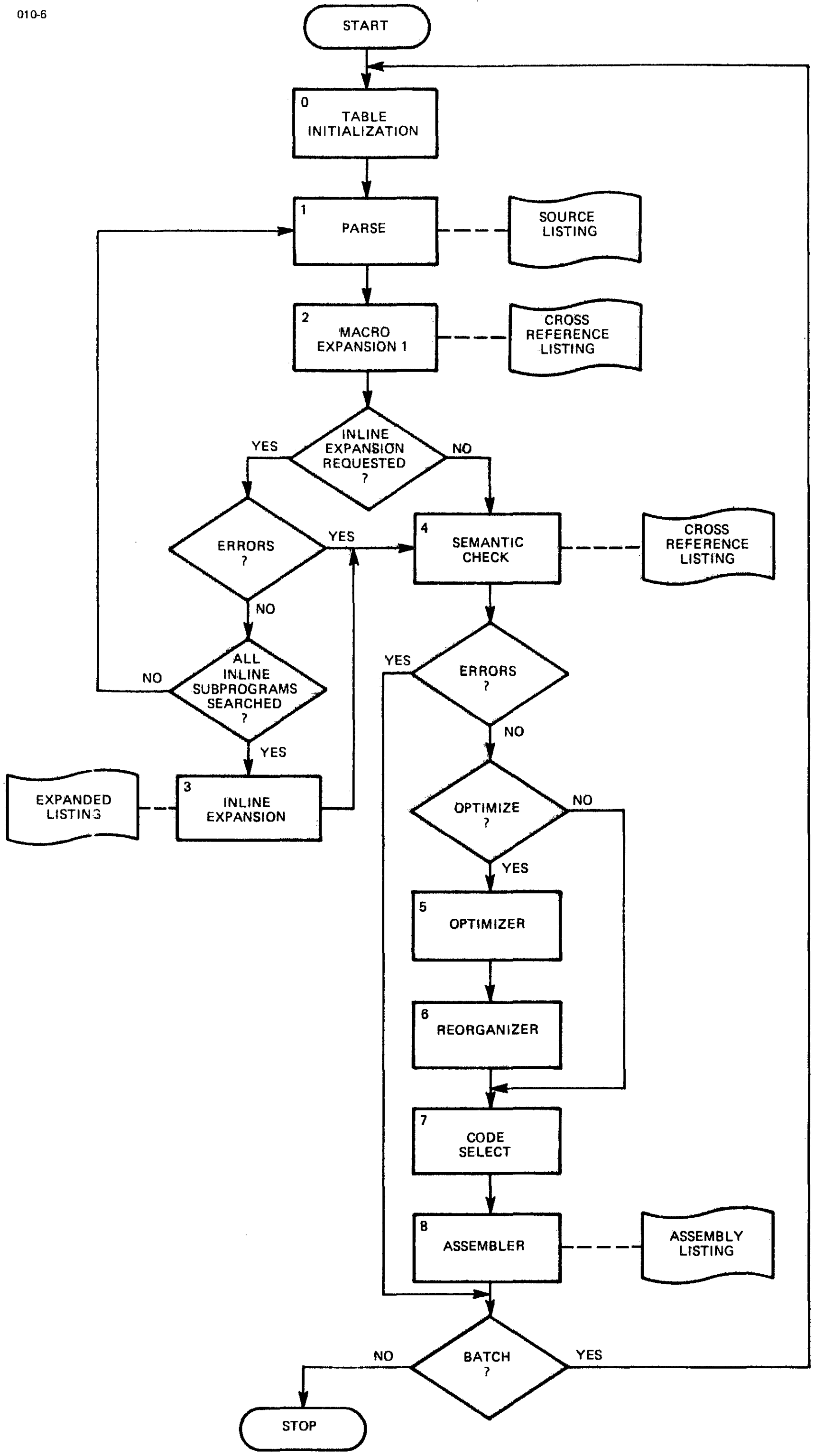


Figure 15-1. F7O and F7Z Compilers Flowchart

Illustrating the Use of In-line Expansion

The following example demonstrates the level of optimization that can be achieved with in-line expansion.

Example:

```
C THIS EXAMPLE SHOWS HOW OPTIMIZATION CAN BE
C SIGNIFICANTLY IMPROVED WHEN A SUBPROGRAM IS
C EXPANDED WITHIN A CALLING PROGRAM UNIT
C
      SUBROUTINE POLEV (AR,X,N,F)
      INTEGER X
      REAL AR(6)
      GO TO (1,2,3,4)N
1     F=AR(1)*X + AR(2)
      RETURN
2     F=(AR(1)*X + AR(2))*X + AR(3)
      RETURN
3     F=((AR(1)*X + AR(2))*X + AR(3))*X + AR(4)
      RETURN
4     F=(((AR(1)*X + AR(2))*X + AR(3))*X + AR(4))*X + AR(5)
      RETURN
      END
```

When POLEV is called with the statement:

```
CALL POLEV (EP,MACH,3,EPSO)
```

and POLEV is expanded in-line, the F7Z compiler replaces this statement with the following code and uses EP, MACH, 3, and EPSO as arguments.

```
      POLEV.X=MACH
      POLEV.N=3
      POLEV.F=EPSO
      GO TO($L001,$L002,$L003,$L004),POLEV.N
$L001 POLEV.F=EP(1)*POLEV.X + EP(2)
      GO TO $L005
$L002 POLEV.F=(EP(1)*POLEV.X + EP(2))*POLEV.X + EP(3)
      GO TO $L005
$L003 POLEV.F=((EP(1)*POLEV.X + EP(2))*POLEV.X + EP(3))*POLEV.X +
     1       EP(4)
      GO TO $L005
$L004 POLEV.F=(((EP(1)*POLEV.X + EP(2))*POLEV.X + EP(3))*POLEV.X
     1       + EP(4))*POLEV.X + EP(5)
$L005 CONTINUE
      EPSO=POLEV.F
      MACH=POLEV.X
```

The optimizer can now perform the scalar and constant propagation on the expanded program unit using the arguments passed by the CALL statement. It can also perform dead code elimination on the unit. These routines yield the following code:

```
      GO TO($L001,$L002,$L003,$L004),3
$L001 POLEV.F=EP(1)*MACH + EP(2)
      GO TO $L005
$L002 POLEV.F=(EP(1)*MACH + EP(2))*MACH + EP(3)
      GO TO $L005
$L003 POLEV.F=((EP(1)*MACH + EP(2))*MACH + EP(3))*MACH + EP(4)
      GO TO $L005
$L004 POLEV.F=(((EP(1)*MACH + EP(2))*MACH + EP(3))*MACH + EP(4))*
     1       MACH + EP(5)
$L005 CONTINUE
      EPSO=POLEV.F
```

This code is further reduced by the computation of the computed GOTO, the elimination of dead code, and the propagation of the value of POLEV.F into the assignment of EPSO. These optimizations yield the following code:

```
$L003 EPSO=((EP(1)*MACH + EP(2))*MACH + EP(3))*MACH + EP(4)
```

Still further code reduction may be obtained by propagating the values of MACH and the elements of EP if additional uses of these scalar values exist in the calling program unit.

As shown by the preceding example, optimization through in-line expansion does more than eliminate the subprogram linkage operations. It greatly increases the number of global optimizations applied to a program by allowing them to be performed across program unit boundaries.

How F7Z Performs In-line Expansion

When a program requesting in-line expansion is compiled, certain processes are performed by the F7Z compiler before global optimization takes place. First, F7Z produces a standard listing of the main source program and a cross-reference listing, if requested. At this stage the program is translated into an intermediate (internal) code. Then, the subprograms designated for in-line expansion by the main program are located, read into the compiler, and translated into intermediate code. For each of these subprograms, a separate standard listing and cross-reference listing is produced, if requested.

The compiler locates the source code of a subprogram in one of two ways:

- It parses the first statement of each subprogram in a user-specified file until the subprogram being searched for is found.
- It searches a user-specified source file until a module delimiter for that subprogram is found. See Chapter 3 on $INCLUDE for more information on module delimiters.

If the compiler encounters a module delimiter for the first subprogram in the file, it will not use the parsing method. In this case, each of the subprograms in the source file must have a module delimiter. While the use of module delimiters requires more preparation on the part of the programmer, this method is faster than parsing.

If additional requests for in-line expansion are made by the subprograms that are expanded in-line, the compiler performs the same operations for these requests as it did for those made by the main program. If the source code of a subprogram that is to be expanded in-line cannot be located by the compiler, a message is sent to the list device. Compilation is continued until all requests for in-line expansion are satisfied. A fatal error occurring any time during this compilation stage causes F7Z to terminate processing with an end of task code 4. Otherwise, F7Z begins expanding the requested subprograms at the specified calls within the calling program. Diagnostic messages or warnings are sent to the list device for each argument type mismatch encountered.

After all requested subprograms are expanded, the compiler produces a listing of the source code as it appears after in-line expansion, but before optimization. This listing is called the extended listing. All subsequent listings and messages from the compiler refer to this extended listing.

NOTE In-line expansion is not performed by default. It must be explicitly requested by the user.

Intermediate Code Translation

After all designated subprograms are expanded in-line, F7Z rearranges and modifies the intermediate code of all the subprograms. This intermediate code is then translated and represented on the extended listing as follows.

- Data type specification - the data type of all variables and functions used by the program and subprograms, including those not declared by the user, are explicitly declared by the compiler at the beginning of the program.
- FORMAT statements - all FORMAT statements occurring in all the subprograms are grouped together and located above the first executable statement in the main program.
- Comments - the compiler removes all trailing comments following a statement and any comments located between continuation lines. All other comment lines are retained.
- DO loops - the compiler generates a CONTINUE statement with a compiler generated label for each DO loop that ends on a statement other than a CONTINUE statement or shares a terminal statement with another DO loop.
- Compiler generated labels - the compiler generates unique labels for all FORTRAN labels in a subprogram each time the subprogram is expanded in-line. Compiler generated labels, which have the form $Ln where n is a compiler generated number, prevent multiple definitions of labels that can result from repeated expansions of a subprogram.
- Compiler generated variables - the compiler replaces all statement function dummy arguments and DATA implied DO indexes with compiler generated variables. The format of these variables is @*n*, where *n* is a compiler generated number.

- Renaming of program variables and functions - all occurrences of local, dummy, and common variables in in-line expanded subprograms are prefixed with the first eight characters of the subprogram name followed by a period. This is also done for FORTRAN symbols used in an embedded CAL block.
- Compiler generated EQUIVALENT COMMON - the compiler generates an EQUIVALENT COMMON statement for all common blocks referenced by both the calling program and the expanded subprogram.

  The EQUIVALENT COMMON statement declares a list of variables starting at the same address as does the regular common with the same name.

  Example:

  ```
  COMMON A,B,C
  EQUIVALENT COMMON SUB1.A,SUB1.B,SUB1.C
  ```

- Compiler generated SAVE and COMMON statements - local variables specified in a SAVE statement are promoted by the compiler into COMMON. The common block name for this common is generated by truncating the name of the subprogram to seven characters and appending a dollar sign ($) to it. For example, the statement:

  ```
  SAVE  L1, L5, L3
  ```

  in the main program is represented in the extended listing as:

  ```
  COMMON/.MAIN$/L1, L5, L3
  SAVE/.MAIN$/
  ```

- Expansion of a function in an input/output (I/O) list - when a function call in an I/O list is expanded, the I/O statement is broken down into its component parts as shown in the following example.

Example:

```
C THIS PROGRAM UNIT REQUESTS INLINE EXPANSION
C OF A FUNCTION WITHIN AN I/O LIST.
$INLINE F,*
      X=1
      WRITE (*,10)X,F(X),X
10    FORMAT(1X,3F4.0)
      STOP
      END

      FUNCTION F(Y)
      Y=Y + 2
      F=Y
      RETURN
      END
```

Intermediate code translation of this program yields the following code:

```
      PROGRAM.MAIN
      REAL F.Y, F.F, X,@100
10    FORMAT (1X,3F4.0)
      X=1.0
      WRITE(*,10)
      WDATA X
      F.Y=X
      F.Y=F.Y + 2.0
      F.F=F.Y
      X=F.Y
      @100=F.F
      WDATA @100
      WDATA X
      IOFIN
      STOP
      END
```

Argument Passing for In-Line Expanded Subprograms

When a subprogram is expanded in-line, all arguments passed to and from the subprogram must be defined as they were originally intended to be defined by the user's program. Except for expansions involving embedded assembly code or the retention of local variables across calls to the same subprogram by more than one calling program, proper argument definition is automatically ensured by the F7Z compiler.

Two methods of argument passing are used by F7Z for subprograms that are expanded in-line:

- Pass-by-value
- Pass-by-address

Whether the pass-by-value or pass-by-address method is used is determined by the dummy arguments of the called subprogram. If the dummy argument is an array or character string or is surrounded by slashes (/dummy/), the pass-by-address method is used. For other dummy arguments, the pass-by-value method is used. See *FORTRAN VII Language and Syntax — A Reference* for more information on argument passing methods.

The pass-by-value method uses an arithmetic assignment statement to assign the value of the actual argument to the corresponding dummy argument.

Example:

```
F.Y = X
F.Y = F.Y + 2.0
F.F = F.Y
  X = F.Y
```

This example of extended code, which is taken from the last section, "Intermediate Code Translation," uses the following assignment statement to pass the value of X to the dummy argument Y in the function F:

```
F.Y = X
```

The value of X is passed back to the calling program unit through the assignment statement:

```
X = F.Y
```

Dummy arrays and character strings are always passed by address. If a character variable, an array, or array element with a constant subscript is passed to a dummy array, the compiler generates an EQUIVALENCE statement for the two variables. If a character variable with a variable substring or an array element with a variable subscript is passed to a dummy array, the compiler generates a BIND statement. This statement binds the dummy array to the starting location of the actual argument. If the equivalenced variable is an array that is used in an I/O statement or in an actual argument list of a CALL, the compiler uses a VECTOR function to indicate the length of the array to be transferred.

The user can pass an argument of one type to a dummy argument of a different type. F7Z reconciles this type mismatch through the NOTYPE function, which can occur on either side of the compiler generated assignment statement.

The user must be certain that an expanded program does not change the value of a constant argument that is passed to one of its dummy arguments. If a constant is passed by value to the subprogram, the compiler will not generate the statement that changes the value. If a constant is passed by address to the subprogram, the compiler will generate a warning indicating that if the code is executed, the value of the constant is modified.

Preparing Source Code for In-line Expansion

Ordinarily, source programs consisting entirely of FORTRAN code require no preparation for in-line expansion other than the insertion of the appropriate $INLINE, $INLIB, and $INSKIP directives. These directives are discussed in detail in Chapter 3. However, if two or more program units are expanding the same subprogram in-line and the same value of one of the subprogram variables is used by all calls to the subprogram, that variable must be specified by a SAVE statement. Subprogram variables must also be specified in a SAVE statement if the subprogram is both expanded in one program unit and separately compiled as a single unit that can be called by other programs. See *FORTRAN VII Language and Syntax — A Reference* for more information on the SAVE statement.

Programs which expand subprograms containing embedded assembly code require special directives to the compiler. The compiler must be able to recognize which symbols in a CAL block are FORTRAN variables. Recognition is possible only if each FORTRAN variable referenced in a CAL block either (1) appears in a FORTRAN statement prior to the block or (2) appears in a $SETS or $USES directive in the CAL block or in a prior CAL block. The compiler must also be informed which CAL symbols are not to be used by multiple expansions of an embedded CAL block. $DISTINCT tells the compiler which symbols in the block must be converted to compiler generated symbols to prevent multiple definition.

In addition to argument passing, two other items must be taken into consideration when preparing embedded assembly blocks for in-line expansion. First, due to the substitution of FORTRAN symbols occurring in a CAL statement in an in-line expanded subprogram, the length of that CAL line may exceed 71 characters. CAL lines exceeding 71 characters cause the compiler to generate CAL continuation lines by placing a nonblank character in column 72. While such a CAL statement can be interpreted by the CAL macroprocessor, it will not be recognized as a legal statement by CAL. Therefore, lines of CAL code must be as short as possible. Second, data areas which are declared only in an embedded assembly block are not shared by repeated expansions of the block unless the user restructures the block for this purpose. Therefore, the user should avoid declaring common data areas in an assembly block embedded in a subprogram that is expanded more than once.

When To Use In-line Expansion

The F7Z in-line expansion feature is designed for use in programs that are in the final stage of development. This allows the user to concentrate on tuning the performance of the program when it can be examined in its entirety rather than as it is being written. Therefore, it is not necessary to limit the use of structured design techniques that aid programmers, but significantly reduce execution speed. Through universal optimization, the programmer can improve performance without altering any line of the original source program.

After the program is developed to the point where it can be examined in its entirety, it can be scanned for subprogram calls that can be profitably expanded. The initial temptation may be to simply expand all calls within a program. It must be remembered, however, that in-line expansion is essentially a tradeoff between memory space and execution time. Only expansion of those routines which are called often at execution will yield a significant increase in performance to offset the increased use of memory space. Expansion elsewhere will, at best, yield negligible results or, at worst, produce programs that result in excessively long compilation times, poor performance due to increased size, or compile time failure. The internal table storage area of the F7Z compiler is limited and the run-times of the optimization algorithms increase rapidly as the size of the program approaches 3000 lines.

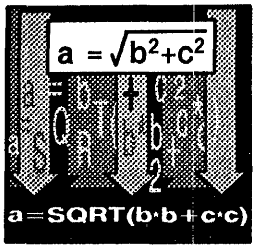
NOTE Failure of compilation due to insufficient internal table space is an indication that in-line expansion was carried out beyond its limits. If this occurs, subprograms with lower run-time profile should be excluded from in-line expansion.

Studies show that a typical program spends most of its execution time in a small percentage of the total program code. In-line expansion of subprograms called in such areas can lead to substantial improvements in performance without a significant increase in program size. These areas include:

- Nested loop structures that contain calls to moderately sized subprograms (approximately 100 lines). These loops should be nested down to the third level or below. Programs often spend much of their execution time in calling and executing subprograms within such loops.
- Calls to subprograms in which constant arguments are used to determine the flow of control within the subprogram. These programs, which often exist solely for convenience in design, are particularly suited for in-line expansion.
- Repeated calls to subprograms containing long computations that would appear redundant if made a part of the original calling program. An example is a subprogram that references arrays with dummy variables as indexes, but are called with constants for the corresponding dummy arguments.

The execution profile analysis (XPA) system and the call recording analysis (CRA) system provide a means for locating subprogram candidates for in-line expansion. XPA and CRA, which are included in the F7Z package, are explained in Chapter 12 and Chapter 13, respectively.

Once the programmer has decided which calls to subprograms are to be expanded, the source program can be recompiled with the necessary in-line directives. Details on the use of these directives are given in Chapter 3.



16

FORTRAN VII Error Messages

In this chapter

We provide you with a description of each of the FORTRAN VII language system error messages. End-of-task (EOT) codes are also provided.

Topics include:

- Compiler error messages
- Run-time library (RTL) error messages
- Special utility error messages (CRA/XPA)
- EOT codes

Introduction

The following sections document the error messages which are issued as a result of an error during compilation, execution of a FORTRAN VII RTL routine, or execution of a special utility. For more information regarding the context of each error message refer to the appropriate section of this manual, *FORTRAN VII Language and Syntax - A Reference*, and any additional manuals listed under the message description. For further information on the RTL error messages listed under the section "Math Errors," refer to the *System Mathematical Run-Time Library - A Reference.*

Compiler Messages

Following is a list of error messages generated by the F7O and F7Z compilers. The format is as follows:

ERROR (*n*) - *description*

Where *n* is the error message number and *description* is a diagnostic message.

Code	Message Text
A00	SIZE OF LOCAL DATA AREA EXCEEDS COMPILER LIMIT OF 16,252,927 BYTES. COMPILATION ABORTED !!
A01	LEFTMOST CHARACTER POSITION OF SUBSTRING EXCEEDS LENGTH OF IDENTIFIER
A02	RIGHTMOST CHARACTER POSITION OF SUBSTRING EXCEEDS LENGTH OF IDENTIFIER
A03	LEFTMOST CHARACTER POSITION OF SUBSTRING IS LESS THAN 1
A04	RIGHTMOST CHARACTER POSITION OF SUBSTRING IS LESS THAN 1
A05	LEFTMOST CHARACTER POSITION OF SUBSTRING EXCEEDS THE RIGHT MOST POSITION
A06	NUMBER OF SUBSCRIPTS DO NOT MATCH DECLARED NUMBER OF DIMENSIONS
A07	ADJUSTABLE BOUND NOT ALLOWED IN MAIN PROGRAM OR BLOCKDATA
A08	UPPER BOUND IS LESS THAN LOWER BOUND
A09	IDENTIFIER HAS PREVIOUSLY BEEN DIMENSIONED
A10	ARRAY MUST NOT HAVE MORE THAN SEVEN DIMENSIONS
A11	WRONG TYPE OF ARGUMENT FOR INTRINSIC FUNCTION/SUBROUTINE
A12	GENERIC FUNCTION NAME CANNOT BE PASSED AS AN ARGUMENT
A13	TOO FEW ARGUMENTS FOR INTRINSIC FUNCTION/SUBROUTINE
A14	TOO MANY ARGUMENTS FOR INTRINSIC FUNCTION/SUBROUTINE
A15	THIS KIND OF ARGUMENT NOT ALLOWED FOR INTRINSIC FUNCTIONS/SUBROUTINE
A16	SUBPROGRAM NOT DECLARED IN INTRINSIC STATEMENT
A17	SUBPROGRAM NOT DECLARED IN EXTERNAL STATEMENT
A18	NUMBER OF ARGUMENTS DIFFERS FROM STATEMENT FUNCTION DEFINITION
A19	NUMBER OF ARGUMENTS DIFFERS FROM FIRST REFERENCE
A20	TYPE OF THIS ARGUMENT DIFFERS FROM FIRST REFERENCE
A21	TYPE OF THIS ARGUMENT DIFFERS FROM STATEMENT FUNCTION DEFINITION
A22	ARGUMENT WAS NOT AN ARRAY NAME IN FIRST REFERENCE
A23	ARGUMENT WAS NOT A PROCEDURE NAME IN FIRST REFERENCE
A24	ARGUMENT WAS AN ARRAY NAME IN FIRST REFERENCE
A25	ARGUMENT WAS A PROCEDURE NAME IN FIRST REFERENCE
A26	THIS ARGUMENT FOR INTRINSIC FUNCTION/SUBROUTINE SHOULD NOT BE AN EXPRESSION

Code	Message Text
D01	REPLICATION FACTOR MUST BE A POSITIVE INTEGER OR A PARAMETER
D02	NOT PREVIOUSLY DEFINED AS A PARAMETER, OR BAD Z CONSTANT
D03	MUST BE A VARIABLE OR AN ARRAY
D04	SUBSCRIPT EXPRESSION MUST BE CONSTANT
D05	SUBSTRING EXPRESSION MUST BE CONSTANT
D06	UNDIMENSIONED ARRAY OR PROCEDURE REFERENCE NOT ALLOWED
D07	SHOULD BE INTEGER VARIABLE
E01	TYPE OF EXPRESSION INCOMPATIBLE FOR ADDITION OR SUBTRACTION
E02	TYPE OF EXPRESSION INCOMPATIBLE FOR MULTIPLICATION OR DIVISION
E03	TYPE OF EXPRESSION INCOMPATIBLE FOR RAISING TO A POWER
E04	OPERANDS OF RELATIONAL OPERATOR CANNOT BE COMPARED
E05	COMPLEX OPERANDS CAN ONLY BE COMPARED FOR (IN)EQUALITY
E06	TYPE OF EXPRESSION MUST BE LOGICAL
E07	TYPE OF EXPRESSION INCOMPATIBLE FOR CONCATENATION (//)
E08	PASSED LENGTH DUMMY ARGUMENT MUST NOT BE CONCATENATED
E09	TYPE OF EXPRESSION MUST BE BIT
E10	LOGICAL EXPRESSION DOES NOT FOLLOW .NOT.
E11	THE CHARACTER POSITIONS OF THIS OPERAND MAY BE REFERENCED IN THE TARGET
E12	LOGICAL OPERAND IN ARITHMETIC EXPRESSION
F01	FORMAT STATEMENT MUST HAVE A LABEL
F02	FORMAT DOES NOT BEGIN WITH A LEFT PARENTHESIS
F03	UNPRINTABLE CHARACTER ENCOUNTERED IN FORMAT
F04	NESTING LEVEL OF LEFT PARENTHESES IN FORMAT EXCEEDS 255
F05	REPEAT COUNT NOT ALLOWED FOR EDIT DESCRIPTOR
F06	MISSING OR ZERO LENGTH IN HOLLERITH EDIT DESCRIPTOR
F08	ENDING RIGHT PARENTHESIS NOT FOUND IN FORMAT
F09	NUMBER EXCEEDS HALFWORD IN FORMAT
F10	ILLEGAL CHARACTER ENCOUNTERED IN FORMAT
F11	ILLEGAL MINUS SIGN ENCOUNTERED IN FORMAT
F12	D FIELD EXCEEDS FIELD WIDTH
F13	NUMBER OF EXPONENT DIGITS EXCEEDS FIELD WIDTH
F14	ZERO REPEAT COUNT IN FORMAT

Code	Message Text
F16	FIELD WIDTH MISSING OR ZERO IN FORMAT
F17	D FIELD MISSING IN FORMAT
F18	E FIELD MISSING, ZERO, OR GREATER THAN 255 IN FORMAT
G02	ILLEGAL OR MISSING OPERAND OF EXPRESSION
G03	ILLEGAL OR MISSING OPERAND OF EXPRESSION
G04	ILLEGAL OR MISSING EXPRESSION
G05	ILLEGAL OR MISSING NAME
G06	ILLEGAL OR MISSING NAME
G07	ILLEGAL OR MISSING NAME
G08	ILLEGAL OR MISSING NAME
G09	ILLEGAL OR MISSING SPECIFIER
G10	ILLEGAL OR MISSING SPECIFIER
G11	ILLEGAL OR MISSING SPECIFIER LIST
G12	ILLEGAL OR MISSING PART OF SPECIFIER LIST
G13	ILLEGAL OR MISSING SPECIFIER LIST
G14	ILLEGAL OR MISSING EXPRESSION/SPECIFIER LIST
G15	ILLEGAL OR MISSING BLOCK IF STATEMENT
G16	ILLEGAL OR MISSING ARGUMENT LIST
G17	ILLEGAL OR MISSING ARGUMENT
G18	ILLEGAL OR MISSING SPECIFIER
G19	ILLEGAL OR MISSING SPECIFIER
G20	ILLEGAL OR MISSING SPECIFIER LIST
G21	ILLEGAL OR MISSING PART OF SPECIFIER LIST
G22	ILLEGAL OR MISSING SPECIFIER LIST
G23	ILLEGAL OR MISSING SUBSTRING OPERATOR
G24	ILLEGAL OR MISSING COMMON BLOCK NAME
G25	ILLEGAL OR MISSING COMMON BLOCK DECLARATION
G26	ILLEGAL OR MISSING COMMON BLOCK DECLARATION LIST
G27	ILLEGAL OR MISSING NAME
G28	ILLEGAL OR MISSING LIST OF NAMES
G29	ILLEGAL OR MISSING LIST OF NAMES
G30	ILLEGAL OR MISSING COMMON BLOCK DECLARATION
G31	ILLEGAL OR MISSING COMPLEX CONSTANT
G32	ILLEGAL OR MISSING COMPLEX CONSTANT PART
G33	ILLEGAL OR MISSING CONSTANT/NAME
G34	ILLEGAL OR MISSING EXPRESSION
G35	ILLEGAL OR MISSING I/O CONTROL LIST ITEM
G36	ILLEGAL OR MISSING I/O CONTROL LIST ITEM

Code	Message Text
G37	ILLEGAL OR MISSING I/O CONTROL LIST ITEM
G38	ILLEGAL OR MISSING I/O CONTROL LIST
G39	ILLEGAL OR MISSING PART OF I/O CONTROL LIST
G40	ILLEGAL OR MISSING PART OF I/O CONTROL LIST
G41	ILLEGAL OR MISSING PART OF I/O CONTROL LIST
G42	ILLEGAL OR MISSING DATA CONSTANT LIST
G43	ILLEGAL OR MISSING DATA DEFINITION
G44	ILLEGAL OR MISSING DATA DEFINITION LIST
G45	ILLEGAL OR MISSING DATA IMPLIED DO LIST
G46	ILLEGAL OR MISSING NAME IN A DATA IMPLIED DO LIST
G47	ILLEGAL OR MISSING LIST OF NAMES
G49	ILLEGAL OR MISSING REPETITION FACTOR
G50	ILLEGAL OR MISSING CONSTANT
G51	ILLEGAL OR MISSING CONSTANT
G52	ILLEGAL OR MISSING NAME/IMPLIED DO LIST
G53	ILLEGAL OR MISSING LIST OF NAMES
G54	ILLEGAL OR MISSING DIMENSION SPECIFICATION
G55	ILLEGAL OR MISSING LIST OF DIMENSION SPECIFICATIONS
G57	ILLEGAL OR MISSING DIMENSION BOUND
G58	ILLEGAL OR MISSING DIMENSION BOUND LIST
G59	ILLEGAL OR MISSING DIMENSION BOUND
G60	ILLEGAL OR MISSING DIMENSION BOUND LIST
G61	ILLEGAL OR MISSING DIMENSION BOUND SPECIFICATION
G62	ILLEGAL OR MISSING DO LOOP CONDITION
G65	ILLEGAL OR MISSING NAME
G66	ILLEGAL OR MISSING DO RANGE
G67	ILLEGAL OR MISSING CONTROL LIST
G68	ILLEGAL OR MISSING NAME
G69	ILLEGAL OR MISSING LIST OF NAMES
G70	ILLEGAL OR MISSING EQUIVALENCE LIST
G71	ILLEGAL OR MISSING EQUIVALENCE LIST
G73	ILLEGAL OR MISSING SPECIFIER
G74	ILLEGAL OR MISSING LABEL
G75	ILLEGAL OR MISSING OPERAND OF EXPRESSION
G76	ILLEGAL OR MISSING EXPRESSION
G77	ILLEGAL OR MISSING LIST OF NAMES
G78	ILLEGAL OR MISSING FORMAT IDENTIFIER
G79	ILLEGAL OR MISSING FORMAT IDENTIFIER
G80	ILLEGAL OR MISSING LABEL

Code	Message Text
G83	ILLEGAL OR MISSING LIST OF LABELS
G84	ILLEGAL OR MISSING LIST OF LABELS
G85	ILLEGAL OR MISSING IF CONDITION
G86	ILLEGAL OR MISSING IMPLICIT DECLARATOR
G87	ILLEGAL OR MISSING IMPLICIT DECLARATOR LIST
G88	ILLEGAL OR MISSING IMPLICIT RANGE
G89	ILLEGAL OR MISSING IMPLICIT RANGE LIST
G90	ILLEGAL OR MISSING TYPE
G91	ILLEGAL OR MISSING SPECIFIER
G92	ILLEGAL OR MISSING SPECIFIER
G93	ILLEGAL OR MISSING KEYWORD
G94	ILLEGAL OR MISSING KEYWORD
G95	ILLEGAL OR MISSING KEYWORD
G96	ILLEGAL OR MISSING SPECIFIER LIST
G97	ILLEGAL OR MISSING PART OF SPECIFIER LIST
G98	ILLEGAL OR MISSING SPECIFIER LIST
G99	ILLEGAL OR MISSING EXPRESSION
G100	ILLEGAL OR MISSING EXPRESSION
G101	ILLEGAL OR MISSING EXPRESSION
G102	ILLEGAL OR MISSING CHARACTER EXPRESSION
G103	ILLEGAL OR MISSING CHARACTER EXPRESSION
G104	ILLEGAL OR MISSING IMPLIED DO LIST
G105	ILLEGAL OR MISSING I/O LIST ITEM
G107	ILLEGAL OR MISSING I/O LIST
G108	ILLEGAL OR MISSING FORMAT IDENTIFIER
G109	ILLEGAL OR MISSING LABEL
G110	ILLEGAL OR MISSING LENGTH SPECIFICATION
G111	ILLEGAL OR MISSING LENGTH SPECIFICATION
G112	ILLEGAL OR MISSING LOGICAL EXPRESSION
G113	ILLEGAL OR MISSING EXPRESSION
G114	ILLEGAL OR MISSING OPERAND OF EXPRESSION
G115	ILLEGAL OR MISSING LOOP CONDITION
G116	ILLEGAL OR MISSING NAMELIST DECLARATION
G117	ILLEGAL OR MISSING LIST OF NAMELIST DECLARATIONS
G118	ILLEGAL OR MISSING NAMELIST NAME
G119	ILLEGAL OR MISSING LIST OF NAMELIST ITEMS
G120	ILLEGAL OR MISSING LIST OF NAMELIST NAMES
G121	ILLEGAL OR MISSING NAMELIST DECLARATION
G122	ILLEGAL OR MISSING NAMELIST NAME
G123	ILLEGAL OR MISSING NAMELIST NAME

Code	Message Text
G126	ILLEGAL OR MISSING SPECIFIER
G127	ILLEGAL OR MISSING SPECIFIER
G128	ILLEGAL OR MISSING KEYWORD
G129	ILLEGAL OR MISSING KEYWORD
G130	ILLEGAL OR MISSING SPECIFIER LIST
G131	ILLEGAL OR MISSING PART OF SPECIFIER LIST
G132	ILLEGAL OR MISSING SPECIFIER LIST
G133	ILLEGAL OR MISSING EXPRESSION IN SUBSTRING REFERENCE
G134	ILLEGAL OR MISSING ARGUMENT
G135	ILLEGAL OR MISSING EXPRESSION
G136	ILLEGAL OR MISSING DO INCREMENT
G137	ILLEGAL OR MISSING DUMMY ARGUMENT LIST
G138	ILLEGAL OR MISSING LIST OF SUBSCRIPTS/ARGUMENTS
G139	ILLEGAL OR MISSING PARAMETER DEFINITION
G140	ILLEGAL OR MISSING PARAMETER DEFINITION LIST
G141	ILLEGAL OR MISSING READ/WRITE
G142	ILLEGAL OR MISSING NAME/COMMON NAME
G143	ILLEGAL OR MISSING LIST OF SAVE ITEMS
G144	ILLEGAL OR MISSING SIGN
G146	ILLEGAL OR MISSING STATEMENT
G147	ILLEGAL OR MISSING STATEMENT AFTER LOGICAL IF
G148	ILLEGAL OR MISSING STATEMENT
G149	ILLEGAL OR MISSING CONSTANT
G150	ILLEGAL OR MISSING DUMMY ARGUMENT
G151	ILLEGAL OR MISSING DUMMY ARGUMENT LIST
G152	ILLEGAL OR MISSING DUMMY ARGUMENT LIST
G153	ILLEGAL OR MISSING ARRAY SUBSCRIPT/FUNCTION ARGUMENT
G154	ILLEGAL OR MISSING ARRAY/FUNCTION REFERENCE
G155	ILLEGAL OR MISSING LIST OF SUBSCRIPTS/ARGUMENTS
G156	ILLEGAL OR MISSING SUBSTRING REFERENCE
G157	ILLEGAL OR MISSING NAME
G160	ILLEGAL OR MISSING NAME
G161	ILLEGAL OR MISSING LIST OF NAMES
G162	ILLEGAL OR MISSING TYPE
G163	ILLEGAL OR MISSING TYPE SPECIFICATION
G164	ILLEGAL OR MISSING NAME
G165	ILLEGAL OR MISSING UNIT= SPECIFIER
G166	ILLEGAL OR MISSING UNIT SPECIFICATION

Code	Message Text
G167	ILLEGAL OR MISSING CONSTANT
G201	MISSING OR MUST BE **
G202	MISSING OR MUST BE //
G203	MISSING OR MUST BE RELATIONAL
G204	MISSING OR MUST BE &
G205	MISSING OR MUST BE LEFT PARENTHESIS
G206	MISSING OR MUST BE RIGHT PARENTHESIS
G207	MISSING OR MUST BE *
G208	MISSING OR MUST BE +
G209	MISSING OR MUST BE COMMA
G210	MISSING OR MUST BE -
G211	MISSING OR MUST BE LOGICAL RELATIONAL
G212	MISSING OR MUST BE /
G213	MISSING OR MUST BE NUMBER
G214	MISSING OR MUST BE COLON
G215	MISSING OR MUST BE .BAND.
G216	MISSING OR MUST BE .AND.
G217	MISSING OR MUST BE EQUAL SIGN
G218	MISSING OR MUST BE .OR.
G219	MISSING OR MUST BE .BNOT.
G220	MISSING OR MUST BE NAME
G221	MISSING OR MUST BE .NOT.
G222	MISSING OR MUST BE .BOR.
G223	MISSING OR MUST BE END OF STATEMENT
G224	MISSING OR MUST BE >>
G225	MISSING OR MUST BE CONSTANT
G226	MISSING OR MUST BE $
G227	MISSING OR MUST BE #
G228	MISSING OR MUST BE INTEGER
G229	MISSING OR MUST BE REAL
G230	MISSING OR MUST BE COMPLEX
G231	MISSING OR MUST BE LOGICAL
G232	MISSING OR MUST BE CHARACTER
G233	MISSING OR MUST BE BIT
G234	MISSING OR MUST BE DOUBLE PRECISION
G235	MISSING OR MUST BE DOUBLE COMPLEX
G236	MISSING OR MUST BE OPEN
G237	MISSING OR MUST BE INQUIRE
G238	MISSING OR MUST BE CLOSE
G239	MISSING OR MUST BE BACKSPACE

Code	Message Text
G240	MISSING OR MUST BE REWIND
G241	MISSING OR MUST BE ENDFILE
G242	MISSING OR MUST BE READ
G243	MISSING OR MUST BE WRITE
G244	MISSING OR MUST BE PRINT
G245	MISSING OR MUST BE ACCEPT
G246	MISSING OR MUST BE TYPE
G247	MISSING OR MUST BE ASSIGN
G248	MISSING OR MUST BE BLOCK DATA
G249	MISSING OR MUST BE CALL
G250	MISSING OR MUST BE COMMON
G251	MISSING OR MUST BE CONTINUE
G252	MISSING OR MUST BE DATA
G253	MISSING OR MUST BE DECODE
G254	MISSING OR MUST BE DIMENSION
G255	MISSING OR MUST BE DO
G256	MISSING OR MUST BE ELSE
G257	MISSING OR MUST BE ENCODE
G258	MISSING OR MUST BE ENDIF
G259	MISSING OR MUST BE ENTRY
G260	MISSING OR MUST BE EQUIVALENCE
G261	MISSING OR MUST BE EXTERNAL
G262	MISSING OR MUST BE FORMAT
G263	MISSING OR MUST BE FUNCTION
G264	MISSING OR MUST BE GO TO
G265	MISSING OR MUST BE IF
G266	MISSING OR MUST BE IMPLICIT
G267	MISSING OR MUST BE INTRINSIC
G268	MISSING OR MUST BE NAMELIST
G269	MISSING OR MUST BE PARAMETER
G270	MISSING OR MUST BE PAUSE
G271	MISSING OR MUST BE PROGRAM
G272	MISSING OR MUST BE RETURN
G273	MISSING OR MUST BE SAVE
G274	MISSING OR MUST BE STOP
G275	MISSING OR MUST BE SUBROUTINE
G276	MISSING OR MUST BE TO
G277	MISSING OR MUST BE ACCESS=
G278	MISSING OR MUST BE ATTRIBUTES=
G279	MISSING OR MUST BE BLANK=

Code	Message Text
G280	MISSING OR MUST BE BLOCKSIZE=
G281	MISSING OR MUST BE COUNT=
G282	MISSING OR MUST BE COUNTBY=
G283	MISSING OR MUST BE DEVCODE=
G284	MISSING OR MUST BE DIRECT=
G285	MISSING OR MUST BE END=
G286	MISSING OR MUST BE ERR=
G287	MISSING OR MUST BE EXIST=
G288	MISSING OR MUST BE FILE=
G289	MISSING OR MUST BE FMT=
G290	MISSING OR MUST BE FORM=
G291	MISSING OR MUST BE FORMATTED=
G292	MISSING OR MUST BE IOSTAT=
G293	MISSING OR MUST BE ISIZE=
G294	MISSING OR MUST BE NAME=
G295	MISSING OR MUST BE NAMED=
G296	MISSING OR MUST BE NEXTREC=
G297	MISSING OR MUST BE NML=
G298	MISSING OR MUST BE NUMBER=
G299	MISSING OR MUST BE OPENED=
G300	MISSING OR MUST BE REC=
G301	MISSING OR MUST BE RECL=
G302	MISSING OR MUST BE RENAME=
G303	MISSING OR MUST BE REPROTECT=
G304	MISSING OR MUST BE RKEY=
G305	MISSING OR MUST BE SEQUENTIAL=
G306	MISSING OR MUST BE SHARE=
G307	MISSING OR MUST BE SIZE=
G308	MISSING OR MUST BE STATUS=
G309	MISSING OR MUST BE TYPE=
G310	MISSING OR MUST BE UNFORMATTED=
G311	MISSING OR MUST BE UNIT=
G312	MISSING OR MUST BE WKEY=
G313	MISSING OR MUST BE THEN
G314	MISSING OR MUST BE NONE
G315	MISSING OR MUST BE GLOBAL
G316	MISSING OR MUST BE LOCAL
G317	MISSING OR MUST BE WHILE
G318	MISSING OR MUST BE ENDDO
G319	MISSING OR MUST BE UNTIL

Code	Message Text
G320	MISSING OR MUST BE LOOP
G321	MISSING OR MUST BE REPEAT
G322	MISSING OR MUST BE EXIT
G323	MISSING OR MUST BE ESCAPE
G324	MISSING OR MUST BE STEP
G325	MISSING OR MUST BE STEPBY
G326	MISSING OR MUST BE STEPFIRST
G327	MISSING OR MUST BE NEXT
G328	MISSING OR MUST BE UBLE COMPLEX
G329	MISSING OR MUST BE UBLE PRECISION
G330	MISSING OR MUST BE RECURSIVE
G331	MISSING OR MUST BE BYTE
G335	MISSING OR MUST BE RWXKEY=
I01	ILLEGAL INTERNAL FILE SPECIFICATION
I02	DUPLICATE SPECIFIER OR ILLEGAL COMBINATION OF SPECIFIERS
I03	THIS KEYWORD SPECIFIER IS NOT ALLOWED ON THIS STATEMENT
I04	INQUIRE MUST HAVE A 'UNIT=' OR A 'FILE=' SPECIFIER
I05	INQUIRE MUST NOT SPECIFY BOTH 'UNIT=' AND 'FILE='
I06	MISSING UNIT SPECIFICATION
I07	DIRECT ACCESS NOT ALLOWED ON LIST DIRECTED OR NAMELIST I/O
I08	THIS KEYWORD SPECIFIER NOT ALLOWED WITH INTERNAL FILE I/O
I09	ILLEGAL I/O LIST ITEM ON INPUT LIST
I10	I/O LIST NOT ALLOWED ON NAMELIST I/O
I11	NOT A NAMELIST NAME
I12	NAMELIST NAME NOT ALLOWED HERE
I13	HOLLERITH NOT ALLOWED IN I/O LIST
I14	INTERNAL FILE I/O MUST BE FORMATTED
I15	ABOVE STATEMENT GENERATES SVC CALL
I16	THIS KEYWORD IS NO LONGER SUPPORTED AND IS IGNORED
L01	SUBPROGRAM NAME <name> DOES NOT MATCH HEADER <name>.

Code	Message Text
L02	SOURCE OF SUBPROGRAM <name> IS NOT FOUND IN THE LIBRARY. or SOURCE OF SUBPROGRAM <name> IS NOT FOUND IN THE LIBRARY, LIBRARY IS NOT SPECIFIED.
L03	SUBPROGRAM <name> CALLING <name> IS RECURSIVE.
L04	<name> NOT INVOKED AS A FUNCTION IN SUBPROGRAM <name>.
L05	<name> NOT INVOKED AS A SUBROUTINE IN SUBPROGRAM <name>.
L06	TYPE OF FUNCTION <name> INCOMPATIBLE WITH ITS INVOCATION IN SUBPROGRAM <name>.
L07	TYPE OF FUNCTION <name> INCOMPATIBLE WITH ITS INVOCATION IN SUBPROGRAM <name>.
L08	FUNCTION <name> NOT DECLARED IN <name> WITH TYPE SAME AS IN OTHER DECLARATIONS.
L09	ARGUMENTS SUPPLIED IN THE INVOCATION OF <name> IN <name> ARE NOT CONSISTENT WITH OTHER INVOCATIONS.
L10	INVOCATION OF FUNCTION <name> IN <name> DOES NOT AGREE WITH OTHER USES OF IT AS A SUBROUTINE.
L11	INVOCATION OF SUBROUTINE <name> IN <name> DOES NOT AGREE WITH OTHER USES OF IT AS A FUNCTION.
L12	CHARACTER FUNCTION LENGTH OF <name> LONGER THAN IT IS SPECIFIED IN <name>
L13	DUMMY VARIABLE <name> DOES NOT RECEIVE CORRECT TYPE IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L14	DUMMY VARIABLE <name> DOES NOT RECEIVE PROCEDURE IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L15	DUMMY VARIABLE <name> DOES NOT RECEIVE ARRAY IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L16	DUMMY VARIABLE <name> DOES NOT RECEIVE MATCHING CHARACTER LENGTH IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L17	DUMMY VARIABLE <name> DOES NOT RECEIVE CHARACTER IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L18	DUMMY CHARACTER ARGUMENT <name> DOES NOT RECEIVE INTEGER VALUE FOR ITS LENGTH IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L19	DUMMY VARIABLE <name> DOES NOT RECEIVE FORTRAN LABEL IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.

Code	Message Text
L20	<name> NOT INVOKED WITH SUFFICIENT ARGUMENTS IN LINE NNNN OF <name>.
L21	THE FILE <name> IS NOT ON A DIRECT ACCESS DEVICE.
L22	THE FILE <name> DOES NOT EXIST.
L23	<name> HAS BEEN USED AS A SUBPROGRAM NAME ELSEWHERE.
L24	SOURCE OF SUBPROGRAM <name> IS NOT THE FIRST MODULE IN THE FILE <name>.
L25	EXTERNAL <name> IN <name> INCONSISTENT WITH ITS USE AS A GLOBAL VARIABLE ELSEWHERE.
L26	DUMMY VARIABLE <name> DOES NOT RECEIVE SCALAR IN THE INVOCATION OF <name> IN LINE NNNN OF <name>.
L27	BLOCK DATA SUBPROGRAM MUST NOT BE EXPANDED INLINE.
L28	NO ENTRY <name> IN SUBPROGRAM <name> INVOKED IN LINE 'LLL' OF <name>
L29	PREVIOUS LINE WILL CORRUPT A CONSTANT IF EXECUTED.
L30	TYPE OF GLOBAL <name> IN <name> INCONSISTENT WITH OTHER DECLARATIONS OF IT ELSEWHERE.
L31	CHARACTER FUNCTION <name> NOT INVOKED WITH MATCHING LENGTH IN LINE NNNN OF <name>.
L32	GLOBAL <name> IN <name> INCONSISTENT WITH ITS USE AS AN EXTERNAL ELSEWHERE.
L33	SOURCE OF SUBPROGRAM <name> IS NOT FOUND IN THE FILE <name>.
L34	SOURCE OF SUBPROGRAM <name> IS NOT FOUND IN THE FILE <name>.
L35	<name> IS NOT A SUBROUTINE.
L36	<name> IS NOT A FUNCTION.
L37	TYPE OR NUMBER OF ARGUMENTS FOR <name> INCONSISTENT WITH OTHER INVOCATIONS.
L38	<name> IS INVOKED WITH MORE THAN NECESSARY ARGUMENTS IN LINE <line number> OF <name>.
L39	REFERENCE OF <name> IN LINE <line number> IS NOT EXPANDED IN LINE.
M01	UNDEFINED LABELS:
M02	UNREFERENCED LABELS:
M03	CONSTANT SUBSCRIPT OUT OF RANGE ON AN ARRAY REFERENCE OF <name>.

Code	Message Text
M04	TYPE OR LENGTH OF STMT FUNC ARGUMENT <name> IS NOT COMPATIBLE WITH DUMMY ARGUMENT.
M05	ASSUMED-SIZE OR ADJUSTABLE ARRAY <name> MUST BE DUMMY ARGUMENT.
M06	DIMENSION DECLARATOR CONTAINS LOCAL VARIABLE FOR DUMMY ARRAY <name>.
M07	ILLEGAL DIMENSION DECLARATOR FOR ARRAY <name>.
M08	<name> EXTENDS THE HEAD OF A COMMON/GLOBAL VIA EQUIVALENCE.
M09	AN EQUIVALENCING CONFLICT OCCURS ON <name>.
M10	EQUIVALENCE RESULTS IN ILLEGAL ALIGNMENT FOR <name>.
M11	ALIGNMENT IMPOSED ON COMMON ELEMENT <name>.
M12	DUMMY ARGUMENT, <name>, MAY NOT BE EQUIVALENCED.
M13	MORE THAN ONE COMMON/GLOBAL ITEM IN AN EQUIVALENCE GROUP CAUSES A CONFLICT ON <name>.
M14	THE PROMOTION OF <name> INTO COMMON/GLOBAL CAUSES AN EQUIVALENCE CONFLICT.
M15	POSSIBLE ILLEGAL BRANCH INTO DO-LOOP.
M16	ILLEGAL BRANCH INTO DO-LOOP.
M17	ILLEGAL OPERATION IN SUBSCRIPT/SUBSTRING EXPRESSION.
M18	TOO FEW CONSTANTS TO INITIALIZE IDENTIFIER <name>.
M19	EXTRA DATA CONSTANTS IGNORED.
M20	A DATA CONSTANT OVERFLOWS VARIABLE <name>.
M21	INVALID INITIALIZATION DATA FOR CHARACTER VARIABLE <name>.
M22	VARIABLE <name> WAS USED AS AN ARRAY BUT NEVER DIMENSIONED.
M23	INVALID INITIALIZATION DATA FOR LOGICAL VARIABLE <name>.
M24	INVALID INITIALIZATION USE OF LOGICAL DATA FOR <name>.
M25	SUBSCRIPT/SUBSTRING OUT OF BOUNDS FOR <name>.
M26	THE VARIABLE, <name>, TO BE INITIALIZED IS NOT A LOCAL VARIABLE.
M27	THE LOCAL VARIABLE, <name>, CAN NOT BE INITIALIZED IN A BLOCK DATA.
M28	EQUIVALENCING OF COMMON/GLOBAL ITEM <name> CAUSES A CONFLICT WITH SAVE DESIGNATION.
M29	A DATA IMPLIED-DO PARAMETER IS ILLEGAL.
M30	FUNCTION ENTRY, <name>, CANNOT APPEAR IN COMMON.

Code	Message Text
M31	THE NUMBER OF SUBSCRIPTS CONFLICTS WITH DECLARATION FOR <name>.
M32	THE INITIAL VALUE FOR A SUBSTRING OF <name> IS OUT OF RANGE.
M33	THE FINAL VALUE FOR A SUBSTRING OF <name> IS OUT OF RANGE.
M34	SUBSTRING USED WITH NON-CHARACTER VARIABLE <name>.
M35	CONSTANT ARGUMENT PADDED WITH BLANKS FOR STMT FUNCTION <name>.
M36	PASSED-LENGTH SPECIFICATION IS NOT ALLOWED ON THE CHARACTER IDENTIFIER <name>.
M37	NAMELIST CONTAINS THE INELIGIBLE IDENTIFIER <name>.
M38	TYPE OF ENTRY-POINT <name> IS NOT TYPE CHARACTER.
M39	THE CHARACTER LENGTH OF ENTRY-POINT <name> <> LENGTH OF. THE FUNCTION.
M40	THE TYPE OF ENTRY-POINT <name> IS INCOMPATIBLE WITH THE PRIMARY FUNCTION'S TYPE.
M41	EVERY LOCAL VARIABLE HAS BEEN PROMOTED INTO ITS OWN GLOBAL BLOCK!
M42	DUMMY ARGUMENT OR ENTRY NAME <name> IS NOT EXPLICITLY TYPED.
M43	NO $SETS IN CALBLOCK CAN CRIPPLE OPTIMIZATION. ALL VARIABLES ASSUMED SET.
M44	NO $USES IN CALBLOCK CAN CRIPPLE OPTIMIZATION. ALL VARIABLES ASSUMED IN USE.
M45	INTEGER VARIABLE <name> IS USED AS A LABEL BUT NEVER "ASSIGN ''D".
M46	THE FOLLOWING IDENTIFIERS ARE NOT EXPLICITLY TYPED DUE TO IMPLICIT NONE.
M47	COMMON BLOCK <name> HAS NOT BEEN EXPLICITLY DECLARED.
M48	IF OPTIMIZING & DUMMY ARRAY <name> IS USED AS ASSUMED-SIZE, INVALID CODE MAY RESULT.
M49	COMMONS AND GLOBALS ARE NOT ALLOWED IN A SHARABLE SUBPROGRAM.
M50	SAVE/DATA STATEMENTS NOT ALLOWED IN A SHARABLE SUBPROGRAM.
M51	LOCAL IDENTIFIER <name> MAY NOT BE INITIALIZED IN A RECURSIVE SUBPROGRAM.

Code	Message Text
M52	COMMON BLOCK OR ARRAY <name> EXCEEDS COMPILER LIMIT OF 16,252,927 BYTES.
M53	EQUIVALENCE GROUP CONTAINING <name> EXCEEDS COMPILE LIMIT OF 16,252,927 BYTES.
M54	PROCEDURE NAME <name> CANNOT BE EQUIVALENCED.
M55	MORE THAN ONE EXTERNAL ENTITY WITH EXACT FIRST 8 CHARACTERS:<name>
M56	INTEGER*1 ARITHMETIC PRODUCES INEFFICIENT CODE, INTEGER*2 IS PREFERRED.
M57	$HOL OPTION AND CHARACTER TYPE USED SIMULTANEOUSLY. SEE FORTRAN 7 USER'S GUIDE.
O01	UNRECOGNIZABLE SYSTEM DIRECTIVE
O02	EOF ENCOUNTERED IN $ASSM CODE
O03	ILLEGAL REGISTER SPECIFIER - $REGS R0,F0,D0 ASSUMED
O04	NAME MISSING
O05	LINE COUNT MUST BE INTEGER GREATER THAN 10
O06	THIS DIRECTIVE ONLY ALLOWED WITHIN $ASSM BLOCK
O07	$NTEST WITH NO PRECEDING $TEST ACTIVE
O08	PRECEDING $NTEST ASSUMED
O09	STATEMENT LABEL MUST NOT BE PREVIOUSLY DEFINED
O10	MUST BE ARRAY OR CHARACTER VARIABLE NAME
O11	PRECEDING $NTRACE ASSUMED
O12	MUST BE A VARIABLE OR AN ARRAY NAME
O13	WIDTH MUST BE INTEGER BETWEEN 64 AND 131
O14	$NTRACE WITH NO PRECEDING $TRACE ACTIVE
O15	MUST BE A VARIABLE, ARRAY, OR COMMON NAME
O16	MISSING / ASSUMED
O17	ILLEGAL PROGRAM NAME - $PROG IGNORED
O18	$ON/$OFF VALUE MUST BE <= 63
O19	INTRINSIC FUNCTION CANNOT BE EXPANDED INLINE
O20	MISSING ARGUMENT
O21	FILE DESCRIPTOR SYNTAX ERROR
O22	ALL/LABEL-LIST MISSING, ALL ASSUMED
O23	SYNTAX IS ILLEGAL
O24	PRIVATE/SHARE MISSING
O25	CONFLICTING SOURCE FILE INFORMATION FOR THE SUBPROGRAM
O26	CANNOT BE A DUMMY PARAMETER
O27	CAL NAME LONGER THAN 8 CHARACTERS
O28	$INSKIP/$BEND MISPLACED, IGNORED
O29	SOURCE FILE IS NOT ON A DIRECT ACCESS DEVICE
O30	LOGICAL UNIT NUMBER IS OUT OF RANGE OR UNRECOGNIZABLE

Code	Message Text
031	LABEL DESCRIPTOR IS UNRECOGNIZABLE OR TOO LONG
032	FILE DESCRIPTOR IS UNRECOGNIZABLE OR TOO LONG
033	ILLEGAL TO RE-USE LOGICAL UNIT FROM AN ACTIVE $INCLUDE
034	LOGICAL UNIT IS NOT ASSIGNED
035	A FREE LOGICAL UNIT IS NOT AVAILABLE FOR $INCLUDE
036	INSUFFICIENT SPACE ON VOLUME TO OPEN SPECIFIED FILE
037	VOLUME IS NOT ON-LINE OR DOES NOT EXIST
038	SPECIFIED FILE DOES NOT EXIST
039	THE FILE IS PROTECTED AND CANNOT BE OPENED
040	THE FILE CANNOT BE OPENED FOR SHARED-READ ACCESS
041	FILE CANNOT BE OPENED DUE TO INSUFFICIENT SYSTEM SPACE
042	NESTING OF $INCLUDES EXCEEDS 7 LEVELS
043	INITIAL **LABEL NOT FOUND, $INCLUDE IGNORED
044	MUST BE END OF STATEMENT
045	ILLEGAL TARGET SPECIFIER
046	$INCLUDE OPTION LIST CONTAINS ILLEGAL OR UNRECOGNIZABLE OPTIONS
047	THIS STATEMENT/DIRECTIVE NOT ALLOWED IN THE INLINE DIRECTIVE FILE
048	THIS NAME IS USED AS AN ENTRY NAME ELSEWHERE
049	THIS NAME IS USED AS A SUBPROGRAM NAME ELSEWHERE
050	$ASSM NOT ALLOWED INSIDE EXPANDED ROUTINES FOR 7000 SERIES.
P01	END STATEMENT MISSING
P02	END OF STATEMENT ASSUMED HERE
P03	UNSUPPORTED FEATURE
P04	FIRST LINE OF STATEMENT IS A CONTINUATION LINE
P05	TOO MANY CONTINUATION LINES
P06	COLUMNS 1-5 OF CONTINUATION LINE ARE NOT BLANK
P07	NULL SOURCE FILE
P08	INVALID CHARACTER IGNORED
P09	SYMBOLIC NAME MUST NOT BEGIN WITH UNDERSCORE
P10	END OF STATEMENT BEFORE END OF CONSTANT
P11	DUPLICATE STATEMENT LABEL
P12	UNRECOGNIZABLE OPERATOR AFTER '.'
P13	UNRECOGNIZABLE SEQUENCE OF CHARACTERS
P14	MISSING PERIOD ASSUMED AFTER OPERATOR
P15	FLOATING POINT CONSTANT NOT ALLOWED HERE

Code	Message Text
P16	INVALID LABEL
P17	ENTRY STATEMENT NOT ALLOWED TO BE NESTED WITHIN LOOP OR IF BLOCK
P18	ENTRY STATEMENT NOT ALLOWED IN MAIN PROGRAM OR BLOCKDATA
P19	THIS STATEMENT IS OUT OF ORDER. STATEMENT IGNORED
P20	THIS LABEL DEFINED ON NON-EXECUTABLE STATEMENT
P21	THIS LABEL PREVIOUSLY USED TO REFERENCE EXECUTABLE STATEMENT
P22	THIS LABEL DEFINED ON EXECUTABLE STATEMENT
P23	ACTIVE DO-LOOP INDEX MUST NOT BE MODIFIED
P24	RETURN MAY ONLY APPEAR IN A FUNCTION OR SUBROUTINE
P25	ALTERNATE RETURN MAY ONLY APPEAR IN A SUBROUTINE
P26	NO CORRESPONDING BLOCK IF
P27	ELSE CANNOT FOLLOW ELSE
P28	ELSE PRECEDED BY UNTERMINATED LOOP
P29	ELSE IF CANNOT FOLLOW ELSE
P30	ELSE IF PRECEDED BY UNTERMINATED LOOP
P31	IF BLOCK BEGINS BEFORE LOOP AND ENDS INSIDE LOOP
P32	MISSING ENDIF
P33	MUST BE AT LEAST 2 ITEMS
P34	ILLEGAL PARENTHESIZED LIST
P35	THIS LABEL MUST NOT BE PREVIOUSLY DEFINED
P36	ILLEGAL DO NESTING
P37	THIS LABEL PREVIOUSLY USED TO REFERENCE FORMAT STATEMENT
P38	LOOP BEGINS BEFORE IF BLOCK AND ENDS INSIDE BLOCK
P39	ILLEGAL STATEMENT FOR DO TERMINAL
P40	THIS STATEMENT NOT ALLOWED IN BLOCK DATA
P41	NO PRIOR ACTIVE LOOP
P42	STEPBY CANNOT FOLLOW STEPBY
P43	NO PRIOR ACTIVE DO
P44	ENDDO OF LABELED DO MUST HAVE SAME LABEL AS THAT OF DO
P45	MISSING ENDDO
P46	MISSING REPEAT
P47	DO LOOP BEGINS BEFORE LOOP-REPEAT AND ENDS INSIDE
P48	LOOP BEGINS BEFORE DO LOOP AND ENDS INSIDE
P49	STEPBY PRECEDED BY UNTERMINATED DO LOOP
P50	STEPBY PRECEDED BY UNTERMINATED IF BLOCK
P51	ENTRY/PROG IGNORED IN INLINE ROUTINES
P52	DO INCREMENT IS ZERO, INFINITE LOOP GENERATED
P53	RECURSIVE SUBPROGRAM CANNOT BE EXPANDED INLINE

Code	Message Text
P54	MUST BE ACTIVE LOOP LABEL
P55	ZERO TRIP DO-LOOP
P56	SINGLE TRIP DO-LOOP
S01	IDENTIFIER CAN NOT BE RE-TYPED
S02	ASSUMED-LENGTH SPECIFICATION MUST NOT APPEAR HERE
S03	LENGTH SPECIFICATION IGNORED
S04	CHARACTER LENGTH MUST NOT EXCEED 32767
S05	LENGTH SPECIFICATION DOES NOT MATCH A LEGAL LENGTH FOR THIS TYPE
S06	SYMBOLIC CONSTANT MUST NOT BE TYPED AFTER BEING DEFINED
S07	COMMON BLOCK NAME CONFLICTS WITH PRIOR USAGE
S08	IDENTIFIER MUST NOT APPEAR IN BOTH COMMON AND SAVE
S09	DUMMY ARGUMENT MUST NOT APPEAR IN COMMON
S10	IDENTIFIER MUST NOT APPEAR IN COMMON MORE THAN ONCE
S11	IDENTIFIER IS MULTIPLY DEFINED
S12	DUMMY ARGUMENT NAME CONFLICTS WITH AN ENTRY POINT NAME
S13	FUNCTIONS DO NOT HAVE AN ALTERNATE RETURN CAPABILITY
S14	DUMMY ARGUMENT MUST NOT BE EQUIVALENCED OR SAVED
S15	ARGUMENT MUST NOT APPEAR TWICE IN THIS ENTRY LIST
S16	ENTRY NAME MUST NOT APPEAR IN COMMON
S17	MULTIPLY DEFINED IMPLICIT
S18	NOT AN INTRINSIC FUNCTION
S19	TOC MANY LETTERS
S20	LETTERS NOT IN ALPHABETICAL ORDER
S21	PREVIOUS PARAMETER DEFINITION IGNORED
S22	NAME HAS ALREADY DEFINED AN EMPTY NAMELIST
S23	NAMELIST NAME PREVIOUSLY REFERENCED IN AN I/O STATEMENT
S24	NAME HAS ALREADY DEFINED A NON EMPTY NAMELIST
S25	ENTRY NAME MUST NOT BE EQUIVALENCED
S26	FUNCTION REFERENCE NOT ALLOWED HERE
S27	STATEMENT FUNCTION DUMMY ARGUMENT MUST BE A VARIABLE NAME
S28	ASSUMED SIZE ARRAY NAME NOT ALLOWED HERE
S29	NAME SHOULD NOT BE SAVED MORE THAN ONCE
S30	NAME HAS PREVIOUSLY BEEN DECLARED GLOBAL
S31	NAME HAS PREVIOUSLY BEEN DECLARED LOCAL
S32	COMMON ELEMENT MUST NOT APPEAR IN GLOBAL OR LOCAL STATEMENT
S33	DUMMY ARGUMENT MUST NOT APPEAR IN GLOBAL OR LOCAL STATEMENT

Code	Message Text
S34	ILLEGAL FUNCTION REFERENCE OR UNDIMENSIONED ARRAY
S35	ADDRESSING MODE CONFLICTS WITH INITIAL SPECIFICATION. THIS MODE IGNORED
S36	MUST NOT RETYPE AN IDENTIFIER USED IN A DIMENSION DECLARATION
S37	IDENTIFIER MUST BE EITHER A VARIABLE OR AN ARRAY
S38	THIS IDENTIFIER PREVIOUSLY APPEARS ON A SAVE STMT. SAVE ATTRIBUTE IGNORED
S39	ATTRIBUTES OF IDENTIFIER ARE INCOMPATIBLE WITH SAVE. SAVE ATTRIBUTE IGNORED
S40	ENTRY NAME APPEARS IN ITS OWN DUMMY ARGUMENT LIST
S41	AN INTRINSIC MUST NOT BE A DUMMY ARGUMENT NAME
T01	ATTIBUTES OF IDENTIFIER ARE INCOMPATIBLE WITH THE SPECIFICATION OR USE
T02	EXPRESSION MUST BE OF TYPE INTEGER
T03	MUST BE AN INTEGER CONSTANT EXPRESSION
T04	MUST BE OF TYPE LOGICAL
T05	MUST BE ARITHMETIC TYPE (OTHER THAN COMPLEX)
T06	MUST BE OF TYPE CHARACTER
T07	MUST BE A CONSTANT
T08	IDENTIFIER MUST BE OF TYPE CHARACTER
T09	NO DIGITS FOLLOWING EXPONENT SPECIFICATION OF FLOATING POINT CONSTANT
T10	VALUE OF CONSTANT EXCEEDS MAXIMUM--MAXIMUM INTEGER VALUE USED
T11	NULL HOLLERITH CONSTANT NOT ALLOWED
T12	INVALID R CONSTANT
T13	INVALID QUOTED CONSTANT TYPE
T14	IDENTIFIER NAME EXCEEDS 36 CHARACTERS
T15	NULL STRING NOT IMPLEMENTED
T16	INVALID CHARACTER(S) WITHIN BIT CONSTANT
T17	INVALID CHARACTER(S) WITHIN 'E' OR 'D' CONSTANT
T18	TOO MANY SIGNIFICANT DIGITS WITHIN CONSTANT
T19	INVALID CHARACTER(S) WITHIN INTEGER OR OCTAL CONSTANT
T20	FLOATING POINT CONSTANT UNDERFLOWS--VALUE SET TO ZERO
T21	FLOATING POINT CONSTANT OVERFLOWS--VALUE SET TO MAXIMUM

Code	Message Text
T22	MUST BE INTEGER CONSTANT 0-99999 OR CHARACTER CONSTANT <= 66 CHARACTERS
T23	EXPRESSION SHOULD BE OF TYPE INTEGER
T24	LENGTH OF CHARACTER DUMMY ARGUMENT MUST BE A CONSTANT
T26	RIGHT HAND TYPE INCOMPATIBLE WITH LEFT HAND TYPE
T27	MUST BE AN INTEGER VARIABLE
T28	PREVIOUS EXPLICIT TYPING OF THIS INTRINSIC IGNORED
T29	NAME WAS PREVIOUSLY REFERENCED AS A FUNCTION
T30	NAME WAS PREVIOUSLY REFERENCED AS A SUBROUTINE
U01	<type> DIVIDE BY ZERO
U02	<type> OVERFLOW
U03	<type> DIVIDE BY ZERO
U04	<type> EXPONENT UNDERFLOW
U05	<type> EXPONENT OVERFLOW
U06	<type> ZERO BASE, NEGATIVE EXPONENT
U07	<type> NEGATIVE EXPONENT
U08	<type> NEGATIVE BASE
U09	<type> SIGN BIT IS NOT EXTENDED

Diagnostic Messages for FORTRAN VII RTL

The following information documents the error messages which are invoked as a result of an error occuring during the execution of a FORTRAN VII RTL routine. For additonal run-time error messages, consult the *OS/32 System Messages Reference Manual.*

Format:

ERR *n* (*a*)
m:*d*

Where:

n is the error message number.

m is the name of the RTL function.

a is the hexadecimal return address in the user's program. This is only printed on OS/32.

d is either a number or a diagnostic message.

Key to Symbols:

\A - ADDRESS
\D - DECIMAL NUMBER
\S - CHARACTER STRING
\Z - HEX NUMBER

All supervisor call (SVC) errors reflect the status returned from the SVC. For more information on each status, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

0 ILLEGAL ERROR NUMBER (INTERNAL ERROR)

The run-time error handler has processed an error number outside the legal range because of a discrepancy in the source program. A possible cause of this error message is that the task memory was overwritten by assignments to array elements with out-of-bounds subscripts. Also, a variable used as a format label in an input/output (I/O) statement without a valid prior ASSIGN statement can cause this error.

Use available debugging methods to isolate and correct the error(s) in the source program.

End-of-File (EOF) Errors

```
-3 EOF DETECTED ON INTERNAL FILE AT \A.
```

An end-of-file (EOF) condition was detected while peforming I/O to an internal file which is located at address \A. Add the keywords END=, ERR=, or IOSTAT= to the statements performing the I/O and add code to the program to handle the EOF condition.

```
-2 SOFT INTERNAL EOF DETECTED
```

An EOF marker (/*) was found while performing an I/O operation to a file. Add the keywords END=, ERR=, or IOSTAT= to the statements performing the I/O and add code to the program to handle the EOF condition.

```
-1 HARDWARE EOF DETECTED
```

An operating system-defined EOF condition was found while performing an I/O operation to a file. Add the keywords END=, ERR=, or IOSTAT= to the statements performing the I/O and add code to the program to handle the EOF condition.

I/O Common Errors

1 `UNABLE TO ASSIGN DEFAULT UNIT \D TO DEFAULT DEVICE \S.`

The RTL was not able to assign the default device \S to the default logical unit (lu) \D while processing a read or write. The default device \S may not exist in the system or the default unit may already be assigned. Alter the source program so it does not use reads and writes or free the required unit for reads and writes. If the default device does not exist have the default device created or added to the program's environment.

2 `UNIT ILLEGAL.`

The lu specified for the I/O statement is not between 0 and 254. The error message only shows the lower byte of the specified word. Correct the source program if the lu is greater than 254 or increase the number of logical units allowed at Link time with the LINK OPTION command parameter lu=. For more information, see the *OS/32 Link Reference Manual.*

3 `UNABLE TO GET ENOUGH STORAGE TO DO I/O.`

The RTL was unable to get the necessary workspace to build the record for performing a requested I/O operation. At run-time, increase the amount of extra memory that the program was given with the LOAD command based upon the largest record length of any file that will be processed. If the largest record length to be processed is 10K, specify a load memory expansion factor of at least 15K in the LOAD command. Optionally, set the default LINK OPTION command parameter WORKSPACE to the necessary minimum value. For more information, *OS/32 Link Reference Manual.*

4 `EXCEED RTL SCRATCH AREA.`

The attempted I/O exceeded the available RTL scratch area for creating the needed information to perform the I/O operation. This error indicates an illegal nesting of I/O operations, in most cases. This can occur when a FORTRAN VII function containing WRITE statements is called on a WRITE statement. Correct the source program by sending the function results to an intermediate variable.

5 RECORD NUMBER \D NOT POSITIVE FOR DIRECT ACCESS

The record number for a direct access I/O should be one or greater. Correct the problem within the source code of the program.

6 ONLY FORMATTED AND SEQUENTIAL ACCESS ALLOWED ON TEXTFILES

An unformatted or direct access I/O was attempted on a file that was designated as a FORMATTED and SEQUENTIAL file. Correct the problem within the source code.

SVC1 Errors

25 UNIT UNASSIGNED.

The lu used in an I/O statement does not have a file or device attached for performing I/O. Either correct the program source or assign the lu to the needed file or device so that the I/O can be performed. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

26 PARITY OR RECOVERABLE ERROR

A parity or a recoverable error, i.e., attempting to perform a write on a write-protected file while trying to write, was detected on a file or device. Retry the I/O after making the needed correction(s). If the error persists, the interface to a device may be incorrectly programmed. On a tape drive, the tape itself may be bad or the tape drive read/write heads and tape path may need to be cleaned. This error can occur for various reasons, including time-outs and other events that depend on the driver that is attached to a device. The system's error logger should be checked for errors that may have been logged by the particular driver. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

27 UNRECOVERABLE ERROR

An unrecoverable error was detected during a FORTRAN VII I/O operation. This error can be caused by format failures on a disk drive, tapes which are physically damaged, and various other reasons. The system's error logger should be checked for errors that may have been logged by the particular driver that was in use. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

28 END OF MEDIUM

This error indicates that an I/O detected an end-of-medium condition during an I/O to a device. This error generally means that the end of a contiguous file has been reached or that the end of a tape has been reached. If a non-contiguous file could not be extended, this error code can also be returned. Possible solutions include using a longer tape, deleting unnecessary files from the existing tape, or adding code to the program to handle the error. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

29 DEVICE UNAVAILABLE

The device or file to which the I/O was directed is not available to perform the I/O. Place the device or file on-line and continue the program execution. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

30 ILLEGAL FUNCTION

The I/O operation being directed to the specified file or device is not allowed. Writing to a file that is read-only causes this error. Correct the problem in the source code of the program or alter the access privileges of the file or device. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

31 DIRECT ACCESS READ ENCOUNTERS EOF RECORD

While performing a direct access read the I/O detected an EOF record. Either correct the program that generated the file or the program that is reading the file. Also, the file being read may be corrected manually and the program execution restarted. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

32 DIRECT ACCESS READS BEYOND END OF INDEX FILE

A read was attempted beyond the end of the file. This is generally a programming problem. Correct the program that generated the file or is reading the file. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

33 ACCESS PRIVILEGES INCOMPATIBLE

An I/O attempt was made on a default logical unit without preassignment. The logical unit is assigned to a file with access privileges which are incompatible to the current I/O privilege. For more information see the *OS/32 SVC Reference Manual.*

SVC7 Errors

50 `ILLEGAL FUNCTION CODE ON SVC7`

An illegal function code was used for an SVC7. You may be running under an unsupported version of the operating system or the function code may not be supported on the particular file or device. Verify that the program is running under the correct release of the operating system or that the SVC7 function is supported for the file or device. If the file is on an optical disk, certain SVC7 are not supported. If the problem cannot be resolved, contact your local Concurrent Computer Corporation service office for assistance. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

51 `ILLEGAL LOGICAL UNIT \D`

The lu specified is illegal. ChecK the lu that was specified and make the necessary correction in the program source code. This error can be caused by specifying a negative number for the lu in an I/O statement. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

52 `SPECIFIED VOLUME NOT MOUNTED OR NON-EXISTENT`

The volume name that was specified in the OPEN, INQUIRE, etc. statement(s) does not exist or is not on-line. Correct the program source code if the volume name(s) is specified in the source code of the program or correct the data that contains the volume name(s). If the volume is not on-line then mark the volume on-line and continue the program. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

53 `ATTEMPT TO ALLOCATE OR RENAME USING EXISTING FILENAME`

An attempt was made to allocate or rename a file with a filename that already exists on the specified volume. Either remove the file or use a different filename for the allocate or rename operation. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

54 ASSIGN TO NON-EXISTENT FILE

The filename specified in the OPEN statement does not exist on the specified volume. Either allocate the file or alter the OPEN statement to create the file if the file does not exist. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

55 INSUFFICIENT SPACE EXISTS FOR ALLOCATE

The specified file in the OPEN statement could not be allocated because the space needed for the file was not available on the disk volume. This error can also occur during a close operation if the system buffers cannot be written to disk due to insufficient space when a file is closed. Either free up the space on the specified disk volume or use another volume. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

56 READ AND WRITE KEYS DO NOT MATCH ON ASSIGN

An attempt was made to assign a file which has protection keys that do not match the protection keys specified in the OPEN statement. Specify the correct protection keys for the file in the OPEN statement. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

57 ATTEMPT TO ALLOCATE WHEN ENTIRE DISK ERW

An attempt was made to allocate a file when the specified volume was assigned exclusive read/write (ERW). This can occur during the time the command processor's MARK command is being processed. Redo the allocation after the disk volume is no longer assigned ERW. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

58 ACCESS PRIVILEGES CANNOT BE GRANTED ON ASSIGN

The desired access privileges cannot be granted. This generally occurs when the file is currently assigned. For example, an ERW assign is not compatible with a current assignment that is SRW. Either wait for the file to be freed or change the desired access privileges. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

59 INCOMPATIBLE ACCESS PRIVILEGES ON CHANGE PRIVILEGES

The desired access privileges cannot be granted. This generally occurs because the file is currently assigned to another lu or when a file is assigned with an account number and is not linked with the ACPRIVILEGE option. For example, an EWO access privilege is not compatible with another assignment that is SRW. Wait for the file to be freed or change the desired access privileges. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

60 FILE NOT ASSIGNED ERW ON REPROTECT OR RENAME

To reprotect or rename a file, the file must currently be assigned ERW. Change the access for the file to ERW. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

61 ATTEMPT TO DELETE A FILE WHICH IS NOT CLOSED BY ALL TASKS

The file that is to be deleted is currently assigned to one or more logical units of one or more tasks. Correct the source code if the file should not have been assigned, or wait until the file is no longer assigned to any logical units. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

62 EXCEED ALLOCATED SPACE ON ASSIGN

The file cannot be assigned because the task would exceed the allowable amount of system space. Use PATCH or LINK to increase the amount of system space that the task is allowed. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual*, *OS/32 Link Reference Manual*, or *OS/32 Patch Reference Manual.*

63 ATTEMPT TO ASSIGN ALREADY ASSIGNED LOGICAL UNIT \D

An attempt was made to assign a file to an lu that is currently assigned to a file. Change the source program to use a different lu for the file. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

64 `LOGICAL UNIT \D NOT ASSIGNED`

An attempt was made to perform a SVC7 function (other than an ASSIGN function) that requires an assigned lu on an lu that is not currently assigned. Assign the lu prior to attempting the specific SVC7 operation either by correcting the program or manually assigning the lu. For more information, see the or *OS/32 Supervisor Call (SVC) Reference Manual.*

65 `SPECIFIED VOLUME NOT A DIRECT ACCESS DEVICE`

Direct access was specified on a file or device which does not support direct access. Make the necessary changes to the source program. Ensure that you did not specify DIRECT access for a device that does not support direct access.

66 `FILE DESCRIPTOR FORMAT BAD ON SVC7`

The file descriptor specified is not valid. This could mean that an account number was specified when the program does not have account privileges. It may also occur when the filename is illegal or contains meaningless data. Correct the program's source code.

67 `ASSIGN: DEVICE NON-EXISTENT, NON-CONNECTABLE, OR BUSY`

The device does not exist, is not a trap generating device, or is already connected to another task. Correct the source code to use a trap generating device that exists or is not already in use. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

68 ALLOCATE OR DELETE ATTEMPTED ON SYSTEM OR GROUP FILE

An attempt was made to allocate or delete a file that is on the group or system account. Correct the source code so it does not attempt to delete or allocate files on the GROUP or SYSTEM account. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

69 SVC7 ERROR: \Z

An unknown SVC7 error occurred that does not have a meaning within the context of the run-time library (RTL). Refer to the *OS/32 Supervisor Call (SVC) Reference Manual* for the meaning of the SVC7 error. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual.*

Format Translator Errors

100 FORMAT DOES NOT BEGIN WITH '('.

The format specified in the I/O statement is not correctly formatted. The format statement must be enclosed in parentheses. Correct the format statement.

101 UNPRINTABLE CHARACTER ENCOUNTERED IN FORMAT

The specified format contains unprintable characters. Correct the format statement.

102 NESTING LEVEL OF '(' IN FORMAT EXCEEDS 255

The specified format statement is too complex. The number of parenthesis pairs nesting is limited to 255. Simplify the format by reducing the number of nested parenthesis pairs or break the I/O format into multiple I/O and format statements.

103 REPEAT COUNT NOT ALLOWED FOR EDIT DESCRIPTOR

A format specifier was given a repeat count when it is not allowed to have a repeat count. Correct the format by simplifying and removing the illegal format repeat specifier.

104 MISSING DIGITS BEFORE P OR H SPECIFIERS

The format specifiers P (Precision) and H (Hollerith) require a number to preceed the actual format specifier. Correct the format by inserting a number before the P/H format specifier or remove the P/H specifiers.

105 MEMORY OVERFLOW DURING FORMAT TRANSLATION

The format translator exhausted the available memory used for translating the format statement. Rerun the program with more memory or simplify the format.

106 NUMBER EXCEEDS HALFWORD IN FORMAT

A number that is used as part of the format (not a number to be read in or written out) is greater than 32767. Simplify the format to use a value that is less than 32767.

107 CHARACTER ILLEGAL OR NOT ALLOWED HERE IN FORMAT

An illegal character or an illegal use of a character was found in the format. Correct the format.

108 ILLEGAL SIGN ENCOUNTERED IN FORMAT

An illegal use of a sign was found within the format. Correct the format.

109 D FIELD EXCEEDS FIELD WIDTH

The D field format identifier was specified so that the total length of the field will be exceeded by the other parameters in the D field format. Correct the D field format specifications.

110 NUMBER OF EXPONENT DIGITS EXCEEDS FIELD WIDTH

The number of exponent digits that was specified in the format item will exceed the size of the format field width. Correct the format specifier.

111 ZERO REPEAT COUNT IN FORMAT

A format item was specified with a repeat count of zero. Correct the number or remove the format specifier from the format.

112 FIELD WIDTH MISSING OR ZERO IN FORMAT

A field width was omitted or was specified as a 0. Correct the format by adding the field width, changing the 0 to a valid value, or removing the format identifier.

113 D FIELD MISSING IN FORMAT

A format specifier that requires a D field had the D field omitted from the format. Correct the format by including the missing fields in the format.

114 EXPONENT FIELD MISSING, ZERO, OR GREATER THAN 255 IN FORMAT

An error was found in a format field that either requires an exponent field or the exponent field specified is not between 1 and 255 inclusive. Correct the format by including the missing exponent field or changing the size of the field to be between 1 and 255 inclusive.

115 RTL STACK OVERFLOW DURING FORMAT TRANSLATION

The format caused the RTL stack to overflow while it was being processed. Simplify the format and check for programming errors that may be causing the recursive format translator calls.

Formatted I/O Errors

150 WRITE ATTEMPT EXCEEDS RECORD SIZE \S

An attempt was made to write a record to a file that would result in the truncation of the formatted record. Reduce the size of the formatted record or increase the record length of the file to be able to handle the size of the formatted record's length.

151 INPUT RECORD TOO SHORT FORM FORMAT \S

The record length of the file that is being read is not long enough to handle the specified format. Correct the use of the format or make the necessary changes in the record length of the particular file.

152 ATTEMPT FORMATTED I/O ON LU \D, WHICH HAS NOT BEEN SO DESIGNATED

A formatted I/O was executed on an lu (\D) which was not designated for formatted I/O operations. Correct the program appropriately so as not to perform formatted I/O on logical units that are not designated for formatted I/O operations.

153 WRITE BEYOND END OF INTERNAL FILE AT \A

A formatted write operation, if completed, would overwrite an internal file. Make the necessary changes to the I/O statement or the internal file so that the I/O operation does not write beyond the end of the internal file.

154 DIRECT ACCESS NOT ALLOWED

A direct access I/O was attempted where direct access to a file is not allowed. Correct the source program.

155 ARGUMENT EXISTS, BUT FORMAT REQUIRES NONE

An argument was specified on a formatted write statement, but the format does not have any format specifiers that operate with arguments (i.e., I, F, A, or G format specifiers). Correct the format itself or remove the argument(s) from the I/O statement.

156 QUOTE OR HOLLERITH EDIT ILLEGAL IN INPUT FORMAT

The format used in a formatted read contains quoted information or a Hollerith string. Correct the format. A format used for input may not contain quoted strings or Hollerith strings.

157 TYPE MISMATCH BETWEEN \S EDIT AND \S

The variable that was specified for the particular format identifier is not compatible with the variable that is being written or read. Either correct the format that was used for the I/O or correct the data that was being read or written. If the program paused, a CONTINUE command can be entered and the format identifier will be adapted to the variable.

158 ILLEGAL INPUT FOR \S EDIT IN FORMAT

The data entered for the format specifier is illegal for the particular format identifier. Correct the format if it does not accept the correct data or enter the correct type of input data.

159 INPUT VALUE UNDERFLOW IN \S EDIT \S

The value that was entered for the specified format has underflowed during the conversion to the binary format. Increase the variable that is being used for the input to a double precision variable or make the necessary corrections to the number that is being input.

160 INPUT VALUE OVERFLOW IN \S EDIT \S.

The value that was entered for the specified format overflowed during the conversion to the binary format. Increase the variable that is being used for the input to larger integer or floating point value or make the necessary corrections to the number that is being input.

175 ATTEMPT UNFORMATTED I/O ON LU \D, WHICH HAS NOT BEEN SO DESIGNATED

An unformatted I/O was attempted on an lu (\D) which was not designated for unformatted I/O operations. Make the necessary corrections in the program and I/O statements.

176 UNFORMATTED INPUT RECORD TOO SHORT

The unformatted input record is not long enough for the I/O that was attempted on the file. Correct the program that generated the data record or correct the program that is reading the data record.

177 ATTEMPT AN UNFORMATTED I/O GREATER THAN RECL: \D

An attempt was made to perform an unformatted I/O greater than the record length (\D) of the file. Correct the program that generated the data record or correct the program that is reading or writing the data record.

178 ATTEMPT TO READ MORE THAN WAS WRITTEN

An attempt was made to read more data from a record than was actually written. Correct the program that generated the data record or correct the program that is reading or writing the data record.

179 ATTEMPT BINARY I/O ON UNIT \D -- ILLEGAL RECL

Binary I/O was attempted on the specified lu (\D) and the record length of the file is illegal. Make the necessary changes to the program that is generating the I/O operation.

List-Directed I/O Errors

```
200 MISMATCH IN TYPE OF DATA AND VARIABLE IN \S INPUT
```

The data that was entered does not match the type required by the variable being input. Correct the data or include in the INPUT statement the ERR= or IOSTAT= keywords to handle the mismatch of the data and the variable type.

```
201 OVERFLOW IN \S INPUT
```

During the conversion of the input data for the list-directed or namelist input a conversion overflow occurred. Either correct the data or add the keywords ERR= or IOSTAT= to the I/O statement to handle the error condition.

```
202 INVALID CHARACTER IN \S INPUT
```

An invalid character was found while processing the list-directed or namelist input. Correct the input data or add the keywords ERR= or IOSTAT= parameter to the I/O statement to handle the error condition.

```
203 MISSING BEGINNING QUOTE IN \S INPUT
```

The necessary leading quote character needed for the \S edit descriptor was not found. Correct the data by adding the necessary quote or adding the keywords ERR= or IOSTAT= to handle the error condition.

```
204 STRING OF LENGTH ZERO IN \S INPUT
```

The data input for the list-directed or namelist input is a null string. Correct the data input. If the data is generated by other sources add the keywords ERR= or IOSTAT= to detect the error condition.

205 ADDITIONAL STORAGE COULD NOT BE OBTAINED IN \S INPUT

During the processing of the I/O, the necessary additional memory could not be obtained for processing input. Run the program with more memory, make the necessary additions to the I/O statement by adding ERR= or IOSTAT=, or correct the errors in the program.

206 CHARACTER STRING > 32K IN \S INPUT

The I/O statement attempted to read a string that is greater than 32kB in size while processing the list-directed or namelist input. Correct the program and or data so the program does not attempt to read a string more than 32kB in size.

207 REAL VALUE UNDERFLOW IN \S INPUT

During the conversion of the input data for the input variable a conversion underflow occurred. Either correct the data or add the keywords ERR= or IOSTAT= parameter to the I/O statement to handle the error condition.

208 MISSING OPEN PARENTHESIS IN \S INPUT

The necessary open parenthesis is missing from the input data. Correct the input data and add the keywords ERR= or IOSTAT= to allow the program to handle the error condition.

209 END OF RECORD NOT ALLOWED HERE IN \S INPUT

An end of record condition was detected during the input processing where its occurrence is not allowed. Correct the input data and add the keywords ERR= or IOSTAT= to allow the program to handle the error condition.

210 PART OF COMPLEX CANNOT BE NULL IN \S INPUT

A complex number was input but a part of the number was not specified. Correct the input data and add the keywords ERR= or IOSTAT= to allow the program to handle the error condition.

211 MISSING VALUE SEPARATOR IN \S INPUT

The input data is missing the required value separator between each variable input item. Correct the input data and add the keywords ERR= or IOSTAT= to allow the program to handle the error condition.

212 REPLICATION FACTOR OF ZERO NOT ALLOWED IN \S INPUT

A format replication was specified where it is not allowed or was set to 0 which is not allowed. Correct the input data and add the keywords ERR= or IOSTAT= to allow the program to handle the error condition.

213 RECORD LENGTH < MAXIMUM REQUIRED TO OUTPUT VALUE IN \S OUTPUT

The record length of the file is too small to perform the I/O. Correct the record length of the file or the output statement that is generating the output.

Namelist I/O Errors

225 INVALID VARIABLE NAME SYNTAX IN NAMELIST INPUT

A namelist input variable name was detected that contains a syntax error. Correct the namelist input data.

226 END OF RECORD BEFORE '=' IN NAMELIST INPUT

The end-of-record was detected after a namelist variable and before the equal sign ('=') was found. Correct the namelist data. Also add the ERR= or IOSTAT= parameters to allow for checking of errors by the program.

227 VARIABLE NOT IN NAMELIST

A variable was specified in the namelist data that is not in the NAMELIST statement. Correct the data or add the variable to the NAMELIST statement.

228 '=' MISSING AFTER VARIABLE NAME IN NAMELIST INPUT

The equal sign ('=') was not placed after the variable name. Correct the namelist data to include the necessary equal signs.

229 SCALAR VARIABLE CANNOT HAVE SUBSCRIPTS IN NAMELIST INPUT

A scalar (nonarray) variable was specified in the input data with array subscripts. Correct the data. If the variable is suppose to be an array make the corrections in the source code.

230 TOO FEW SUBSCRIPTS IN NAMELIST INPUT

The input variable name in the NAMELIST data was specified without the correct number of subscripts. Correct the input data or make the necessary corrections to the source program.

231 SUBSCRIPT OUT OF RANGE IN NAMELIST INPUT

The specified subscript in the input data is not in the correct range for the array variable specified. Correct the input data or if it is correct make the necessary corrections in the source code of the program.

232 TOO MANY SUBSCRIPTS IN NAMELIST INPUT

The input variable name in the NAMELIST data was specified without the correct number of subscripts. Correct the input data or make the necessary corrections to the source program.

233 TOO MANY VALUES FOR ARRAY IN NAMELIST INPUT

Too many data values were specified for an array in the NAMELIST input data. Correct the input data by reducing the number of input items for the array that is being processed.

234 RECORD LENGTH < 38 FOR NAMELIST I/O

The record length of the file that is being used with NAMELIST I/O is less than 38 bytes in length. Increase the record length of the appropriate file to be at least 38 bytes in length. This correction can be done from the OS level if necessary or from within the program.

235 QUOTE IN CHARACTER DATA HAS BEEN SPLIT ACROSS RECORD IN NAMELIST OUTPUT

A quote in a namelist output (designated by consecutive single quotes) causes one quote to occur at the end of a record and its corresponding quote to occur at the beginning of the next record. Make the corrections in the program so the data item does not split.

236 ERROR IN SUBSTRING SPECIFICATION IN NAMELIST INPUT

An error was found in the substring specification in the namelist data item. Correct the substring specification in the input data.

237 TOO MANY SIGNIFICANT DIGITS IN HEX CONST IN NAMELIST INPUT

The hex constant being processed has too many digits (less than eight significant digits for a 'Y' constant; less than four significant digits for an 'X' constant). Correct the input data so it does not have an excessive number of digits.

238 CR IN HEX, HOLLERITH, OR R CONST IN NAMELIST INPUT

A carriage return was detected inside of HEX, HOLLERITH, or R CONST data item during the NAMELIST input. Correct the NAMELIST data.

239 R CONSTANT LONGER THAN 4 CHARS IN NAMELIST INPUT

The R constant that was specified in the NAMELIST input is longer than four characters in length. Correct the NAMELIST input data.

240 ZERO LENGTH IN HOLLERITH OR R CONSTANT IN NAMELIST INPUT

The Hollerith data or R constant length in the NAMELIST input is zero digits in length. Correct the NAMELIST input data.

241 NO DIGITS IN HEX CONSTANT IN NAMELIST INPUT

The Hex constant in the NAMELIST data does not have any digits. Correct the NAMELIST data input.

Auxiliary I/O Errors

300 NO MATCH FOUND IN AUX. I/O PARAMETER LIST (INTERNAL ERROR)

While processing an auxiliary I/O statement, a keyword was encountered that is not valid for this particular function. Verify that the program did not cause the problem by overwriting arrays or other data areas. Make the necessary correction to the program if this is the cause. If the program is not the cause, please report the problem with a software change request (SCR) and supporting documentation.

301 VALUE \D FOR SPECIFIER > MAX VALUE ALLOWED: \D

The value for the specifier (\D) is greater than the maximum value allowed. An example is specifying a value of 256 for a file's blocking factor when 255 is the maximum blocking factor allowed for files. Make the necessary corrections to the values that are being passed to the auxiliary I/O statement or perform program validation on the program's input.

302 INVALID SPECIFIER CODE TYPE (INTERNAL ERROR)

While processing an auxiliary I/O statement a specifier was encountered that is not valid for this particular function. Verify the program did not cause the problem by overwriting arrays or other data areas. Make the necessary correction to the program if this is the cause. If the program is not the cause, please report the problem with a SCR and supporting documentation.

303 LENGTH NOT PRESENT WITH CHARACTER ADDRESS (INTERNAL ERROR)

A character item was found on the argument list passed to the auxiliary I/O which was not followed by a character length. Verify the program did not cause the problem by overwriting arrays or other data areas. Make the necessary correction to the program if this is the cause. If the program is not the cause, please report the problem with a SCR and supporting documentation.

304 MNEMONIC STRING INVALID: \S

The MNEMONIC string passed to the auxiliary I/O statement is not a valid mnemonic for this particular auxiliary I/O statement. Correct the MNEMONIC string that was passed to the auxiliary I/O statement. Also ensure that the program has not corrupted the program memory space.

305 NO UNIT SPECIFIER PRESENT IN AUXILIARY I/O STATEMENT

The auxiliary I/O statement does not have a [UNIT=]lu specifier present. All auxiliary I/O statements require that a lu be specified. Correct the auxiliary I/O statement by adding the [UNIT=]lu parameter to the auxiliary I/O statement.

306 OPEN STMT: FILE SPECIFIER NOT PRESENT AND RENAME='YES'

A request to renew a file cannot be performed because the new name for the file was not specified. Correct the auxiliary I/O statement by including the new name of the file in the auxiliary I/O statement.

307 OPEN STMT: BLANK SPECIFIER ONLY ALLOWED FOR FORMATTED FILES

The auxiliary I/O statement opened a file and specified the BLANK= specifier, but the open is not for a FORMATTED (ASCII only) file.

308 OPEN STMT: FILE SPECIFIER NOT PRESENT AND STATUS OLD, NEW,
OR RENEW

The auxiliary I/O statement opened a file and specified the STATUS= parameter but no file descriptor was specified for the auxiliary I/O OPEN statement to use. Correct the auxiliary I/O statement by including the FILE= specifier to allow the auxiliary I/O statement to operate.

309 OPEN STMT: ATTEMPT TO CREATE A NAMED SCRATCH FILE

An attempt was made to open a scratch file (temporary file) as a permanent file. Correct the auxiliary I/O statement by either not using a file descriptor (i.e., specifying FILE=fd) for the temporary file or not specifying the file as a temporary file.

310 DIRECT ACCESS, COUNTBY=RECORD, AND RECL NOT SPECIFIED

An auxiliary I/O statement was specified that requires that COUNTBY='RECORD' and the record length of the file to be specified. Make the necessary correction to the auxiliary I/O statement so it functions correctly.

311 OPEN STMT: COUNTBY=SECTOR AND FILE TYPE IS NOT CONTIG OR EC

The auxiliary I/O statement specified COUNTBY='SECTOR' and the file is not a contiguous nor an extendible contiguous file. Make the necessary program correction to the auxiliary I/O statement and the program design, as required.

312 OPEN STMT: TYPE SPECIFIED INCOMPATIBLE WITH EXISTING FILE

The TYPE= specifier was specified on an auxiliary I/O statement and the file that exists with the particular file descriptor is not compatible with the specified value of the TYPE= parameter. Make the necessary corrections to the auxiliary I/O statement so it operates correctly.

313 OPEN STMT: SIZE SPECIFIER NOT PRESENT & FILE TYPE=CONTIG, EC OR LR

The auxiliary I/O statement requires the SIZE= parameter to open the specified file. Make the necessary correction to the auxiliary I/O statement so it operates correctly.

314 OPEN STMT: STATUS=NEW FOR EXISTING FILE

The auxiliary I/O statement does not expect the specified file to exist, yet it does. Make the necessary corrections to the auxiliary I/O statement so it either renews (reallocate) the file, deletes the file, or handles the error, as needed.

315 OPEN STMT: STATUS=OLD ON FILE WHICH DOES NOT EXIST

A file which does not exist was specified for the auxiliary I/O statement. Make the necessary corrections to the auxiliary I/O statement so it renews (reallocate) the file or handles the error as needed.

316 OPEN STMT: ATTEMPT TO ALLOCATE A DEVICE

An illegal attempt was made to allocate a device. Make the necessary corrections to the auxiliary I/O statement so that it does not attempt to allocate a device.

317 OPEN STMT: ATTEMPT TO CHANGE FILE FORM ON USED LU

An attempt was made to change the FORM of the file assigned to an lu, (i.e., by specifying UNIT=), after it was opened and written to or read. Make the necessary corrections by closing the file and reopening it with the desired attributes or by not attempting to change the FORM of the file attached to the lu.

318 OPEN STMT: ATTEMPT TO CHANGE BLOCKSIZE ON USED LU

An attempt was made to change the BLOCKSIZE on a file that is attached to an lu and had an I/O operation performed on it. Make the necessary corrections to the auxiliary I/O statement so it specifies the correct BLOCKSIZE of the file that is assigned to the specified lu (UNIT=).

319 OPEN STMT: ATTEMPT TO CHANGE RECORD LENGTH ON USED LU

An attempt was made to change the record length of the file that is assigned to the lu (UNIT=) after the file had an I/O operation performed on it. Make the necessary corrections to the auxiliary I/O statement so it specifies the correct record length of the file that is assigned to the specified lu (UNIT=).

320 `OPEN STMT: ATTEMPT TO CHANGE FILE TYPE ON USED LU`

An attempt was made to change the file type (indexed, contiguous etc.) of the file that is assigned to the lu (UNIT=) after the file had an I/O operation performed on it. Make the necessary corrections to the auxiliary I/O statement so it specifies the correct file type of the file that is assigned to the specified lu (UNIT=).

321 `OPEN STMT: LU NOT ASSIGNED ON REPROTECT`

An attempt was made to reprotect a file that is not currently assigned to an lu. Assign the file to an lu with exclusive read/write access and then reprotect the file.

322 `OPEN STMT: LU NOT ASSIGNED ON RENAME`

An attempt was made to rename a file that is not currently assigned to an lu. Assign the file to an lu with exclusive read/write access and then rename the file.

323 `OPEN STMT: SIZE = \D > CURRENT CONTIG FILE SIZE OF \D`

An attempt was made to assign an existing contiguous file with a specified size, but the number of sectors specified in the SIZE= parameter is greater than the actual number of sectors in the file. Make the necessary correction to the OPEN statement by renewing the file to the desired size or omitting the SIZE= parameter.

324 `OPEN STMT: FORMATTED RECL \D > CURRENT RECL \D`

An attempt was made to assign an existing file with a specified record length, but the record length specified in the RECL= parameter is greater than the actual record length of the file. Make the necessary correction to the OPEN statement by renewing the file to the desired record length or omitting the record length parameter.

325 OPEN STMT: READ AND WRITE KEYS MUST BE SPECIFIED ON REPROTECT

An attempt was made to reprotect a file (i.e., REPROTECT='YES'), but the read and/or write keys were not specified (i.e., RKEY=, WKEY=). Correct the auxiliary I/O statement so it specifies the read and write protection keys for the reprotect option of the auxiliary I/O statement.

326 OPEN STMT: DIRECT ACCESS SPECIFIED BUT RANDOM NOT SUPPORTED

An attempt was made to open a file with direct access, but direct access is not supported on the file or device being assigned. This may occur if you try to open a device (e.g., CON:, PR:, etc.) with ACCESS='DIRECT'. Make the necessary correction to the auxiliary I/O statement.

327 OPEN STMT: UNFORMATTED SPECIFIED BUT BINARY NOT SUPPORTED

An attempt was made to issue an auxiliary I/O statement that is requesting a binary file format, but the device or file does not support binary I/O. For example, opening a device (such as PR:) with FORM='UNFORMATTED' parameter. Make the necessary corrections to the auxiliary I/O statements.

328 OPEN STMT: UNFORMATTED BLOCKSIZE \D GREATER THAN FILE RECL \D

The specified BLOCKSIZE is greater than the record length of the file that is being opened or created. Make the necessary corrections to the auxiliary I/O statement so that the BLOCKSIZE is not greater than the record length of the file.

329 CLOSE STMT: STATUS=KEEP ON CLOSE STATEMENT AND FILE IS SCRATCH FILE

An attempt was made to close and retain a file that is a scratch (temporary) file. Correct the auxiliary I/O statement as required.

330 INQUIRE STMT: NEITHER UNIT NOR FILE SPECIFIER PRESENT

An auxiliary I/O INQUIRE statement was executed and neither UNIT= nor FILE= was specified. Correct the auxiliary I/O statement so that one or both of the keywords UNIT= or FILE= is specified.

331 INQUIRE STMT: NO LU AVAILABLE TO ASSIGN FILE

There are no logical units available to the auxiliary I/O statement to assign the specified file to perform the inquire function. This may occur if you are inquiring by FILE=, in which case the INQUIRE routine requires an extra lu to assign temporary files. Close an lu so the INQUIRE statement will execute or relink/patch the task with more logical units for the task to use during execution. For more information, see the *OS/32 Link Reference Manual* or *OS/32 Patch Reference Manual.*

332 LU \D BUSY AND UNUSABLE

The lu specified by \D is currently unavailable. This error may occur if, for example, a function called as one of the I/O list items in a write statement performs I/O on that lu. Correct the source code so this does not occur.

333 BACKSPACE COUNT NOT POSITIVE: \D

The count for an auxiliary I/O BACKSPACE statement should be one or greater. Correct the count for the auxiliary I/O statement so that it is one or greater. If the intention is to rewind the file then use the auxiliary I/O REWIND statement.

334 OPEN STMT: RECL NOT POSITIVE: \D

The record length on an auxiliary I/O statement is not positive. Correct the auxiliary I/O statement so that the specified record length is positive. Add the validation code to the program as needed.

335 BLOCKSIZE NOT POSITIVE OR TOO SMALL FOR UNF. I/O: \D

The BLOCKSIZE specified for an auxiliary I/O statement is not positive or is too small for the unformatted I/O that is being requested. Correct the auxiliary I/O statement so that the specified BLOCKSIZE is positive and is large enough for the specified unformatted I/O.

336 OPEN STMT: SIZE NOT POSITIVE: \D

The value specified with the SIZE= parameter should be greater than or equal to one (or zero, if being used with an index file that is being reallocated or allocated). Make the necessary corrections to the value specified with the SIZE= parameter.

337 OPEN STMT: ISIZE NOT POSITIVE: \D

The value specified with the ISIZE= parameter should be greater than or equal to one (or zero, if being used with an index file that is being reallocated or allocated). Make the necessary corrections to the value specified with the ISIZE= parameter. If the value is negative, the program probably has overwritten arrays or other areas of the program task space.

338 OPEN STMT: ATTEMPT TO CHANGE COUNTBY ON USED LU

An attempt was made to change the COUNTBY method on an lu that is currently opened. Correct the auxiliary I/O statement so that it does not change the COUNTBY= mode of the opened lu.

Pack fd Errors

```
349   INVALID FILE SPECIFIER OR ERROR ON PACK FILE DESCRIPTOR-SVC2
```

An error was detected on the file descriptor defined by the file descriptor specifiers of the auxiliary I/O statements. Correct the filename that is being specified to the auxiliary I/O statement if the file descriptor contains errors. If the file descriptor is correct and the error is being reported, ensure that the file descriptor format is valid for the version of the FORTRAN VII RTL being used. The FORTRAN RTL supplied as part of the FORTRAN VII R05.01 package does not support account numbers (i.e. /1 /0 /345 etc.). Also, ensure that the program task image has the ACPRIVILEGE option specified if the program requires account numbers. If the program is being run under the multi-terminal monitor (MTM) then this error may be generated. For more information, see the *OS/32 Supervisor Call (SVC) Reference Manual*, *OS/32 Link Reference Manual*, or *OS/32 Patch Reference Manual.*

Math Errors

500 `\S:  TOO MANY ARGUMENTS`

The subroutine or function was called with too many arguments. Make the necessary corrections to the source code so that the routine is called with the correct number of arguments.

501 `\S:  TOO FEW ARGUMENTS`

The subroutine or function was called with too few arguments. Make the necessary corrections to the source code so that the routine is called with the correct number of arguments.

502 `\S:  AN ARGUMENT OF INCORRECT TYPE`

One of the arguments to the subroutine or function is not the correct type. Make the necessary corrections to the source code so that the routine is called with the proper argument type (i.e., INTEGER*4, REAL*4 etc.).

503 `\S:  ZERO ARGUMENT`

An argument with a value of zero was passed to a routine which does not allow the value zero. Make the necessary corrections to the source code so that the routine is called with the correct values for the arguments.

504 `\S:  NEGATIVE ARGUMENT`

An argument with a negative value was passed to a routine which does not allow negative argument values. Make the necessary corrections in the source code so that the routine called with the correct values for the arguments.

505 `\S:  ARGUMENTS (0,0)`

The arguments of the routine are zero and the routine does not allow zero arguments. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

506 \S: ARGUMENT TOO LARGE

The absolute value of the argument is too large for the routine that was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

507 \S: ARGUMENT OUT OF RANGE : POSITIVE

The specified routine was called with an argument that is greater than the maximum value allowed for the routine invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

508 \S: ARGUMENT OUT OF RANGE : NEGATIVE

The specified routine was called with an argument that is less than the minimum value allowed for the routine invoked. Make the necessary corrections in the source code so that the invocation of the routine is with the correct values for the arguments.

509 \S: REAL PART TOO LARGE

A routine was invoked with an imaginary number which contains a real value portion with an absolute value that is too large for the routine which was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

510 \S: REAL PART OUT OF RANGE : POSITIVE

A routine was invoked with an imaginary number which contains a real value portion that is greater than the maximum allowed for the routine that was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

511 \S: REAL PART OUT OF RANGE : NEGATIVE

A routine was invoked with an imaginary number which contains a real value portion that is less than the minimum allowed for the routine that was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

512 \S: IMAGINARY PART TOO LARGE

A routine was invoked with an imaginary number which contains an imaginary value portion with an absolute value that is too large for the routine that was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

513 \S: IMAGINARY PART OUT OF RANGE : POSITIVE

A routine was invoked with an imaginary number which contains an imaginary value portion that is greater than the maximum allowed for the routine that was invoked. Make the necessary corrections in the source code so that the invocation of the routine is with the correct values for the arguments.

514 \S: IMAGINARY PART OUT OF RANGE : NEGATIVE

A routine was invoked with an imaginary number which contains an imaginary value portion that is less than the minimum allowed for the routine that was invoked. Make the necessary corrections in the source code so that the invocation of the routine is with the correct values for the arguments.

515 \S: OVERFLOW ON CONVERSION

An overflow occurred during the conversion of a number while it was being processed by the specified routine. Make the necessary corrections in the source code so that the program can handle the conversion of the number.

516 \S: VALUE OF SIZE IS ILLEGAL

The specified size of the value is illegal for the routine that was invoked. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

517 \S: EXPONENTIAL OVERFLOW

An exponential overflow occurred while program execution was located in the routine \S. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

518 `\S:  EXPONENTIAL UNDERFLOW`

An exponential underflow occurred while program execution was located in the routine \S. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

519 `\S:  ZERO DIVISOR`

An operation within the routine \S is attempting to use zero as a divisor. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

520 `\S:  ARGUMENTS (ZERO,NEGATIVE)`

The routine \S was called with arguments that are less than or equal to zero when only arguments with positive values are allowed. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

521 `\S:  NEGATIVE EXPONENT`

A routine was invoked with an argument that has a negative exponent when only numbers with positive exponents are allowed. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

522 `\S:  ZERO BASE, NEGATIVE EXPONENT`

The routine \S was invoked with an argument that is zero or has an exponent value that is negative. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

```
523   \S:  NEGATIVE INDEX
```

A negative index was passed to the routine \S which does not allow negative indexes. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

```
524   \S:  SIZE TOO LARGE
```

The size which was specified for the routine \S is too large for the routine to handle. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

```
525   \S:  SIZE NOT POSITIVE
```

The size which was specified for the routine \S is not positive. The routine requires that the size be positive. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

Miscellaneous Errors

526 `\S: ILLEGAL FUNCTION CODE`

The function code specified for the routine \S is not valid for the operation that is desired. Make the necessary corrections in the source code so that the routine is called with the correct values for the arguments.

527 `\S: ILLEGAL TRAP CODE`

The routine \S encountered a trap code that is illegal or is not supported by the FORTRAN RTL. If the trap code is not needed then make the necessary corrections to the source code so that the trap code is not placed on the task queue. If the reason code is required, make the necessary changes to the source code to handle the trap code.

528 `\S: ILLEGAL LEVEL OF TRAPPING`

An illegal level of trapping was detected in the routine \S. Make the necessary corrections to the source code to prevent the illegal trapping.

529 `\S: ILLEGAL CALL`

An illegal call was detected by the routine \S. Make the necessary corrections to the source code to prevent the illegal call.

530 `\S: TARGET VARIABLE FOR FUNCTION IS OF WRONG TYPE`

The target variable for the function \S is not the correct type for this function. Make the necessary changes to the source code so the target variable is the correct type for the function that is being called.

531 `\S: NEGATIVE BASE`

The routine \S found that the argument that supplies the base for routine is negative. Make the necessary changes to the source so the routine is called with the correct values.

```
532    \S:   ARGUMENT OF INCORRECT CLASS
```

The routine \S was called with one or more arguments that are not of the proper class (i.e., INTEGER*4, INTEGER*2, etc.). Make the necessary corrections to the source code so that the arguments passed are of the correct class.

```
533    \S:   INCORRECT NUMBER OF ARGUMENTS
```

The routine was called with the incorrect number of arguments. Make the necessary corrections to the source code so that the correct number of arguments are called.

```
534    \S:   NO TRAPS
```

The program received a trap but does not have any traps enabled. Enable the desired trap so the RTL can process the desired trap.

```
535    \S:   NO ALTERNATE RETURN IN ARGUMENT LIST
```

An attempt was made to execute a RETURN statement with an alternate return specifier (e.g., RETURN 1). Correct the source code when the routine was called so that it includes the proper number of alternate return labels.

```
536    \S:   EXCEED RTL SCRATCH PAD,TEST/TRACE WITH I/O
```

The routine \S exhausted the RTL scratch pad that is available for use. Correct the source code. Normally, the RTL scratch pad can only be exhausted by the illegal nesting of FORTRAN I/O statements. Examine the source code for the illegal nesting of FORTRAN I/O and other routines that use the RTL scratch pad and return the space used when exiting the routine.

Special Error Messages

```
_ _U      -  'INSUFFICIENT MEMORY'
```

Not enough memory to get RTL stack space, load program with more memory.

```
_ _U      -  'TASK REQUIRES FEP BE PRESENT'
```

Program requires the FORTRAN ENHANCEMENT PACKAGE; the system options do not have the writable control store (WCS) option set.

```
.RTLST - 'ABORT:  INSUFFICIENT RTL SPACE'
```

There is not enough RTL stack space remaining to initialize for I/O. This can be caused by illegally nested I/O and must be remedied by altering the source code to decrease the nesting level. The task is terminated with an end of task code of 255.

```
_IOERMS (.IOMES) - 'ERROR EXISTS, CANNOT GIVE DETAILS DUE TO LACK OF
                    RTL SPACE'
```

There is not enough RTL stack space remaining to build an error message buffer. This can be caused by illegally nested I/O in which an error is encountered and must be remedied by altering the source code to decrease the nesting level.

```
_IOERMS (.IOMES) - 'ERROR \D (\A) \S : NO DETAILS, TEXT CANNOT BE RE
                   FROM F7RTLxx.ERR/S'
```

Error \D was detected, however, the error text file cannot be assigned/read. The error file should exist on the system volume on the system account. The RTL assigns the error file dynamically to a free lu. The error file may not be preassigned.

```
.XPA     -   'XPA - INSUFFICIENT USER MEMORY'
```

Not enough memory for work space; load program with more memory.

```
.XPA     -   'XPA - INSUFFICIENT SYSTEM SPACE FOR TIMER TRAP'
```

The system space must be increased by operator.

```
.XPA     -   'XPA - ALLOCATE ASSIGN ERROR <type> : FILE : file.XPA/P'
```

MAX lu, the output lu for XPA data, cannot be allocated/assigned.

```
XPA_SET -   'XPA_SET - INCORRECT ARGUMENT TYPE'
```

The arguments passed to XPA_SET are incorrect.

```
.CRA/.CRAXPA - 'CRA[XPA] - I/O ERROR xxxx WRITING CRA FILE'
```

An I/O error occurred while writing the call recording analysis (CRA) file. xxxx is the SVC1 status.

```
.CRA/.CRAXPA - 'CRA[XPA] - INSUFFICIENT MEMORY FOR TABLES'
```

Not enough memory for table space; load program with more memory.

```
.CRA/.CRAXPA - 'CRA[XPA] - ADDRESS MAP NOT FOUND ON MAP FILE'
```

ADDRESS option was not used when the map was requested by the Link MAP command. The task must be relinked and an address map generated.

```
.CRA/.CRAXPA - 'CRA [XPA] - voln:filename.MAP DOES NOT EXIST ON
                /P, /G OR /S'
```

The MAP file could not be located. It must exist on the same volume as the task and be on either the private, group, or system account.

```
XPATAB - 'MAX RANGE MUST BE >= MIN RANGE'
```

Range with which the analysis is to be performed must be of the form *range1-range2* where *range2 >= range1*.

```
XPATAB - 'NOT ENOUGH BUCKETS, INCREASE STEP SIZE OR PROGRAM
          LOAD SIZE'
```

Work size is insufficient for the requested analysis. Either increase the STEP size or reload XPATAB with a larger load size.

```
XPATAB - 'XPA FILE INCOMPLETE - PARTIAL XPATAB PRODUCED'
```

The program being analyzed did not run to completion which resulted in a partial XPATAB file.

Nonzero End-of-Task Codes

240 Insufficient user memory for XPA/CRA. Reload program with more work space.

241 XPA - Insufficient system space to store timer trap. Have operator increase system space.

CRA - Address map not found on MAP file or MAP file not found.

242 I/O error occurred writing CRA file. Disc is probably bad.

251 Error on release storage in a real-time program. The extra space required to save task environment was not released correctly.

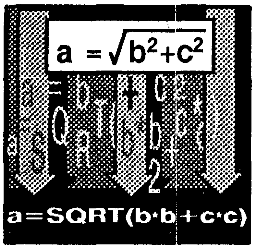
252 Task queue service, trap on empty queue.

253 Illegal reason code or bad item on task queue.

254 Insufficient space to save environment. Real-time task is incorrectly established.

255 Insufficient RTL space or the task is cancelled by the operator's command 'CANCEL'.

1000 Overflow on task queue.



A

RTL Subprograms

In this appendix

We introduce you to the routines which supplement the FORTRAN VII RTL. These routines are not directly callable from FORTRAN code.

Topics include:

- Program initiation and termination routines
- Formatted input/output (I/O) routines
- Unformatted, namelist and list-directed I/O routines
- Alternate returns for subprograms

Introduction

The descriptions in this appendix are supplied for information only. None of these routines are directly callable from FORTRAN code. The calling sequence for these routines is generally outside the scope of the standard FORTRAN subprogram calling sequence. The intrinsic routines, however, can be called from user-written assembly code which provides the correct standard interface.

All RTL routines follow these conventions:

- all modified registers are saved and restored except where they are used to return results,
- general-purpose register 1 is never corrupted, and
- code is PURE.

The FORTRAN VII compilers generate code to call the RTL routines listed in this appendix. FORTRAN VII R05-05 and earlier generate the code as listed. FORTRAN VII R06 generates entry points enclosed in parentheses.

Program Initiation and Termination Routines

Initiation and termination routines provided by the RTL are as follows:

- .U (_U) - This routine is called at the beginning of every FORTRAN main program. It reserves 1.5kB of memory for the RTL scratchpad. GPR1 points to the top of this reserved area. .U obtains another 256 bytes for the static communication area. The address of this area is placed at absolute location X'20' in the user-dedicated location (UDL). Each logical unit (lu) is checked to see if it is assigned to a printing device. If so, the bit in the static communication area corresponding to that lu is set to indicate that carriage control conversions are to be performed. The addresses of the top and bottom of the scratchpad area, the operating system revision and level, and the processor type are also placed in the static communication area. .U clears the single precision floating point register (SPFPR) and/or double precision floating point register (DPFPR), depending on which task options are chosen.
- .V (_V) - This routine terminates a program, releasing the RTL scratchpad and optionally closing all logical units. It is a result of an END statement. It is also called by EXIT, EXITRE, and STOP.

- .STOP (_STOP) - This program is called from the code generated for the STOP statement. STOP can have an unsigned integer or a character string as an argument.

  Examples:

  ```
  STOP
  STOP 7
  STOP MESSAGE
  ```

  .STOP logs the message specified in the STOP statement and terminates execution of the task.

- .PAUSE (_PAUSE) - This program is called from the code generated for the PAUSE statement. PAUSE can have an unsigned integer or a character string as an argument. .PAUSE logs the message specified in the PAUSE statement, then pauses the task.

Formatted I/O Routines

- Initialization Routines: (Pretranslated FORMAT Statements)

  ```
  .WXSC  (_RWXS)  -  WRITE EXTERNAL SEQUENTIAL
  .WXDC  (_RWXD)  -  WRITE EXTERNAL DIRECT
  .WNSC  (_RWIFS) -  WRITE INTERNAL
  .PRNTC (_APTXS) -  PRINT
  .TYPEC (_APTXS) -  TYPE
  .ENCDC (_EDIBS) -  ENCODE

  .RXSC  (_RWXS)  -  READ EXTERNAL SEQUENTIAL
  .RXDC  (_RWXD)  -  READ EXTERNAL DIRECT
  .RNSC  (_RWIFS) -  READ INTERNAL
  .ACPTC (_APTXS) -  ACCEPT
  .DECDC (_EDIBS) -  DECODE
  ```

- Data Transfer Routines:

  ```
  .RWDTF (_RWDRF) -  DATA TRANSFER
  ```

- Termination Routines:

.IOFNF	(_IOFNF)	-	TERMINATION

- FORMAT Translator:

.FORMT	(_TRFMF)	-	FORMAT TRANSLATOR

Unformatted, Namelist, List-Directed I/O Routines

- Initialization Routines:

.RDSU	(_RWSU)	-	UNFORMATTED READ SEQUENTIAL
.RDDU	(_RWDU)	-	UNFORMATTED READ DIRECT
.RDSN	(_RWSN)	-	NAMELIST READ SEQUENTIAL
.ACPTN	(_ACPTN)	-	NAMELIST ACCEPT
.RDSL	(_RWDL)	-	LIST-DIRECTED READ SEQUENTIAL
.ACPTL	(_ACPTL)	-	LIST-DIRECTED ACCEPT

.WRSU	(_RWSU)	-	UNFORMATTED WRITE SEQUENTIAL
.WRDU	(_RWDU)	-	UNFORMATTED WRITE DIRECT
.WRSN	(_RWSN)	-	NAMELIST WRITE SEQUENTIAL
.PRNTN	(_ACPTN)	-	NAMELIST PRINT
.TYPEN	(_ACPTN)	-	NAMELIST TYPE
.WRSL	(_RWDL)	-	LIST-DIRECTED WRITE SEQUENTIAL
.PRNTL	(_ACPTL)	-	LIST-DIRECTED PRINT
.TYPEL	(_ACPTL)	-	LIST-DIRECTED TYPE

- Data Transfer Routines:

.RWDTU	(_RWDTU)	-	UNFORMATTED READ/WRITE DATA
.RWDTL	(_RWDTL)	-	LIST-DIRECTED READ/WRITE DATA

- Termination Routines:

```
.IOFNU  (_IOFNU) -  UNFORMATTED TERMINATE
.IOFNL  (_IOFNL) -  LIST-DIRECTED TERMINATE
```

Auxiliary I/O Command Routines

```
.OPEN   (_OPEN)
.CLOSE  (_CLOSE)
.REW    (_REW)
.BACK   (_BACK)
.ENDF   (_ENDF)
```

Conversion Routines

.ATOP and .ATOD convert ASCII representation of numbers to SPFP and DPFP numbers. Conversely, to convert from floating point to ASCII, .FTOA, and .DTOA are used. .CTOI converts character data to integer and .ITOC converts integer to character.

Alternate Returns for Subroutines (.ARET)

The .ARET routine ensures that the alternate return count is greater than zero and less than or equal to *n*, where *n* is the total number of asterisks (*) and ampersands (&) in the dummy argument list. If *n* is not in the proper range, it returns immediately to the calling routine. If it is in the proper range, .ARET searches the argument list for the indicated alternate return address. In other words, if *k* is the alternate return count, .ARET searches for the *k*th argument word in the argument list. An argument word contains an alternate return address if bits 4 through 7 of the argument word are X'C'. If it finds such an alternate return address, it stores it in the normal return address save area and returns to the calling program. If it does not find an alternate return address, it issues an error message and returns to the calling program.

Debug Routines

- Test and Trace Routines:

```
.TEST  (_TEST)
.ATEST (_ATEST)
.CTEST (_CTEST)
.FTRCE (_VTRCE)
.VTRCE (_VTRCE)
.ATRCE (_ATRCE)
.STRCE (_STRCE)
.ASTRE (_ASTRE)
.ASTRG (_ASTRG)
.STRCD (_STRCD)
.FTRCD (_FTRCD)
```

- Argument Checking Routines:

  The routine .CHECK is called from all user callable RTL routines in the argument checking RTL. It can be turned off with a call to ICHECK. See Chapter 11 for more information on the ICHECK routine. When .CHECK is entered, GPR12 points to a block of data with which it checks the argument list; GPR13 contains the link address in the calling routine; GPR14 contains the pointer to the argument list; GPR15 contains the return address in the user program. Upon return, .CHECK sets GPR12 with:

  0 the list is acceptable,
  -1 the class or type of call is incorrect, or
  +1 the number, class, or type of arguments is incorrect.

The data block to which GPR12 points is organized as follows:

1.	NAME	6 bytes, ASCII
2.	Revision and update levels	2 bytes
3.	Class/type of call	1-byte (remark 1)
4.	Minimum number of arguments	1-byte
5.	Maximum number of arguments	1-byte (remark 2)
6.	Aliasing option	1-byte (remark 3)
7.	Acceptable classes of argument repeated as necessary	1-byte

8.	Argument class delimiter	X'CC' (remark 4)
9.	Acceptable type of argument repeated as necessary	1-byte
10.	Argument type delimiter	1-byte (remark 5)

Remark 1: The class and type values are as given in Chapter 5. In this byte, the most significant bit (MSB) is reset.

Remark 2: The maximum number of arguments is set to 0 if the number of arguments are arbitrary (e.g., MAXO).

Remark 3: The aliasing option is 1 if the subprogram can take either INTEGER*2 or INTEGER*4 arguments (e.g., IEOR), zero otherwise.

Remark 4: X'CC' indicates the end of all acceptable class types.

Remark 5: If this byte is X'FF', then all the remaining arguments have the same choices of type and class as the one that was previously processed. The value X'DD' for the byte indicates the end of all acceptable argument types.

See the section entitled "Passing Argument" in Chapter 5 for byte values.

- Error Response Routines:

  Run-time error conditions are handled by a system of four subprograms, .ERR, .ERRO, .ERRX, and .MES. These routines cannot be called from the user program. The error messages have the form:

```
ERROR  d (X):
r      :m
```

Where:

d is a number which identifies the kind of error.

x is the address in the user program where the routine was called.

r is the name of the RTL in which the error occurred.

m is an explanation of the error.

The explanatory texts are kept on a disk file, located on the system volume. When the error handler is activated, it attempts to assign this file to the highest lu. If the attempt fails because the file is not found or the highest lu is already assigned, the explanatory text *m* is not incorporated in the error message. The error number *d* can be used to identify the error.

Intrinsic Functions:

- The following mathematical functions are intrinsic functions which accept arguments in registers:

```
.CSQRT
.CDSQR
.CLOG
.CDLOG
.DTAN
.DACOS
.DNINT
.ALOG1
.ATAN
.TANH
.AINT
.CEXP
.CDEXP
.CABS
.CDABS
.DLOG1
.DATAN
.SIN
.ALOG
.ATAN2
.SQRT
.CSIN
.CDSIN
.CSIN
.DSIN
.DLOG
.DATN2
.COS
.ASIN
.SINH
.EXP
.CCOS
.CDCOS
.CCOS
.DCOS
.DASIN
.DSINH
.TAN
.ACOS
.COSH
.ANINT
```

- The intrinsic functions that accept arguments via an argument list are:

```
@ISHFT
@IAND
@NOT
@BCMPL
@BCLR
@CSIN
@CDSIN
@CCOS
@CDCOS
@CONJG
@DCONJ
@DTAN
@DACOS
@DCOSH
@DMIN1
@SNGL
@IDNIN
@DDIM
@TAN
@ATAN
@AMAX0
@AMAX1
@AINT
@ISIGN
@IDIM2
@COS
@CLOG
@CDLOG
@DSQRT
@DLOG1
@DATAN
@DTANH
@IDINT
@DBLE
@DMOD
@SQRT
@ALOG
@ATAN2
@AMIN0
@AMIN1
@INT2
@ANINT
@AMOD
@SIGN
@DIM
@ACOS
@IOR
@IEOR
@BTEST
@BSET
@CSQRT
@CDSQR
@CABS
@CDABS
@DEXP
@DLOG
@DATN2
@DPROD
@DINT
@DABS
@EXP
@ALOG1
@SINH
@MAX0
@MAX1
@IFIX
@FLOAT
@NINT
@IABS
@CMPLX
@TANH
@CEXP
@CDEXP
@AIMAG
@DIMAG
@DSIN
@DASIN
@DSINH
@DMAX1
@DFLOA
@DNINT
@DSIGN
@SIN
@ASIN
@COSH
@COSH
@MIN0
@MIN1
@INT
@MOD
@IDIM
@DCOS
@DREAL
@DCMPL
```

- Optimizable Intrinsic Functions:

This is a list of intrinsic functions that are candidates for code motion transformations.

- *ABS
- ACOS
- *AIMAG
- *AINT
- ALOG
- ALOG10
- *AMAX0
- *AMAX1
- *AMIN0
- *AMIN1
- *AMOD
- *ANINT
- ASIN
- ATAN
- ATAN2
- CABS
- CCOS
- CDABS
- CDCOS
- CDEXP
- CDLOG
- CDSIN
- CDSQRT
- CEXP
- CLOG
- *CMPLX
- *CONJG
- COS
- COSH
- CSIN
- CSQRT
- *DABS
- DACOS
- DASIN
- DATAN
- DATAN2
- *DBLE
- *DCMPLX
- *DCONJG
- DCOS
- DCOSH
- DDIM
- DEXP
- *DFLOAT
- *DIM
- DIMAG
- *DINT
- DLOG
- DLOG10
- *DMAX1
- *DMIN1
- *DMOD
- *DNINT
- *DPROD
- DREAL
- *DSIGN
- DSIN
- DSINH
- DSQRT
- DTAN
- DTANH
- EXP
- *FLOAT
- *IABS
- *IAND
- *ICHAR
- *IDIM
- *IDINT
- *IDNINT
- *IEOR
- *IFIX
- *IMAG
- *INT
- *INT1
- *INT2
- *IOR
- *ISHFT
- *ISIGN
- LGE
- LGT
- LLE
- LLT
- LOG
- LOG1
- LOG2
- LOG10
- *MAX
- *MAX0
- *MAX1
- *MIN
- *MIN0
- *MIN1
- *MOD
- *NINT
- *NOT
- *REAL
- *SIGN
- SIN
- SINH
- *SNGL
- SQRT
- TAN
- TANH

* Explicit invocation causes in-place expansion of the code for these routines.

The following intrinsic functions are not optimizable but are expanded in place.

BCLR
BCMPL
BSET
BTEST
CHAR
CTOI
GENSIG
GOAPU
GOCPU
IBCLR
IBITS
IBSET
ICBYTE
ILBYTE
INBYTE
INDEX
IRTCNT
ISBYTE
ISHFTC
ITOC
LEN
LOKOFF
LOKON
MVBITS
RSCHDL
TESET

RTL Constants

The following are entry points to routines containing RTL constants.

.CONLEN
.STACKSZ

The record length of a console (multi-terminal monitor (MTM) or OS/32) is set at 80 bytes. This default is in a global data area !CONLEN!. For consoles with record lengths other than the default, the user should modify .CONLEN accordingly.

Index

Document Comment Foı

In reference to...

Developing Programs With FORTRAN VII — A Guide — 48-010 F00 R04

We try to make our documentation easy to use, easy to understand, and fre from errors. We invite your comments and suggestions to assist us in impro our documentation to suit your needs.

Please send us comments, corrections, suggestions, etc. Use the SCR syst to report software documentation or software problems.

I think this manual...

	Strongly Agree	Agree	Disagree	Strong Disagr
is easy to read	□	□	□	□
is easily understood	□	□	□	□
is concise & to the point	□	□	□	□
covers the subject	□	□	□	□
has enough detail	□	□	□	□
is well organized	□	□	□	□
provides easy-to-locate information	□	□	□	□
is aesthetically pleasing	□	□	□	□
has clear illustrations	□	□	□	□
has enough illustrations	□	□	□	□
has meaningful examples	□	□	□	□
has a helpful index	□	□	□	□

My other comments...

Please make any additional specific comments. (Include chapter, page, tabl figure number.)

____

____

____

____

About myself...

Job Function:
□ Dev. Engineer
□ Technician
□ Service Eng.
□ Sys. Analyst
□ Administrator
□ Operator
□ Sys./App. P
□ Casual user
□ Other _____

What hardware system are you using?____________________

____

What revision level of system software are you using?____________

____

Name/Title: ________________________________

Company/Organization: __________________________

Address:___________________________________

May we contact you? □ Yes □ No

Telephone: __________________ Date: ____________

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 22 OCEANPORT, NJ

POSTAGE WILL BE PAID BY ADDRESSEE

Concurrent Computer Corporation
2 Crescent Place
Oceanport, NJ 07757

ATTN:
DOCUMENTATION DESIGN & DEVELOPMENT, M.S. 345

FOLD FOLD

STAPLE STAPLE

Document Comment Forn

In reference to...

______________________________ ______________

Manual Title *Number & Revision Leve*

We try to make our documentation easy to use, easy to understand, and free from errors. We invite your comments and suggestions to assist us in improvin our documentation to suit your needs.

Please send us comments, corrections, suggestions, etc. Use the SCR system to report software documentation or software problems.

I think this manual...

	Strongly Agree	Agree	Disagree	Strongly Disagree
is easy to read	□	□	□	□
is easily understood	□	□	□	□
is concise & to the point	□	□	□	□
covers the subject	□	□	□	□
has enough detail	□	□	□	□
is well organized	□	□	□	□
provides easy-to-locate information	□	□	□	□
is aesthetically pleasing	□	□	□	□
has clear illustrations	□	□	□	□
has enough illustrations	□	□	□	□
has meaningful examples	□	□	□	□
has a helpful index	□	□	□	□

My other comments...

Please make any additional specific comments. (Include chapter, page, table c figure number.)

____

____

____

____

About myself...

Job Function:

□ Dev. Engineer
□ Technician
□ Service Eng.

□ Sys. Analyst
□ Administrator
□ Operator

□ Sys./App. Pro
□ Casual user
□ Other ______

What hardware system are you using? ______________________________

____

What revision level of system software are you using? ______________

____

Name/Title: ______________________________________

Company/Organization: ____________________________

Address: ____

May we contact you? □ Yes □ No

Telephone: ____________________ Date: ____________

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 22 OCEANPORT, NJ

POSTAGE WILL BE PAID BY ADDRESSEE

Concurrent Computer Corporation
2 Crescent Place
Oceanport, NJ 07757

ATTN:
DOCUMENTATION DESIGN & DEVELOPMENT, M.S. 345

FOLD FOLD

STAPLE STAPLE