IBM

VS FORTRAN Version 2

SC26-4223-2

Interactive Debug Guide and Reference

Release 3

VS FORTRAN Version 2

SC26-4223-2

Interactive Debug Guide and Reference

Release 3

| **Third Edition (March 1988)**

| This edition replaces and makes obsolete the previous edition, SC26-4223-1.

| This edition applies to Release 3 of VS FORTRAN Version 2, Program Numbers 5668-805 and 5668-806, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Changes" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent publication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the | operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's program may be used. Any functionally equivalent program may be used instead.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, | comments may be addressed to IBM Corporation, Programming Publishing, P. O. Box 49023, San Jose, | California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

This book describes how to use VS FORTRAN Version 2 Interactive Debug to monitor and test the running of VS FORTRAN programs. This book can be used easily and effectively by scientists, engineers, other professionals, and students—all those who use VS FORTRAN for engineering and scientific problem solving.

How This Book Is Organized

- **"Part One. User's Guide,"** discusses how to use Interactive Debug.
  - **Chapter 1, "Introduction,"** discusses the features, requirements and restrictions of Interactive Debug.
  - **Chapter 2, "Debugging in Full Screen Mode,"** discusses how to invoke and use Interactive Debug in full screen mode.
  - **Chapter 3, "Debugging in Line Mode,"** discusses how to invoke and use Interactive Debug in line mode.
  - **Chapter 4, "Debugging in Batch Mode,"** discusses how to invoke and use Interactive Debug in batch mode.
  - **Chapter 5, "Using Interactive Debug Files,"** discusses how to use the Interactive Debug files AFFOUT, AFFIN, AFFPRINT, AFFON, AFFLST, and AFFPIF.
  - **Chapter 6, "Debugging Tasks,"** discusses how to perform common debugging tasks
  - **Chapter 7, "Special Considerations When Using Interactive Debug,"** describes special considerations in using Interactive Debug.
  - **Chapter 8, "A Sample Debugging Session,"** is a tutorial shows some of the basic features of Interactive Debug.
- **"Part Two. Command Reference,"** describes all Interactive Debug commands.
- **Appendixes**
  - **Appendix A, "Interactive Debug Messages,"** contains a list of Interactive Debug messages.
  - **Appendix B, "ISPF Dialog Variable Names Defined by Interactive Debug,"** lists ISPF dialog variable names which are defined by Interactive Debug.

How to Use This Book

- Read Part One to learn how to use Interactive Debug. The chapters should be read in order of appearance.
- After you are familiar with how to use Interactive Debug, use Part Two and the Appendixes as reference material. Part Two is alphabetically organized, with running headers on each page, to make commands easy to find.

Syntax Notation

For the syntax notation of all Interactive Debug commands, refer to page 100.

Summary of the VS FORTRAN Version 2 Publications

The following table lists the VS FORTRAN Version 2 publications and the tasks they support.

Task	VS FORTRAN Version 2 Publications	Order Numbers
Evaluation and Planning	General Information	GC26-4219
	Licensed Program Specifications	GC26-4225
Installation and Customization	Installation and Customization for VM	SC26-4339
	Installation and Customization for MVS	SC26-4340
Application Programming	Language and Library Reference	SC26-4221
	Programming Guide	SC26-4222
	Interactive Debug Guide and Reference	SC26-4223
	Reference Summary	SX26-3751
Diagnosis	Diagnosis Guide	LY27-9516

Summary of Changes

Release 3, March 1988

Major Changes to the Product

- A new main debugging panel in full screen mode that allows you to:
  - Display AUTOLIST data in a monitor window, in addition to displaying your source listing in a source window and your interaction with Interactive Debug in a log window.
  - Change the size and configuration of the windows with the enhanced WINDOW command and the new commands SIZE and ZOOM.
- Automatic refresh of the main debugging panel by Interactive Debug after a specified number of lines of output have been written to the log.
- Vector tuning assistance that allows you to:
  - Gather vector length and stride information with the new LISTVEC and VECSTAT commands.
  - Gather DO loop sampling and timing information with the enhanced LISTSAMP and LISTTIME commands.
  - Get on-line help for vector report messages with the enhanced HELP command.
  - View the source order vector report in full screen mode.
- Double-byte data support. After you specify the new DBCS YES command, double-byte data can be:
  - Displayed in the source listing (full screen mode only).
  - Contained in variable names and array names.
  - Contained in character literals, both entered and printed.
  - Used in terminal input and output to VS FORTRAN Version 2 programs.
- The addition of the new full screen commands RESTORE and RETRIEVE.
- Recognition of the restart file (AFFIN) and log file (AFFOUT) in line mode debugging.
- Enhancements to the include file (AFFON):
  - Defaults can be defined for listed and unlisted program units.
  - A program information file can be defined for a program unit. This file is needed by Interactive Debug to gather vector tuning information.
  - Debugging hooks can be placed only at DO loops, to improve DO loop timing accuracy.
  - Sequence numbers in columns 73 through 80 on 80 column records are ignored.

- Enhancement to the LISTTIME command to show average times.
- Enhancements to the format of the output produced by the ANNOTATE command.
- Addition of the default ddnames AFFLST and AFFPIF for the listing filename and program information filename respectively, for use in MVS batch mode debugging.
- New IAD EXEC which allows you to invoke Interactive Debug in full screen mode using ISPF without PDF.

Major Changes to This Manual

- Documentation of the major product enhancements has been added.
- Information on the Interactive Debug files AFFOUT, AFFIN, AFFPRINT, and AFFON is now found in Chapter 5, "Using Interactive Debug Files" on page 39.
- Information on the commands UP, DOWN, LEFT, and RIGHT has been added.
- Appendix B, Interactive Debug Command Summary, has been deleted. (You can find the information in the *VS FORTRAN Version 2 Reference Summary*.)
- A new Appendix B, ISPF Dialog Variable Names Defined by Interactive Debug, has been added.

Release 2, June 1987

Major Changes to the Product

- Support for 31-character symbolic names, including the underscore (_) character.
- Addition of a program sampling capability, to assist in identifying areas of a program that use the most CPU time.
- Addition of four new commands:
  1. ANNOTATE, to support program sampling.
  2. LISTSAMP, to support program sampling.
  3. LISTINGS, to display the listings panel when using ISPF Version 2.
  4. RECONNECT, to allow a CLOSED unit to be reset to its original (preconnected) state so that it can be implicitly opened again.
- Capability of providing substring notation for character variables.
- Capability of referencing array subscripts outside the range of the declared dimensions.
- Reduced dependency on TSO, including removal of the TSOLIB requirement in CMS batch mode debugging.
- Capability to debug programs invoked in TSO LOADGO.
- Support for debugging CMS MODULE files in ISPF.

Major Changes to This Manual

Documentation of the major product enhancements has been added.

Contents

Figures

Part One. User's Guide

Chapter 1. Introduction

VS FORTRAN Version 2 Interactive Debug is a flexible and efficient tool that assists you in monitoring the running of VS FORTRAN Version 1 and VS FORTRAN Version 2 programs. For convenience, "Interactive Debug" is used to refer to VS FORTRAN Version 2 Interactive Debug, and "VS FORTRAN" is used to refer to both VS FORTRAN Version 1 and VS FORTRAN Version 2.

Interactive Debug allows you to:

- Start, suspend, and continue program execution
- Examine, change, and display values of variables
- Gather and display program performance information
- Trace program transfers
- Control the action taken for run-time errors
- Save output in a file

You can use Interactive Debug in full screen mode, line mode, or batch mode. (Please refer to the separate chapters for each of the modes.)

Features

31-character names: You can use symbolic names up to 31 characters in length, including the underscore (_) character, for both local and global names.

On-line help information: By using the HELP command, you can get on-line help information on all Interactive Debug commands and tasks, as well as for vector report messages in the optional vector report source listing. Help is available in Time Sharing Option (TSO), Conversational Monitor System (CMS), and ISPF (Interactive System Product Facility).

Windows: In full screen mode, Interactive Debug allows you to view your source listing in a source window, view your interaction with Interactive Debug in a log window, and view variable and array information in a monitor window.

Program animation: In full screen mode, you can highlight the command currently running in the source window and control the pace of execution.

Split-screen display: Using ISPF with the Program Development Facility (PDF), you can split your screen to perform debugging on one section of the screen while editing your program on the other section.

Ability to issue AT, DESCRIBE, LIST, and OFF with PF keys: In full screen mode, you can assign the AT, DESCRIBE, LIST, and OFF commands to a PF key, and then perform many tasks by "pointing" with the cursor, instead of typing identifiers on the command line. This allows you to use Interactive Debug without knowing complex syntax or remembering command names.

Selective debugging of program units: An optional run-time control file (AFFON) lets you specify which program units will be debugged, and which statements in those program units will have debugging hooks. Those not selected for debugging will run at normal speed.

Manipulation of external files while debugging: Commands that are similar to VS FORTRAN Version 2 I/O statements (for example, ENDFILE, BACKSPACE, CLOSE, and REWIND) allow you to manipulate external sequential files. Also, while remaining in debug mode, you can browse or edit external sequential files used by the program being debugged.

Ability to issue system commands while debugging: Without terminating your debug session, you can issue commands at the system level.

On MVS/XA, ability to debug programs in either addressing mode: Programs that run in 31-bit addressing mode and reside either above or below the 16-megabyte line can be debugged, with restrictions (see page 6).

Debugging optimized and/or vectorized code: Programs compiled with optimization and/or vectorization can be debugged, with restrictions (see page 6).

Debugging reentrant code: Programs that are compiled with the RENT option and run in reentrant mode can be debugged, with restrictions (see appropriate bullet under "Compiler Requirements" on page 5).

Using sequence numbers instead of ISNs for source statements: Programs compiled with the SDUMP (SEQ) option will generate the SDUMP statement table using the sequence numbers you supply in columns 73 through 80. This makes it possible for you to debug programs using those numbers instead of internal statement numbers (ISNs).

Program sampling: This capability enables you to analyze an application program and identify areas that take the most CPU time to run. This information can assist you in improving your program's performance.

Saving output: Interactive Debug allows you to place the output from some of the commands in a print data set for later examination.

Set up expected input: A log of the debugging session is placed in a file (AFFOUT). This file can subsequently be used as input to Interactive Debug to re-create a previous debugging session.

Vector tuning assistance: Interactive Debug assists in the analysis and tuning of vectorized programs by providing the capability of:

- Gathering vector length and stride information at run time
- Summarizing program sampling counts by DO loop
- Timing DO loops
- Providing the capability of displaying a vector report source listing in the source window (full screen mode only)
- Providing help on the vector report messages in the vector report source listing

Double-byte data support: Interactive Debug enables you to specify double-byte data in Interactive Debug commands that contain symbolic names or character data as parameters. a source listing containing double-byte data.

Requirements

System Requirements

Interactive Debug can be used in:

- MVS/System Product Version 1 (5740-XYN or 5740-XYS) —all current releases, with TSO/E (5665-285).
- MVS/XA: MVS/System Product Version 2 (5665-291 or 5740-XC6) —all current releases. and MVS/XA DFP Version 2 (5665-XA2)—all current releases, with TSO/E (5665-285).
- VM/System Product (5664-167)—Release 4 or later, with or without VM/SP HPO (5664-173) —Release 4 or later.

  Use of the double-byte character set requires VM/System Product Release 5.
- VM/XA System Facility (5664-169)—Release 2.
- VM/XA System Product (5664-308)—Release 1, with bimodal CMS.

  VS FORTRAN Version 2 Release 3 will support VM/XA System Product Release 1 with bimodal CMS concurrent with the availability of ISPF support.

Using Interactive Debug in full screen mode requires:

- In MVS, ISPF Version 2 (5665-319), with or without ISPF/PDF Version 2 for MVS (5665-317).
- In VM, ISPF Version 2 for VM (5664-282), with or without ISPF/PDF Version 2 for VM (5664-285).

ISPF/PDF is required in order to use the following capabilities:

PDF browse and edit facilities in split-screen mode
Automatic browse of the debug print file and log at session end
Start of debugging using Interactive Debug's foreground invocation panel

VS FORTRAN Version 2 Release 2 will run in the VM/SP operating system only with the OS simulation facility. It cannot run with the DOS simulation capability of VM/CMS active.

TSO/E is required on MVS, or if TSO commands are issued in batch mode.

Compiler Requirements

VS FORTRAN programs to be debugged with Interactive Debug must have been compiled with:

- VS FORTRAN Version 1 Release 4 or later, or with any release of VS FORTRAN Version 2
- The SDUMP compiler option
- The TEST compiler option, if the program is reentrant and will reside in a shared area (DCSS or LPA). Reentrant VS FORTRAN Version 2 programs can be debugged in user storage with or without the TEST option.

Vector tuning information can be gathered only for those programs compiled with the IVA suboption of the VECTOR option of VS FORTRAN Version 2 Release 3.

Library Requirements

Programs compiled with releases subsequent to Version 1 Release 4 (including Version 2 releases) require link editing with the corresponding library release or a later library. For example, programs compiled on the Release 4.1 compiler would require link editing with the Release 4.1 library or a later library.

Using Interactive Debug in batch mode requires the VS FORTRAN Version 2 library.

Sampling requires the VS FORTRAN Version 2 Release 2 or later library.

Storage Requirements

Interactive Debug requires about 400K bytes of storage to begin execution (plus the storage required to load the program that will be debugged). Interactive Debug also acquires additional dynamic storage during execution. The amount varies according to the nature of the program being debugged, and the type and quantity of debugging commands issued.

Restrictions

Vectorized and Optimized Code: If a program unit is compiled with optimization and/or vectorization, Interactive Debug will issue a warning message indicating that the results of some debugging commands are unpredictable.

Overlays: Interactive Debug cannot debug programs that use overlays.

Multitasking Facility: If you are using the Multitasking Facility (MTF), only the main task can be debugged.

Migration

AFFON File: AFFON files of debugging jobs run with VS FORTRAN Interactive Debug Version 2 Release 2 or earlier must be modified to run on Release 3. See page 46 for more information.

AFFOUT File: AFFOUT files must be defined to Interactive Debug with a RECFM of FB and LRECL of 80. See page 39 for more information.

MOVECURS Command: Because there are now three windows in the main debugging panel, MOVECURS works differently from last release. In particular, if you specify a command on the command line, and then press a PF key assigned to MOVECURS in order to move the cursor to the affected window, the PF key must be set to

```
MOVECURS;
```

and not

```
MOVECURS
```

as in the previous release. See pages 163, 168, and 183 for more information.

Chapter 2. Debugging in Full Screen Mode

When Interactive Debug runs with ISPF (Interactive System Product Facility) in either CMS or TSO, it is running in *full screen mode* because the entire screen is available.

Full screen mode debugging allows you to view the source listing in a source window, commands and output in a log window, and variable and array values in a monitor window.

Invoking Interactive Debug in CMS

Using ISPF/PDF

1. Invoke ISPF/PDF. The ISPF/PDF Primary Option Panel will be displayed.
2. Select option 4, FOREGROUND. The Foreground Selection Panel will be displayed.
3. Select the Interactive Debug option (usually option 11).

   The foreground Interactive Debug panel will be displayed. See Figure 1.

   Note: To proceed directly to this panel, type `4.11` on the command line of the Primary Option Panel.

```
-------     FOREGROUND VS FORTRAN VERSION 2.3.0 INTERACTIVE DEBUG    --------
COMMAND ===>

ISPF LIBRARY:
   PROJECT ===>
   LIBRARY ===>
   TYPE    ===>
   MEMBER  ===>              (Blank for member selection list)

CMS FILE:
   FILE ID    ===>                      (TEXT or MODULE file or LOADLIB  )
   MEMBER     ===>           (If member of a LOADLIB)
   IF NOT LINKED, SPECIFY:
   OWNER'S ID ===>           DEVICE ADDR. ===>      LINK ACCESS MODE ===>

READ PASSWORD ===>

EXECUTION TIME OPTIONS:
   DEBUG  ===>                (Enter DEBUG or NODEBUG)
   OTHER  ===>


S/SLIB TXTLIB:   (VSF2FORT, TSOLIB and CMSLIB already specified)
         ===>          ===>          ===>          ===>
```

Figure 1. Foreground Interactive Debug Panel for CMS

4. Fill in either the ISPF LIBRARY fields or the CMS FILE fields.

   If you are not already familiar with ISPF libraries, you should probably use the CMS FILE fields. If both groups of fields are filled in, the ISPF LIBRARY fields will be ignored. If you do not provide the file ID, the ISPF library member name will be used.

Here are some additional guidelines:

- For a TEXT file, fill in the FILE ID line with the filename optionally followed by a filetype of TEXT. Leave the MEMBER line blank.
- For a MODULE file, fill in the FILE ID line with the filename followed by a filetype of MODULE. Leave the MEMBER line blank.
- For a LOAD LIBRARY member, fill in the FILE ID line with the filename and do not specify a filetype. Fill in the MEMBER line with the library member name of your program.
- For the files described in Chapter 5, "Using Interactive Debug Files" on page 39, fill in the FILEID or MEMBER field with the appropriate name:

  fname INCLUDE * - for AFFON
  fname LIST A - for AFFPRINT
  fname LISTING * - default listing name
  fname LOG A - for AFFOUT
  fname PIF * - default program information file name
  fname RESTART * - for AFFIN

5. Fill in the READ PASSWORD field to access the disk if you don't already have access.
6. Fill in the DEBUG field under EXECUTION TIME OPTIONS with DEBUG.

   If you specify NODEBUG, Interactive Debug will not be invoked unless you include a special object module to override the default. For further information about overriding the default options, see "Specifying Default Run-Time Options" on page 84.
7. Optionally, specify additional run-time options in the OTHER field.

   You can continue the list of other options onto the second line if necessary. Separate options, and keywords within one option, with commas. Do not type in any extra blanks. For more information on run-time options, see *VS FORTRAN Version 2 Programming Guide*.
8. Optionally, specify up to four additional TXTLIBS on the last line of the panel.

   These may be accessed in addition to VSF2FORT whenever CMS determines that it needs to search a TXTLIB (for example, when loading a TEXT file and resolving all the external references).

   If you want your program to operate in link mode, specify VSF2LINK as an additional TXTLIB. (For more information on link mode, see *VS FORTRAN Version 2 Programming Guide*.)

Loading Multiple Text Files

When you use the ISPF panel to start debugging a TEXT file, you can specify only a single TEXT file (program name). Should you be debugging a program that calls other programs, the calls will be automatically resolved for you — provided the called programs are either in a TXTLIB (specified on the panel as a SYSLIB TXTLIB) or referenced in the calling program by their file names. (The file names must be the same as the SUBROUTINE or FUNCTION names.) If your called programs are not in a SYSLIB TXTLIB or referenced by their file names, you will need to follow a different procedure. Here are three alternative methods:

Loading from a LOADLIB

Before invoking ISPF, place your main program and all its called programs in a LOADLIB. You can use the CMS LKED command to build a LOADLIB.

The input to LKED is a file containing any combination of text files and LKED control statements. (LKED control statements are the same as those accepted by the MVS linkage editor.) For example, you could create a file of control statements:

```
INCLUDE MYPROG
INCLUDE SUB1
INCLUDE MYMATH
ENTRY MAIN
NAME MYPROG(R)
```

Assume this file is named "MYFILE TEXT." Each control statement must be preceded by at least one blank.

Now, enter the FILEDEF and LKED commands for your LOADLIB:

```
FILEDEF SYSLIB DISK VSF2FORT TXTLIB *
LKED MYFILE (MAP
```

This produces a file named "MYFILE LOADLIB," which contains a link-edited program named MYPROG. It also produces a listing file named "MYFILE LKEDIT," which contains a map of your link-edited program.

Specify the LOADLIB name and member name in the FOREGROUND Interactive Debug panel.

Loading a concatenated TEXT file

Alternatively, you can concatenate all the TEXT files into a single TEXT file. Use the CMS COPYFILE command with the APPEND option, or an editor such as XEDIT or the ISPF/PDF editor, to create a composite TEXT file. Then, specify this composite TEXT file in the FOREGROUND Interactive Debug Panel.

Loading a module file

If you have generated a MODULE file containing your VS FORTRAN program, you can specify the module filename and filetype on the ISPF panel. However, the module must be generated with a higher origin than the default in order to leave room for the invocation program. (ORIGIN 22000 will probably be high enough.) If the module origin is too low, the following message will appear:

```
LOAD ADDRESS TOO LOW
```

Using ISPF without PDF

The product tape provides the IAD EXEC which allows you to invoke Interactive Debug using ISPF without PDF. To use it, simply follow these steps:

1. Link to the product minidisk(s) containing ISPF and VS FORTRAN Version 2.
2. Type the following, where name is the program name (filetype of TEXT):

   ```
   IAD name
   ```

   To see the full syntax of the IAD EXEC, simply type

   ```
   IAD
   ```

The following should then appear:

```
Proper syntax is:

     IAD name <type> <( <options> <MEMBER=member name>>

where:
  "name" is the name of the file containing the FORTRAN
     program.

  "type" is the type of the file containing the FORTRAN
     program: TEXT, MODULE, LOADLIB or blank.  If blank
     is specified, the default type is TEXT.
    (Note: If type is LOADLIB a member must be specified.)

  "options" are the VS FORTRAN execution_time options.
    (Note: DEBUG is defaulted.)

  "member name" is the name of the member if LOADLIB was
     specified and blank otherwise.
```

3. After pressing ENTER, the Interactive Debug main debugging panel (page 14) should appear.

The IAD EXEC issues FILEDEF statements for AFFON, AFFIN, AFFOUT, and AFFPRINT. It defines Interactive Debug files with the following names:

name INCLUDE *	- for AFFON
name LIST A	- for AFFPRINT
name LISTING *	- default listing name
name LOG A	- for AFFOUT
name PIF *	- default program information file name
name RESTART *	- for AFFIN

For more information on these Interactive Debug files, see Chapter 5, "Using Interactive Debug Files" on page 39.

Invoking Interactive Debug in TSO

Using ISPF/PDF

1. Invoke ISPF/PDF. The ISPF/PDF Primary Option Panel will be displayed.
2. Select option 4, FOREGROUND. The Foreground Selection Panel will be displayed.
3. Select the Interactive Debug option (usually option 11).

   The foreground Interactive Debug panel will be displayed. See Figure 2 on page 11.

```
---------- FOREGROUND VS FORTRAN VERSION 2.3.0 INTERACTIVE DEBUG -------
COMMAND ===>

ISPF LIBRARY:
   PROJECT ===>
   LIBRARY ===>
   TYPE    ===>
   MEMBER  ===>                (Blank for member selection list)

OTHER PARTITIONED DATA SET:
   DATASET NAME  ===>

FILE ID FOR DEBUG FILES
           ===>                          PASSWORD ===>

EXECUTION TIME OPTIONS:
   DEBUG   ===>                (Enter DEBUG or NODEBUG)
   OTHER   ===>
```

Figure 2. ISPF Foreground Interactive Debug Panel for TSO

4. Fill in either the ISPF LIBRARY fields or the OTHER PARTITIONED DATA SET fields.

   If both groups of fields are filled in, the ISPF LIBRARY fields will be ignored.

   If the OTHER PARTITIONED DATA SET line is filled in, you should specify both the data set and member names. If you do not specify the member name, you will see a panel with a list of member names. You then choose the member name you need. Specify the data set and member names as follows:

   ```
   'data.set.name(member)'
   ```

   Note: If you omit the quotation marks, ISPF will supply the data set prefix specified as the default in your TSO profile.

5. Optionally, use the FILE ID FOR DEBUG FILES line to specify the file ID of the following Interactive Debug files:

userid.fileid.INCLUDE	- for AFFON
userid.fileid.LOG	- for AFFOUT
userid.fileid.PRINT	- for AFFPRINT
userid.fileid.RESTART	- for AFFIN
userid.fileid.LIST	- default listing name
userid.fileid.PIF	- default program information file

   If you do not provide the file ID for any of the first four files, the ISPF library member name will be used. The last two files (userid.fileid.LIST or userid.fileid.PIF) always use the ISPF library member name.

6. Fill in the PASSWORD field to specify a password for any password-protected data sets.

7. Fill in the DEBUG field under EXECUTION TIME OPTIONS with DEBUG.

   If you do not specify DEBUG, Interactive Debug will not be invoked unless you include a special object module to override the default. (For further information about overriding the default options, see "Specifying Default Run-Time Options" on page 84.)

8. Optionally, specify additional run-time options in the OTHER field.

You can continue the list of other options onto the second line if necessary. Separate options, and keywords within one option, with commas. Do not type in any extra blanks. For more information on run-time options, see *VS FORTRAN Version 2 Programming Guide*.

Using ISPF without PDF

1. Define the necessary data sets for running ISPF and Interactive Debug.

   Modify your allocations for the ISPF data sets to include allocations for the CLIST library, the IAD panel library, message library and command table. Examples of lines which could be included in a CLIST are shown in Figure 3.

   CAUTION: We advise you to use the procedures in this manual to establish your Interactive Debug libraries; they are the only ones supported by IAD. If you use another approach, such as the ISPF LIBDEF service, you may get unpredictable results.

```
/*********************************************************************/
/*                                                                   */
/* THIS IS AN EXAMPLE OF A CLIST FOR SETTING UP ISPF W/O PDF.  IT */
/* INCLUDES DEFINITIONS FOR THE DATA SETS REQUIRED BY VS FORTRAN  */
/* VERSION 2 INTERACTIVE DEBUG.  THE ISPF NAMES SHOWN IN THIS     */
/* EXAMPLE MAY BE DIFFERENT FROM THE ONES USED AT YOUR LOCATION.  */
/*                                                                   */
/*********************************************************************/

FREE  FI(ISPPLIB ISPMLIB ISPLLIB ISPTLIB ISPPROF)
ALLOC FI(ISPPROF)  DA('userid.ISPF.PROFILE') SHR REUSE
ALLOC FI(SYSPROC)  DA('VSF2.VSF2CLIB') SHR REUSE
ALLOC FI(ISPMLIB)  DA('VSF2.VSF2MLIB' 'ISP.V2R1M0.ISPMLIB') SHR REUSE
ALLOC FI(ISPPLIB)  DA('VSF2.VSF2PLIB' 'ISP.V2R1M0.ISPPLIB') SHR REUSE
ALLOC FI(ISPTLIB)  DA('VSF2.VSF2TLIB' 'ISP.V2R1M0.ISPTLIB') SHR REUSE
ALLOC FI(ISPLLIB)  DA('VSF2.VSF2LOAD' 'ISP.V2R1M0.ISPLOAD') SHR REUSE
ALLOC FI(FT05F001) DA(*)
ALLOC FI(FT06F001) DA(*)
```

Figure 3. Sample Modifications to the ISPF Invocation CLIST

2. Modify the ISPF Master Application Menu to include an option for Interactive Debug, if the menu does not already include this option. The menu may also have to be modified to call an application panel created by your installation (see step 3).

   Figure 4 on page 13 shows how you might modify the ISPF master application menu to provide an option for Interactive Debug (here, option 2) and invoke a prompt panel (here, called USERISP).

```
%----------------------  ISPF MASTER APPLICATION MENU  ----------------------
%OPTION  ===>_ZCMD                          %     +USERID   - &ZUSER
%   1 +SAMPLE1     - Sample application 1          +TIME     - &ZTIME
%   2 +VSF IAD     - VS FORTRAN Interactive Debug  +TERMINAL - &ZTERM  <-----(new
%   3 +.           - (Description for option 3)    +PF KEYS  - &ZKEYS       option)
%   4 +.           - (Description for option 4)
%   5 +.           - (Description for option 5)
%   X +EXIT        - Terminate ISPF using list/log defaults
%
+Enter%END+command to terminate ISPF.
%
)INIT
  .HELP    = ISP00005     /* Help for this master menu             */
  &ZPRIM   = YES          /* This is a primary option menu         */
)PROC
  &ZSEL = TRANS( TRUNC (&ZCMD,'.')
               1,'PANEL(ISP@PRIM)' /* Sample primary option menu  */
               2,'PANEL(USERISP)'  /* SAMPLE PANEL TO INVOKE IAD  */  <-----(new
           /*********************************************************/        prompt)
           /*                                                       */
           /* Add other applications here.                          */
           /*                                                       */
           /*********************************************************/
             ' ',' '
               X,'EXIT'
               *,'?' )
)END
```

Figure 4. Sample Modifications to ISPF Master Application Menu

3. Create an application panel that prompts for the name of your load module and then invokes a CLIST created by your installation (see step 4).

   You must create an application panel at your installation similar to the one shown in Figure 5. The sample prompts the user for the name of the text file, and then invokes a CLIST called USERDBG.

```
)ATTR DEFAULT(%+_)
     /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for      */
     /*  + TYPE(TEXT) INTENS(LOW)       information only            */
     /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)             */
  ! TYPE(INPUT) INTENS(LOW) PAD(' ')  /* input field padded with ' ' */
)BODY
%--------------------  SAMPLE VSF IAD PANEL  ---------------------------------
%COMMAND ===>_ZCMD                                                        +
%
+   ENTER THE NAME OF YOUR PROGRAM BELOW...
+  %===>!MEM
+
+   ENTER THE NAME OF YOUR LIBRARY BELOW...
+  %===>!LIB

+   ENTER THE LIST OF YOUR EXECUTION TIME OPTIONS BELOW...
+  %===>!FDEBUG
+
)PROC
 VER(&MEM,NB,NAME)
 VPUT (MEM,LIB,FDEBUG) PROFILE
 &ZSEL='CMD(%USERDBG)'
)END
```

Figure 5. Sample Application Panel (USERISP) to Prompt for the Text File Name

4. Create a CLIST to run the VS FORTRAN program and pass the run-time options.

   The sample CLIST in Figure 6 invokes AFFLOAD which in turn loads the VS FORTRAN Version 1 program with the DEBUG parameter. For this example, we have called the CLIST "USERDBG."

```
PROC 0
CONTROL NOLIST MAIN NOFLUSH NOMSG
/*    MEM    - MEMBER  NAME                                          */
/*    LIB    - LIBRARY NAME                                          */
/*    FDEBUG - EXECUTION TIME OPTIONS                                */
ISPEXEC VGET (MEM,LIB,FDEBUG,ZPREFIX)
SET &FAFFID = &MEM                          /* DEFAULT TO MEMBER NAME  */
SET &ZFAFFIN = &STR('&ZPREFIX..&FAFFID..RESTART')
SET &ZFAFFOU = &STR('&ZPREFIX..&FAFFID..LOG')
SET &ZFAFFON = &STR('&ZPREFIX..&FAFFID..INCLUDE')
SET &ZFAFFPR = &STR('&ZPREFIX..&FAFFID..PRINT')
FREE FI(AFFPRINT AFFIN AFFOUT AFFON AFFLOAD)
ALLOC FI(AFFIN) DA(&ZFAFFIN) SHR
IF &LASTCC ¬= 0 THEN ALLOC FI(AFFIN) DUMMY RECFM(F)
ALLOC FI(AFFON) DA(&ZFAFFON) SHR
IF &LASTCC ¬= 0 THEN ALLOC FI(AFFON) DUMMY RECFM(F)
DELETE &ZFAFFOU
ALLOC FI(AFFOUT) DA(&ZFAFFOU) MOD CATALOG SPACE (1) CYLINDERS
DELETE &ZFAFFPR
ALLOC FI(AFFPRINT) DA(&ZFAFFPR) MOD CATALOG SPACE(1) CYLINDERS
ALLOC FI(AFFLOAD) DA(&LIB) SHR
ISPEXEC SELECT PGM(AFFLINKF) PARM(&MEM/&FDEBUG) NEWAPPL(AFF) NEWPOOL
FREE FI(AFFPRINT AFFIN AFFOUT AFFON AFFLOAD)
```

Figure 6. Sample CLIST (USERDBG) to Invoke the Program without PDF

Using the Main Debugging Panel

When you enter Interactive Debug, the *main debugging panel* is displayed. Figure 7 on page 15 shows the main debugging panel for program SAMPLE found in Chapter 8, "A Sample Debugging Session" on page 93.

Note: You may initially be presented with the Interactive Debug Listings Panel (page 22) if listings are not defined for every debuggable program unit.

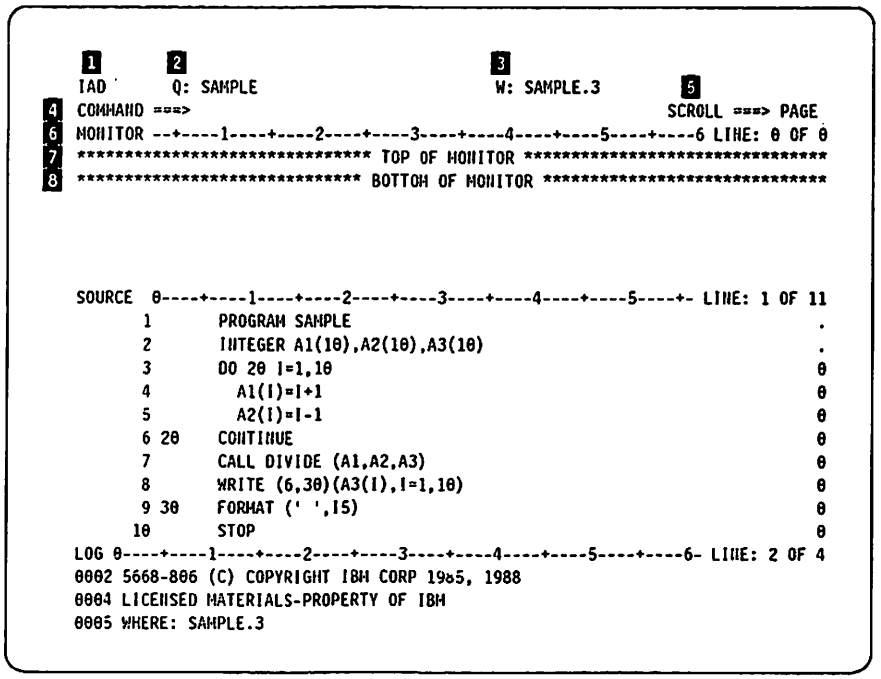

Figure 7. Sample Main Debugging Panel

The main debugging panel consists of a two header lines, a monitor window, a source window, and a log window. The boxed numbers in Figure 7 correspond to the discussion below.

The header contains:

1 **IAD** Indicates Interactive Debug status. It may be followed by a /R (read), /E (error), /W (write), or /F (finished).

2 **Q:** The name of the currently qualified program unit. Normally, the name is that of the program unit running. In the sample panel, the program unit is

SAMPLE

3 **W:** The statement in the VS FORTRAN program at which execution has been suspended. Also displayed is the name of the program unit where the statement is located. In the sample panel, this is statement 3 in program unit SAMPLE, or

SAMPLE.3

4 **COMMAND ===>** The command line. Interactive Debug, ISPF, and PDF commands may be entered here.

5 **SCROLL ===>** Indicates the default scrolling amount to be used by the UP, DOWN, LEFT, and RIGHT commands.

Each window contains:

6 Header line. This line contains:

- The window identifier (either MONITOR, SOURCE, or LOG)
- A character scale
- The current position (line number) of the contents of the window

7 Top-of-data marker

8 Bottom-of-data marker

The log and monitor window contain up to 1000 lines; there is no limit for the source window.

Each window is either:

- Closed (not displayed).
- Open (displayed with contents).
- Empty (displayed with no contents). An empty window shows only the top-of-data and bottom-of-data markers. The monitor window is initially empty.

Let's now examine each window more closely.

Monitor Window

```
MONITOR --+----1----+----2----+----3----+----4----+----5----+---- LINE: 1 OF 10
0001 SAMPLE.A1(1)     =          0
0002 SAMPLE.A1(2)     =          0
0003 SAMPLE.A1(3)     =          0
0004 SAMPLE.A1(4)     =          0
0005 SAMPLE.A1(5)     =          0
0006 SAMPLE.A1(6)     =          0
```

Figure 8. Sample Monitor Window

The monitor window consists of two parts: prefix area and output from the AUTOLIST command (page 111). Figure 8 shows the monitor window for program SAMPLE (page 93), after `autolist a1` has been entered on the command line.

The contents of the window are automatically refreshed each time a command is entered. If you were now to run the program SAMPLE statement by statement, using the STEP command, you would see the values change for array A1.

During the session, you can view up to 1000 lines of information in the monitor window.

Specifying AUTOLIST with no operands will terminate monitoring and remove all information from the monitor window. If the monitor window was originally open, it will become empty and will show the "TOP OF MONITOR/BOTTOM OF MONITOR" markers.

Source Window

```
SOURCE  0----+----1----+----2----+----3----+----4----+----5----+- LINE: 1 OF 11
       1       PROGRAM SAMPLE                                              .
       2       INTEGER A1(10),A2(10),A3(10)                                .
       3       DO 20 I=1,10                                                0
       4         A1(I)=I+1                                                 0
       5         A2(I)=I-1                                                 0
       6 20    CONTINUE                                                    0
```

Figure 9. Sample Source Window

The source window consists of three parts: prefix area, source listing area, and suffix area.

1. **Prefix area:** the eight leftmost columns of the source window, containing the statement numbers (ISNs or sequence numbers) that the compiler uses to identify each statement in the program. They are extracted from the source listing.

   The commands AT and OFF can be typed in the prefix area. They can be typed on more than one prefix line, but only one per line. AT and OFF commands that are specified in the prefix area are logged as if they had been entered on the command line.

   Whenever a displayed line in the source window has an AT breakpoint associated with it, the prefix is highlighted, as specified on the Interactive Debug color panel (page 118).

   If an AT or OFF command is typed incorrectly, the erroneous command is redisplayed in the prefix area with a different attribute (highlighted or different color if supported) and followed by a question mark. On the next interaction, if the user has not taken any action to correct the error, the prefix is restored to its previous contents.

2. **Source listing area:** contains the source lines of the currently qualified program unit extracted from the listing.

   If annotation (page 104) is set on, then the source listing area can also contain bar charts or vector messages.

3. **Suffix area:** contains the statement run-time count or sampling count. The suffix area can be removed by changing the appropriate field in the Interactive Debug Profile Panel (page 170).

The source window will be empty if:

- You specified "NOSOURCE" when you compiled the program.
- The source listing for the currently qualified program unit does not exist (you erased it).
- The source listing exists, but the disk containing the source listing is not accessed.
- You specified "NO" in the DISPLAY field of the Interactive Debug Listings Panel.

Log Window

```
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 15
0001 VS FORTRAN VERSION 2 RELEASE 3 INTERACTIVE DEBUG
0002 5668-806 (C) COPYRIGHT IBM CORP. 1985, 1988
0003 LICENSED MATERIALS-PROPERTY OF IBM
0004 WHERE: SAMPLE.3
0005 * list a1
0006 SAMPLE.A1(1)     =          0
0007 SAMPLE.A1(2)     =          0
0008 SAMPLE.A1(3)     =          0
0009 SAMPLE.A1(4)     =          0
0010 SAMPLE.A1(5)     =          0
0011 SAMPLE.A1(6)     =          0
0012 SAMPLE.A1(7)     =          0
0013 SAMPLE.A1(8)     =          0
0014 SAMPLE.A1(9)     =          0
0015 SAMPLE.A1(10)    =          0
```

Figure 10. Sample Log Window

The log window consists of two parts: prefix area and a session log, which is a record of recent interactions between you and Interactive Debug. Input lines are preceded by asterisks. ISPF commands and full screen commands are not logged. The sample log window in Figure 10 shows that the `list a1` command was entered.

During the session, the last 1000 log lines are retained in storage to be viewed by the user. If the output log file AFFOUT (page 39) exists, the entire log will be written to AFFOUT.

Changing the Way Your Windows Look

Commands used: DOWN (page 123), LEFT (page 140), MOVECURS (page 163), POSITION (page 168), RIGHT (page 182), SEARCH (page 183), SIZE (page 188), UP (page 199), WINDOW (page 206), ZOOM (page 208)

Here are some ways to manipulate the windows:

- To open a window, use the WINDOW OPEN command. You can open a window even if it's empty.
- To close a window, use the WINDOW CLOSE command.
- Various window configurations are possible. The initial configuration is shown in Figure 7 on page 15. To change the window configuration, use the WINDOW command with no operands.
- To change window sizes, use the SIZE command.
- To scroll up, down, left, or right within a window, use the UP, DOWN, LEFT, and RIGHT commands. If you have not changed the default program function key (PF key) settings, you can also use the PF7 and PF8 keys to scroll up and down, and PF10 and PF11 to scroll left and right.
- To change the top line of any window, use the POSITION command.

Entering Commands

In full screen mode, you can enter a command in the following ways:

- Type the command on the command line.
- Modify the command in the log window.
- Use a program function key (PF key).
- Use the RETRIEVE command.
- Use the command as a cursor-sensitive command (does not apply to all Interactive Debug commands).

Typing the Command on the Command Line

The simplest way to enter an Interactive Debug command is to type it on the command line.

If a command is too long to fit on the command line, enter as much of the command as you can, followed by the *continuation character* (-) to indicate that the command is not yet complete. Press ENTER. Then enter the remainder of the command.

The continuation character (-) must be the last character entered on the command line. You may enter up to 251 characters, including blanks, in one command. If a continuation segment requires leading blanks, type a quotation mark (") first, then the required leading blanks. Interactive Debug will remove the quotation mark and recognize the leading blanks.

While Interactive Debug is awaiting the continuation of a command, COMMAND will be replaced on the panel by MORE... and you will receive the message

```
AFFA019I CURRENT COMMAND IS INCOMPLETE, PENDING MORE INPUT
```

If you do not wish to continue the command, type END (or press a PF key that has been assigned to END) and the command will be ignored.

Modifying the Command in the Log Window

If you want to enter a command that has already been logged, you can move the cursor into the log window, modify the desired command (such as blanking out the leading asterisk), and press ENTER. This will cause the command to be copied to the command line. You may then modify the command further before pressing ENTER. This method allows you to avoid re-typing a long variable name.

If the output line contains double-byte characters, only the visible portion of the line will be copied to the command line. The integrity of the double-byte characters will be preserved.

Using PF Keys

Using program function keys (PF keys) to enter commands saves you time because you can simply press a PF key instead of typing in a long command.

To define PF keys, enter the ISPF command KEYS on the command line. ISPF then presents you with a panel that displays the current PF key definitions and allows you to change them. You can assign any Interactive Debug command or

a stack of commands to any PF key. To set a PF key to a stack of commands, like STEP and WHERE, type the following on the PF key definition line:

```
STEP; WHERE
```

When you have made all the changes you want, press the END key.

For information on setting the MOVECURS command to a PF key, see page 163.

Use the ISPF command PFSHOW to display your PF key settings across the bottom of the screen. Use PFSHOW OFF to hide the settings.

Using the RETRIEVE Command

Enter the command RETRIEVE (page 179) either on the command line or from a PF key to retrieve up to the twelve most recent commands to the command line. (Note that if you are using ISPF Version 2 Release 3 or later, RETRIEVE will behave differently. See page 179 for details.)

Using Cursor-Sensitive Commands

You can specify operands of a command by pointing to them with the cursor, instead of typing them on the command line. The commands that can be issued this way are called *cursor-sensitive* commands. The following are cursor-sensitive commands:

AT	POSITION
DESCRIBE	RIGHT
DOWN	SEARCH
HELP (for vector report messages)	SIZE
LEFT	UP
LIST	WINDOW CLOSE
OFF	ZOOM

A cursor-sensitive command affects the window in which the cursor is located. For example, if you want to scroll up four lines in the log window, type `UP 4` on the command line, position the cursor in the log window, and press ENTER.

If, however, the cursor is not in a window, the following window hierarchy applies:

- If the source window is open, the source window will be affected.
- If the source window is not open and the log window is open, the log window will be affected.
- If neither the source window or the log window are open, the monitor window will be affected.

Note: The rules above do not apply to the WINDOW CLOSE command: if WINDOW CLOSE is specified without a window name and the cursor is not in a window, an error message will be displayed.

The two cursor-sensitive commands AT and OFF can also be typed in the prefix area of the source window (see page 17).

Non-full screen cursor-sensitive commands are recorded in your session log as if the equivalent command had been typed on the command line. For example, if you type AT over ISN 12 in the source window while program unit SUB1 is displayed, the command `AT 12` is recorded in the log.

If the command line contains information when you press a PF key for a cursor-sensitive command, the information will be appended to that command. For example, if you enter 12 on the command line, and then press a PF key set to the AT command, AT 12 will be run and recorded in the log.

Interactive Debug allows you to type over multiple ISN or sequence number fields with AT or OFF commands before pressing ENTER.

Changing the Defaults for Your Debugging Session

In full screen mode, your Interactive Debug session can be customized by changing the default values for:

- The way your debugging session runs
- The window configuration
- The color of the various parts of the main debugging panel
- Listing information
- PF keys (see page 19)

Changing the Way Your Debugging Session Runs

Commands used: PROFILE (page 170)

Use the PROFILE command to display and change the current settings for various parameters that affect the way Interactive Debug runs. To see these parameters, enter the command PROFILE. A display panel will then appear on your screen. You can modify this display any time during a debugging session.

One of the parameters on the profile screen is "Output refresh value." This specifies the number of output lines after which Interactive Debug automatically refreshes the main debugging panel.

For each refresh, the following will occur:

- The "W:" field is updated.
- The values of variables are updated in the monitor window.
- The currently executing line in the program is highlighted in the source window.
- The last line of output is displayed in the log window.

Changing the Window Configuration

Commands used: SIZE (page 188), WINDOW (page 206)

Initially, the main debugging panel is divided into three windows (monitor, source, and log), each section spanning the full width of the screen. You can, however, change the way the windows are arranged, by using the WINDOW command. Using WINDOW allows you to choose among six predefined configurations, and save the one you find most useful. The SIZE command allows you to further customize the window configuration by making a window bigger or smaller.

You can save the customized configuration by doing the following:

1. Use the SIZE command until you are satisfied with the window configuration.
2. Enter WINDOW SAVE on the command line.

To restore the default window and size settings, enter WINDOW on the command line to display the Window Configuration Selection Panel. Then enter RESET on the command line.

Changing the Display Color

Commands used: COLOR (page 118)

You will probably find debugging more convenient if you highlight certain parts of the Interactive Debug main debugging panel, such as the current statement in the source window, and the statement identifiers at which breakpoints have been set. You can change the color, highlighting, or intensity of various fields on the panel, using the COLOR command.

Changing Listing Information

Commands used: LISTINGS (page 148)

If there is at least one debuggable program routine that has no listing defined, the Interactive Debug listings panel is automatically displayed at Interactive Debug initialization. This can occur if these routines are not all specified in AFFON (page 43), or subsequently not found in an attempt made by Interactive Debug to search a file or data set whose name is generated by Interactive Debug from the given application program name. You can enter missing data set definitions on the listings panel. Press the END key when you are done.

Note: If a restart file is present, this automatic display of the listings panel is deferred until the end of the restart file is reached. If a QUIT command is run from the restart file, the automatic intervention is bypassed.

You can also request the listings panel from the debugging session by entering the LISTINGS command. Here are some sample listings panels:

```
                    INTERACTIVE DEBUG LISTINGS PANEL              ROW  1 OF 5
COMMAND ===>                                                  SCROLL ===> PAGE

PROGRAM UNIT NAME              CMS LISTING FILE      DISPLAY

MAIN_____________________   XYZPROG  LISTING  A1    Y

THCON____________________   THCOM    LISTING  E1    N
AFFA756E LISTING FILE COULD NOT BE FOUND ON ACCESSED DISKS.
THE281J__________________   THE281J  LISTING  A1    Y

THY1866__________________   THY1866  LISTING  *_    N

****************************** BOTTOM OF DATA ********************************
```

Figure 11. Sample Interactive Debug Listings Panel (CMS)

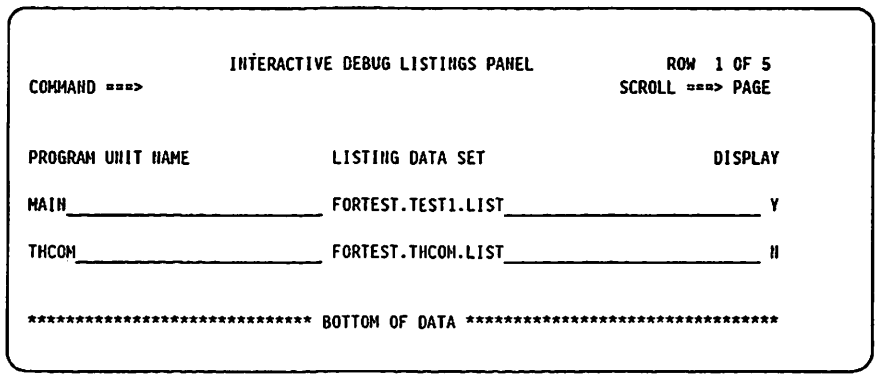

Figure 12. Sample Interactive Debug Listings Panel (TSO)

If the source listing you want does not appear on the panel, you must specify the file ID or data set name for the program unit source listing. To fill in the listings panel, enter the name of the program unit source listing in the CMS LISTING FILE or LISTING DATA SET column. Interactive Debug then automatically fills in the names of all program units found in that listing, and changes the DISPLAY column to YES. If the listing is in a PDS member, include the member name in parentheses at the end of the data set name.

To display a listing in the source window, the DISPLAY column for that listing must specify YES. You may want to change some of these values to NO if you do not want particular program units to be displayed.

In Figure 11 on page 22, the CMS message LISTING FILE COULD NOT BE FOUND ON ACCESSED DISKS indicates that the specified file was not found on any of your accessible disks. In TSO, the equivalent message is LISTING DATASET COULD NOT BE FOUND IN ALLOCATED DATASETS. To correct the problem, make sure the file ID for the listing containing the program unit is correctly spelled and is accessible. If it is necessary to use system commands (CMS or TSO) to make the listing accessible, you can enter them on the command line prefixed by "CMS" or "TSO," as appropriate.

The message SPECIFIED PROGRAM UNIT NAME NOT FOUND IN LISTING indicates that the listing file was found, but did not contain the named program. To correct the problem, make sure the file ID for the listing containing the program unit is correctly spelled and is accessible.

If you receive any of the following messages:

LISTING FILE COULD NOT BE OPENED
LISTING DATASET COULD NOT BE OPENED
LISTING COULD NOT BE LOADED INTO STORAGE
INSUFFICIENT STORAGE TO LOAD LISTING

then the listing data set was found, but cannot be used as a listing in the source window. This could be caused by an invalid record length (it should not be greater than 137 bytes), a very large listing, or security protection by another user (in TSO).

After filling in the listings panel, you return to the main debugging panel by entering END, usually PF key 3.

Using the Split Screen Feature (ISPF/PDF Only)

With ISPF/PDF, you can split the physical screen into two logical screens. With a split screen, you can then edit a source or listing file (or even recompile it, in TSO) while debugging.

To split the screen, use the SPLIT command (or a PF key assigned the SPLIT function, normally PF2), and then use EDIT to look at the appropriate file or data set.

Note: The second screen in split-screen mode cannot be used to run a second session of Interactive Debug (or any other debugging product). You cannot run any program in the second screen that would intercept attentions, unless you let that program terminate before trying to continue with Interactive Debug.

Recompiling a Program while Using a Split Screen (TSO Only)

You can use the split screen to perform a number of tasks. For example, in TSO, you might want to split the screen, recompile a program, and then restart the debugging session using the new compilation. You would need to complete the following steps:

1. Split the screen into two portions.
2. Go into edit mode on the lower half of the screen (usually panel 2).
3. Make changes to the source program in the lower half of your screen.
4. Request the VS FORTRAN compilation panel (usually 4.3), and specify the member name to be compiled.
5. When the program has compiled, request the link-edit panel (usually 4.7), and specify the member name to be link-edited.
6. When the program has been link-edited, end the split-screen mode and issue the RESTART command on the command line of the execution panel.

   You can now debug the newly-compiled program.

Ending the Debugging Session

Commands used: QUIT (page 174)

In ISPF without PDF

Enter the QUIT command to end the Interactive Debug session.

In ISPF/PDF

After entering QUIT, ISPF will then automatically enter BROWSE so you can examine the complete output log (AFFOUT). If you have used the PRINT keyword on any Interactive Debug commands that allow it, ISPF first enters BROWSE for the AFFPRINT file.

After browsing these files, enter the END command, or use the PF key assigned to END (usually PF 3). You will then be presented with the standard ISPF FORE-GROUND PRINT OPTIONS panel, allowing you to print each file and then to keep or delete the file. A sample panel is shown in Figure 13 on page 25.

```
------------------------------ FOREGROUND PRINT OPTIONS ------------------------
OPTION ===>


   PK - Print file and keep              K - Keep file (without printing)
   PD - Print file and delete            D - Delete (erase) file (without printing)

   If END command is entered, file is kept without printing.

FILE ID: MAIN LOG A

SPOOL OPTIONS:
   NUMBER OF COPIES ===> 1               SPOOL CLASS ===> A
   BIN NUMBER       ===>                 'FOR' USER  ===>
   3800 KEYWORDS    ===>

FOR SPOOLING TO ANOTHER USER OR MACHINE:
   USER / MACHINE ID ===>
   NODE / LINK ID    ===>
   TAG TEXT          ===>
```

Figure 13. Foreground Print Options Panel in ISPF in CMS

You can fill in the file ID field and any spooling options, and then enter one of the four options listed at the top of the panel. To leave this panel without printing your log file, enter the END command or use the PF key assigned to END.

Bypassing the BROWSE Step

It is possible to bypass the BROWSE step by modifying the AFFFX11 EXEC in CMS, or the AFFFC11 CLIST in TSO.

In CMS

1. Edit the AFFFX11 EXEC.
2. Add the lines `zfbrows = ''` and `zfprint = ''` to the EXEC as follows:

```
zfbrows = ''
zfprint = ''
/* Browse print file if it exists, set message if not */
afftype = 'PRINT' /* afftype is used in message AFFS006 */
```

3. Alternatively, you can use '/*' and '*/' to comment out the lines

```
If zfbrows ¬= '' then 'ISPEXEC BROWSE FILE('lid')'
If zfprint ¬= '' then
  'ISPEXEC SELECT CHD(ISRFXPRT ISRFPPRT' lid')'
```

These lines occur twice--once for displaying the print file and once for displaying the log file. For example, after you comment out the lines for displaying the log file, the EXEC should look like this:

```
/* Browse log file if it exists, set message if not */
formsg = ''
afftype = 'LOG'
lid = zfname 'LOG A'
'STATE' lid
If rc <= 0  then Do
  Address 'ISPEXEC'
/*  If zfbrows ¬= '' then 'ISPEXEC BROWSE FILE('lid')'
    If zfprint ¬= '' then
      'ISPEXEC SELECT CHD(ISRFXPRT ISRFPPRT' lid')'*/
    Address CMS
End
Else formsg = 'AFFS006'
```

If you want to bypass browsing the print menu, search for the first occurrence of these lines and comment them out also.

In TSO

1. Edit the AFFFC11 CLIST.
2. Search for the following line:

```
SET ZFBROWS = BROWSE        /* ASK FOR BROWSE        */
```

3. Change the line to read as follows:

```
SET ZFBROWS = &Z            /* BYPASS BROWSE         */
```

You can also bypass the print menu by setting ZFPRINT to a null string, as follows:

```
SET ZFPRINT = &Z            /* BYPASS PRINT          */
```

Chapter 3. Debugging in Line Mode

Interactive Debug may be run in *line mode* in either a CMS or a TSO environment. In line mode debugging, input and output are presented sequentially, one line at a time. Line mode debugging is intended primarily for users with access to only a typewriter-like terminal or when ISPF is not installed.

Invoking Interactive Debug in CMS

Using the LOAD and START Commands

This method will *not* make a permanent copy of the executable program. The LOAD command creates a temporary copy of your executable program in virtual storage. The object code from which the executable program is built is either in TEXT files or in text libraries, or both.

1. Make the appropriate VS FORTRAN Version 2 Library text libraries, as well as your own text libraries, available:

   a. For link mode execution:

   ```
   GLOBAL TXTLIB VSF2LINK VSF2FORT CHSLIB userlib...
   ```

   b. For load mode execution:

   ```
   GLOBAL TXTLIB VSF2FORT CHSLIB userlib...
   ```

   For more information on link mode and load mode execution, refer to the *VS FORTRAN Version 2 Programming Guide*.

2. Create a temporary copy of your executable program in virtual storage:

   ```
   LOAD myprog...
   ```

3. Issue:

   ```
   GLOBAL LOADLIB VSF2LOAD
   ```

4. Run the temporary copy of your program that has been built in virtual storage, and invoke Interactive Debug:

   ```
   START * DEBUG
   ```

Using the LOAD and GENMOD Commands

This method will create an executable program that is stored as a nonrelocatable file on your CMS disk. The object code from which the executable program is built is either in a TEXT file or in a member of a text library.

Note: If you have created a nonrelocatable file with a file type of MODULE for your program in the past, you can run the program by completing steps 4 through 6 in the "Running the Program" section that follows.

Creating the MODULE File

1. Make the appropriate VS FORTRAN Version 2 Library text libraries, as well as your own text libraries, available.

   a. For link mode execution:

   ```
   GLOBAL TXTLIB VSF2LINK VSF2FORT userlib...
   ```

b. For load mode execution:

```
GLOBAL TXTLIB VSF2FORT userlib...
```

For more information on link mode and load mode execution, refer to the *VS FORTRAN Version 2 Programming Guide.*

2. Create a temporary copy of your executable program in virtual storage:

```
LOAD myprog...
```

3. Create a nonrelocatable file on your CMS disk:

```
GENMOD modname
```

This command builds a file with the file name assigned as modname, and a file type of MODULE. This program can be run at any time.

Running the Program

4. You may need to issue one or more GLOBAL commands before running your program. Issue the following command if the simulation of extended precision floating-point instructions is required on the machine you are using:

```
GLOBAL TXTLIB CHSLIB
```

5. Issue:

```
GLOBAL LOADLIB VSF2LOAD
```

6. Run the program that is stored as a nonrelocatable file, and invoke Interactive Debug:

```
modname DEBUG
```

where modname is the file name of your MODULE file, as originally specified in the GENMOD command.

With the LKED Command

This method link-edits your program and stores it as a relocatable load module in a member of a CMS LOADLIB.

Note: If you have created a load module from your program in the past, you can run the program by completing steps 3 through 5 in the "Running the Program" section that follows.

Creating a Load Module

1. Issue:

```
FILEDEF SYSLIB DISK VSF2FORT TXTLIB fm
```

where fm is either the file mode of the CMS disk that contains the library VSF2FORT, or an asterisk (*).

2. Issue:

```
LKED myprog (LIBE libname NAME membname
```

where:

myprog — is the file name of the TEXT file that contains your object code.

libname — is the file name of the LOADLIB file into which the resulting load module is to be placed as a member.

membname is the name of the member in the LOADLIB file designated by libname above, into which the resulting load module is to be placed.

Running the Program

3. Issue:

   ```
   GLOBAL LOADLIB VSF2LOAD libname
   ```

   where libname is the file name of the LOADLIB file into which your load module was placed as a member by the LKED command.

4. Issue the following command if the simulation of extended precision floating-point instructions is required on the machine you are using:

   ```
   GLOBAL TXTLIB CMSLIB
   ```

5. Issue FILEDEF statements for the AFFON (page 43) and AFFPRINT (page 42) files, such as:

   ```
   FILEDEF AFFON DISK progname AFFON A
   FILEDEF AFFPRINT DISK progname AFFPRINT A
   ```

6. Run your program, and invoke Interactive Debug:

   ```
   OSRUN membname PARM=DEBUG
   ```

   where membname is the name of the member that contains the load module created with the LKED command.

Using an Invocation EXEC

The easiest way to invoke Interactive Debug in CMS line mode is to use an invocation EXEC.

Figure 14 shows an EXEC that invokes a VS FORTRAN program with the DEBUG option. The EXEC allocates a print file, AFFPRINT, and a log file, AFFOUT. And if they exist, the EXEC also allocates an include file, AFFON, and a restart file, AFFIN. (For more information on these files, see page 39.) Specifying DEBUG on the START statement invokes Interactive Debug. see "Specifying Default Run-Time Options" on page 84.

This sample EXEC assumes your program is compiled, and the entire program is contained in one TEXT file with filetype TEXT. It is also assumed that the TEXT file will be linked with the VS FORTRAN Version 2 Library to run in load mode. If you are running in link mode, change the GLOBAL TXTLIB statement to:

```
GLOBAL TXTLIB VSF2LINK VSF2FORT CMSLIB
```

For more information on link mode and load mode execution, refer to the *VS FORTRAN Version 2 Programming Guide*.

```
/* FORTIAD EXEC: Invokes VS FORTRAN Interactive Debug in line mode. */
arg program .

/* If there is an include file, use it. */
'STATE' program 'INCLUDE *'
if rc = 0 then
  'FILEDEF AFFON DISK' program 'INCLUDE *'
else
  'FILEDEF AFFON DUMMY'

/* If there is a restart file, use it. */
'STATE' program 'RESTART *'
if rc = 0 then
  'FILEDEF AFFIN DISK' program 'RESTART *'
else
  'FILEDEF AFFIN DUMMY'

/* Create a log file, deleting the old log, if any. */
'FILEDEF AFFOUT DISK' program 'LOG (RECFM F LRECL 80'

/* Create a print file, deleting the old print, if any. */
'FILEDEF AFFPRINT DISK' program 'PRINT'

/* Define VS FORTRAN V2 library and required system libraries. */
'GLOBAL TXTLIB VSF2FORT CMSLIB'
'GLOBAL LOADLIB VSF2LOAD'

/* Load and run the program. */
'LOAD' program '(CLEAR'
'START * DEBUG'

exit rc
```

Figure 14. Sample EXEC to Invoke Interactive Debug in CMS Line Mode

If the EXEC in this example is invoked without specifying a second parameter, DEBUG will be the default option. If the EXEC were named FORTIAD, specifying

```
fortiad m06
```

would cause the program named M06 to be invoked with the DEBUG option.

Invoking Interactive Debug in TSO

To invoke Interactive Debug in TSO line mode, you can use either of the following:

```
CALL progname 'DEBUG'

LOADGO progname 'DEBUG'
```

The most convenient method is to use a TSO CLIST. For example, if the CLIST is in 'userid.CLIST.CLIST(FORTIAD)', you can invoke Interactive Debug by typing

```
FORTIAD program-name
```

The following sample CLIST assumes that your program is compiled and the object code is in a data set named program.OBJ.

```
/* FORTIAD CLIST: INVOKES VS FORTRAN INTERACTIVE DEBUG IN LINE MODE. */
PROC 1 PROGRAM
CONTROL MSG NOFLUSH NOLIST NOSYMLIST NOCONLIST

/* MAKE SURE THE OBJECT CODE FOR THE PROGRAM EXISTS. */
IF &SYSDSN(&PROGRAM..OBJ) ¬= OK THEN DO
  WRITE DATA SET &PROGRAM..OBJ COULD NOT BE FOUND.
  EXIT
END

/* IF THERE IS AN INCLUDE (AFFON) FILE, USE IT. */
IF &SYSDSN(&PROGRAM..INCLUDE) = OK THEN +
  ALLOCATE FILE(AFFON) DATASET(&PROGRAM..INCLUDE) SHR
ELSE +
  ALLOCATE FILE(AFFON) DUMMY

/* IF THERE IS A RESTART FILE, USE IT. */
IF &SYSDSN(&PROGRAM..RESTART) = OK THEN +
  ALLOCATE FILE(AFFIN) DATASET(&PROGRAM..RESTART) SHR
ELSE +
  ALLOCATE FILE(AFFIN) DUMMY

/* DELETE OLD LOG FILE, THEN ALLOCATE A NEW ONE. */
IF &SYSDSN(&PROGRAM..LOG) = OK THEN +
  DELETE &PROGRAM..LOG
ALLOCATE FILE(AFFOUT) DATASET(&PROGRAM..LOG) NEW +
         SPACE(10,10) TRACKS RELEASE +
         RECFM(F,B) LRECL(80) BLKSIZE(4000)

/* DELETE OLD PRINT FILE, THEN ALLOCATE A NEW ONE. */
IF &SYSDSN(&PROGRAM..PRINT) = OK THEN +
  DELETE &PROGRAM..PRINT
ALLOCATE FILE(AFFPRINT) DATASET(&PROGRAM..PRINT) NEW +
         SPACE(10,10) TRACKS RELEASE

/* ALLOCATE VS FORTRAN FILES. */
ALLOCATE FILE(FT05F001) DATASET(*)
ALLOCATE FILE(FT06F001) DATASET(*)

/* ALLOCATE VS FORTRAN V2 LIBRARY. */
ALLOCATE FILE(SYSLIB) DATASET('SYS1.VSF2FORT') SHR

/* LINK EDIT AND RUN THE PROGRAM. */
LOADGO &PROGRAM..OBJ 'DEBUG'

/* FREE FILES. */
FREE FILE(AFFON AFFIN AFFOUT AFFPRINT FT05F001 FT06F001 SYSLIB)
```

Figure 15. Sample CLIST to Invoke Interactive Debug in TSO Line Mode

Entering Commands

After invoking Interactive Debug, the copyright information will be displayed:

```
VS FORTRAN VERSION 2 RELEASE 3 INTERACTIVE DEBUG
5668-806 (C) COPYRIGHT IBM CORP. 1985, 1988
LICENSED MATERIALS-PROPERTY OF IBM
```

At this point, execution is temporarily suspended to allow you to enter debugging commands. You can enter a command in the following ways:

- Type the command following the Interactive Debug prompt.
- Use a program function key (PF key).

Typing the Command Following the Interactive Debug Prompt

Commands are normally entered following the Interactive Debug prompt. Commands may also be issued when execution is suspended because of an input request originating in the VS FORTRAN program while TERMIO IAD is in effect. This is described in "Entering Terminal Input" on page 78.

If a command is too long to fit on one line, enter as much of the command as you can, followed by the continuation character (-) to indicate that the command is not yet complete. Press ENTER. Then enter the remainder of the command.

The continuation character (-) must be the last character entered on the command line. You may enter up to 251 characters, including blanks, in one command. If a continuation segment requires leading blanks, type a quotation mark (") first, then the required leading blanks. Interactive Debug will remove the quotation mark and recognize the leading blanks.

Interactive Debug waits for you to type in the continuation of a command by displaying the following prompt:

```
PENDING:
```

If you do not wish to continue the command, type END and the command will be ignored.

Using Program Function Keys

If you are using a 3270-type terminal on VM/SP, you can use program function keys (PF keys) to enter commands in line mode.

Using PF keys to enter commands saves you time because you can simply press a PF key instead of typing in a long command.

To define PF keys, use the SYSCMD command (page 191) with the CP command SET PF.

For example, to set PF key 4 to the STEP command, issue:

```
syscmd cp set pf4 step
```

Chapter 4. Debugging in Batch Mode

Interactive Debug can run in *batch mode*, which creates a non-interactive debugging session. Batch mode debugging can be done in either a CMS or MVS (with or without TSO) environment.

In batch mode, Interactive Debug takes its input from a file or data set with the ddname AFFIN, and writes its normal output to one with the ddname AFFOUT. During a batch session, you cannot interact with the batch job from your terminal. Commands that require user interaction (such as prompt panels) cannot be used.

You might want to run a debugging session in batch mode if:

- You want to restrict the resources used. Batch mode generally uses fewer resources than interactive mode.
- You have a program that might tie up your terminal for long periods of time. If you use batch mode, you can continue to use your terminal for other work while the batch job runs.
- You are using Interactive Debug to collect performance or run-time data about your program. For example, batch mode might be helpful if you want to use LISTFREQ to get statement frequency information, but you do not want to do any debugging.

The following commands *cannot* be used in batch mode:

AUTOLIST	COLOR	DOWN
HELP	LEFT	LISTINGS
MOVECURS	POSITION	PREVDISP
PROFILE	REFRESH	RESTART
RESTORE	RETRIEVE	RIGHT
SEARCH	SIZE	UP
WINDOW	ZOOM	

The invocation procedure you use to start a batch session will depend on the batch procedures set up for you by your installation. Be sure you understand how batch mode is invoked on your system before running Interactive Debug in batch mode.

Invoking Interactive Debug in CMS

There are many batch facilities that can be run on CMS (CMSBATCH, BATCHMON, VMBATCH, and so on). Interactive Debug considers batch mode to occur whenever there is no physical terminal attached to the console, which is true of all the batch facilities listed above. The following gives an overview of an EXEC or job file for a batch job to be run on CMS:

1. The first part of the job is usually a set of control statements for the batch machine, such as a job statement, accounting information, console routing control, and resource limit specifications.

2. The job probably needs CMS commands to link and access the disks containing the application program, the Interactive Debug product, input files, and any other EXECs or programs needed.

3. Next, the job needs commands to run your application with the DEBUG option. These are similar to the commands in the sample EXEC to invoke Interactive Debug in line mode (FORTIAD EXEC), but must include definitions for the AFFIN (page 40) and AFFOUT files (page 39). The sample FORTIAD EXEC is described on page 30.

   If you have a FORTIAD EXEC, modify it as above if necessary, and then enter the following in your job file:

   ```
   EXEC FORTIAD myprog
   ```

   If you want to include the information contained in FORTIAD directly in your job file, it might look like that shown in Figure 16.

```
GLOBAL TXTLIB VSF2FORT CMSLIB
GLOBAL LOADLIB VSF2LOAD
FILEDEF AFFIN DISK myprog AFFIN *
FILEDEF AFFOUT DISK myprog AFFOUT A (RECFM F LRECL 80
FILEDEF AFFON    DUMMY
FILEDEF AFFPRINT DISK myprog AFFPRINT A
LOAD myprog (CLEAR
START * DEBUG
```

Figure 16. Sample Commands for a Batch Debugging Session in CMS

   These sample statements invoke the program with the DEBUG option. They allocate an AFFPRINT file (page 42), but not an AFFON file (page 43). They assume you are starting with TEXT files you want to link with the VS FORTRAN Version 2 Library to run in load mode.

4. Finally, you need CMS commands to send back any output files that were written to disk.

   For example, to send the AFFOUT file to your reader, you might use these commands:

   ```
   CP SPOOL PUN TO userid NOCONT
   PUNCH myprog AFFOUT
   ```

   To send the AFFPRINT file to the printer, you might use these commands:

   ```
   CP SPOOL PRT FOR userid
   PRINT myprog AFFPRINT
   ```

When the EXEC or job file is complete, submit it to the batch machine using the procedures set up for your installation.

Invoking Interactive Debug in MVS

In MVS, Interactive Debug considers batch mode to occur whenever no physical terminal is attached. There are two ways to run Interactive Debug in MVS: with TSO and without TSO.

Using TSO

The sample JCL below runs the program with subroutine timing and tracing active. The program is assumed to be compiled and link edited, with the load module in: userid.FORTRAN.LOAD(program).

The JCL defines AFFON, AFFIN, AFFOUT, and AFFPRINT data sets (page 39). The AFFON entry restricts debugging hooks to only subroutine entry and exit hooks, which is optimum for subroutine tracing and timing. The AFFIN file contains debugging commands to activate tracing and timing, execute the program to completion, and list program unit names.

```
//BATCHIAD JOB (accounting-information),'programmer-name',
//             MSGLEVEL=1,MSGCLASS=Z,USER=userid,
//             TIME=(0,5),NOTIFY=userid,CLASS=A,
//             PASSWORD=password
//FORTIAD EXEC PGM=IKJEFT01,DYNAMNBR=100,REGION=2048K
//STEPLIB  DD  DSN=SYS1.VSF2FORT,DISP=SHR
//SYSTSIN  DD  *
CALL 'userid.FORTRAN.LOAD(program)' 'DEBUG'
/*
//SYSTSPRT DD  SYSOUT=*,DCB=(RECFM=F,LRECL=255,BLKSIZE=255)
//AFFON    DD  *
(all) entry
/*
//AFFIN    DD  *
termio library
trace entry
timer * on
go
listtime
quit
/*
//AFFOUT   DD  SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//AFFPRINT DD  SYSOUT=*
//FT06F001 DD  SYSOUT=*
```

Figure 17. Sample JCL for Batch Mode Debugging in MVS with TSO

Lowercase variables, such as `accounting-information` and `programmer-name`, must be substituted with the appropriate information. (The ability to substitute an asterisk (*) for your user ID and password is system-dependent.) You may need to add DD cards to the last job step to define files used by your program.

Without Using TSO

The sample JCL below performs program sampling. It consists of three job steps: compile, link-edit, and run with Interactive Debug.

The JCL defines AFFPRINT, AFFOUT, AFFIN, AFFLST, and AFFPIF data sets (page 39). Because listing or program information data sets were not defined using the AFFON file, they are taken from AFFLST and AFFPIF respectively.

```
//SAMPLE   JOB (accounting-information),'programmer-name'
//            MSGLEVEL=(1,1),MSGCLASS=Z,USER=userid
//            TIME=(1,0),NOTIFY=userid,CLASS=A,
//            PASSWORD=password
//*********************************************************************
//* COMPILE THE FORTRAN PROGRAM.                                      *
//*********************************************************************
//COMP    EXEC PGM=FORTVS2,REGION=2048K,PARM='SDUMP,SOURCE'
//STEPLIB  DD  DSN=SYS1.VSF2COMP,DISP=SHR
//SYSIN    DD  DSN=fortran-source-name,DISP=SHR
//SYSTERM  DD  SYSOUT=*
//SYSPRINT DD  DSN=&&LISTSET,DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10),RLSE),DCB=BLKSIZE=3429
//VSF2PIF  DD  DSN=&&PIFSET,DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10),RLSE)
//SYSLIN   DD  DSN=&&OBJSET,DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10),RLSE),DCB=BLKSIZE=3120
//*********************************************************************
//* LINK EDIT THE PROGRAM.                                            *
//*********************************************************************
//LKED    EXEC PGM=IEWL,REGION=768K
//SYSLIB   DD  DSN=SYS1.VSF2FORT,DISP=SHR
//SYSLIN   DD  DSN=&&OBJSET,DISP=(OLD,DELETE)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSPRINT DD  SYSOUT=*
//SYSLMOD  DD  DSN=&&LOADSET(MAIN),DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE),DCB=BLKSIZE=32760
//*********************************************************************
//* PERFORM PROGRAM SAMPLING.                                         *
//*********************************************************************
//GO      EXEC PGM=MAIN,REGION=2048K,PARM='DEBUG'
//STEPLIB  DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DSN=SYS1.VSF2LOAD,DISP=SHR
//AFFON    DD  DUMMY
//AFFLST   DD  DSN=&&LISTSET,DISP=(OLD,DELETE)
//AFFPIF   DD  DSN=&&PIFSET,DISP=(OLD,DELETE)
//AFFIN    DD  *
ENDDEBUG SAMPLE
LISTSAMP *.* TOP(100)
ANNOTATE *
QUIT
/*
//AFFOUT   DD  SYSOUT=*,DCB=(RECFM=F,LRECL=80)
//AFFPRINT DD  SYSOUT=*
//FT06F001 DD  SYSOUT=*
```

Figure 18. Sample JCL for Batch Mode Debugging in MVS without TSO

Lowercase variables, such as `accounting-information` and `programmer-name`, must be substituted with the appropriate information. (You can usually substitute an asterisk (*) for your user ID and password if you prefer.) You may need to add DD cards to the last job step to define files used by your program.

Running a Batch Debugging Session

In batch mode, all Interactive Debug prompts are suppressed. Whenever a simulated terminal input line is read, it is echoed to the simulated terminal output, prefixed with an equal sign and asterisk (=*). If the AFFIN file does not exist or the AFFOUT file is not defined, the program is terminated.

Standard corrective action is taken for all VS FORTRAN errors (unless the VS FORTRAN program calls ERRSET to change them). This will cause the program to terminate for unrecoverable errors.

Note: Interactive Debug avoids any real terminal interaction in batch mode. However, it cannot guard against interactions required by the application program or by SYSCMD commands. It is your responsibility to restrict the use of interactive commands during batch sessions.

Ending a Debugging Session

It is your responsibility to make sure that the appropriate batch output files are returned to you or sent to a printer. The batch output files are:

- AFFOUT for debug output (required in batch mode)
- AFFPRINT for debug print output (optional in batch mode)
- Any output files that your VS FORTRAN program writes

Chapter 5. Using Interactive Debug Files

This chapter describes six files that are used by Interactive Debug, and how they should be defined before invoking Interactive Debug. The six files are:

- Log file (AFFOUT)
- Restart file (AFFIN)
- Print file (AFFPRINT)
- Include file (AFFON)
- Listing file (AFFLST)
- Program information file (AFFPIF)

AFFOUT and AFFIN are required for batch mode debugging.

AFFLST and AFFPIF can be used in MVS batch mode debugging only.

Log File (AFFOUT)

During your debugging session, Interactive Debug creates a log of activity for you to examine; you can view this log after completion of the debugging session. The log information is contained in the AFFOUT file. AFFOUT is optional in full screen mode and line mode debugging; it is required for batch mode debugging.

How to Define AFFOUT to Interactive Debug

AFFOUT does not need to exist prior to execution of the VS FORTRAN program, but it must be defined:

In CMS

- Full screen mode: The ISPF invocation procedures define a file named *fname* LOG (where *fname* is the name specified on the FILE ID line of the invocation panel).
- Line mode and batch mode: Issue a FILEDEF statement for AFFOUT.

In TSO

- Full screen mode: The ISPF invocation procedures define a data set named *userid.fname*.LOG (where *fname* is the name specified on the MEMBER line or on the FILE ID FOR DEBUG FILES line on the invocation panel).
- Line mode and batch mode: Issue a DD statement for AFFOUT. (AFFOUT is required for batch mode debugging.)

Characteristics

AFFOUT has the following characteristics:

- All Interactive Debug I/O (except full screen commands) is logged. Unless TERMIO LIBRARY has been specified, all program-initiated terminal I/O is also logged.
- The file must be created with a RECFM of FB and an LRECL of 80.
- Each line starts with a '=' (equal sign).
- Each input line starts with a '=*' (equal sign, asterisk).
- Input and output occurring within an attention exit are not logged; however, the entering of an attention exit is logged.

Restart File (AFFIN)

AFFIN is a file of debugging commands, initially created with either an editor or obtained by editing the output from a previous debugging session. AFFIN is optional in full screen and line mode debugging; it is required for batch mode debugging.

How to Define AFFIN to Interactive Debug

AFFIN must exist and be defined prior to execution of the VS FORTRAN program:

In CMS

- Full screen mode: The ISPF invocation procedures supplied by IBM assume that the AFFIN file is called *fname* RESTART (where *fname* is the name specified on the FILE ID line or MEMBER line of the invocation panel). If you are using a previous output log file as the restart file, use the RESTART command to change the allocation or the content of the log file.
- Line mode and batch mode: Issue a FILEDEF statement for AFFIN.

In TSO

- Full screen mode: It is assumed to be *userid.fname*.RESTART (where *fname* is the name specified on the MEMBER line or on the FILE ID FOR DEBUG FILES line on the invocation panel). If you are using a previous output log file as the restart file, use the RESTART command to change the allocation or the content of the log file.
- Line mode and batch mode: Issue a DD statement for AFFIN. (AFFIN is required for batch mode debugging.)

Characteristics

AFFIN must have the following characteristics:

- It must have a RECFM of F or FB and an LRECL of 80.
- No sequence numbers are permitted in columns 73 through 80.
- SYSCMD and CMS or TSO commands are not permitted without operands.
- Interactive commands, such as HELP, are not permitted.

The file will be read until end-of-file or a QUIT command is encountered. (A QUIT command is forced if end-of-file is reached in batch mode.) If the log file of a previous debugging session is used as the restart file, Interactive Debug will ignore any output contained in the file (anything preceded by only an equal sign, or only an asterisk). It accepts any line preceded by =*, or by nothing, as input.

In full screen mode and line mode, additional input can be entered from the terminal after the commands in the file have been run.

Special Considerations

Using the output log file as an restart file in full screen mode or line mode:

You may want to use the log file (AFFOUT) as input to a subsequent debugging session if, for example, you had to discontinue a debugging session but had not yet solved the problem. To do this, follow these steps:

1. Use the QUIT command to stop debugging.
2. Keep the AFFOUT log file, and edit it to remove the QUIT command.
3. Prior to using the AFFOUT file as input to Interactive Debug, you must rename it, as described above.
4. The log file can now be used as input to retrace the steps taken in the previous session.

After the log file has been run, you should be at the same position as when you stopped the previous session.

Using the output log file from a full screen or line mode debugging session as an restart file to a batch debugging session:

Although it is possible in principle to use the log file from a full screen mode or line mode debugging session as AFFIN input for a batch debugging session, you should be aware that there are some differences.

- TERMIO is likely to have different effects in batch. In MVS it is not possible to connect a data set to a terminal device in batch. However, you can use the DEBUNIT run-time option to specify one or more units that will be treated as terminals. It is then possible, for example, for you to use your output from a full screen session as a restart file, with TERMIO set to IAD.

  In CMS, you can still use TERMINAL in a FILEDEF command. However, do not issue TERMIO LIBRARY if there will be any terminal input, because your program would then attempt to actually get terminal input and the batch job would stall.

  You may want to add TERMIO MSG and NOMSG commands in order to get some notification as the job progresses. (This should normally be used with restraint.)

- Error handling is different in batch. In order to avoid unplanned interactions, the debugger always forces standard fixup for errors in batch mode. Thus ERROR EXIT will never cause an exit to be taken. Error limits are in effect as if Interactive Debug were not present.

- SYSCMD commands with no system command specified are considered an error in batch mode (but cause no harm otherwise). You should also avoid any system commands that might themselves require interaction.
- Prompts are suppressed in batch. For example, GO with a statement identifier does not prompt for confirmation in batch mode, even if an optimized program unit is being debugged.

Including program input in AFFIN in batch mode:

When TERMIO IAD is in effect, all terminal input is obtained from AFFIN, preceded with %, and interspersed with the IAD comments.

In batch mode in MVS, you do not have a terminal available, so it is therefore impossible to connect VS FORTRAN files to a terminal. However, by specifying the DEBUNIT run-time option, which specifies a device to be treated as a terminal, you can use the TERMIO command with MVS batch.

The DEBUNIT option may already be set up for you as a local default whenever you specify the DEBUG option, or you may need to specify it at run-time when you run your program. If you need to specify it, the format is as follows:

```
DEBUNIT(S1[,S2,...])
```

where S is a unit number (such as 5), or an inclusive range of unit numbers (such as 35-40).

Note that in CMS, the commas must be replaced by blanks, unless the program will be invoked by an EXEC2 or REXX EXEC and uses the extended PLIST facilities. If I/O is to be issued to the units, the job stream must also include a FILEDEF (in CMS) or a DD card (in TSO) for each unit.

Remember that, when running in batch, the program must get its input from real data sets, and must send its output to real data sets. In CMS batch, you should not specify TERMIO LIBRARY if there will be any terminal input, because this requests interaction and will cause the batch job to fail.

For more information on TERMIO, see "Entering Terminal Input" on page 78 and "TERMIO" on page 192.

Print File (AFFPRINT)

Commands used: ANNOTATE (page 104), DESCRIBE (page 121), LIST (page 141), LISTBRKS (page 145), LISTFREQ (page 146), LISTSAMP (page 149), LISTSUBS (page 154), LISTTIME (page 156), LISTVEC (page 159), TRACE (page 197), WHERE (page 205)

The commands above allow you to send command output to a print file instead of to the terminal (or in the case of batch mode debugging, to AFFOUT). This can be useful, for example, when you are listing the contents of a large array and want to keep that output separate from the output log. This print file is referred to as the AFFPRINT file. AFFPRINT is optional.

How to Define AFFPRINT to Interactive Debug

AFFPRINT does not need to exist prior to execution of the VS FORTRAN program, but it must be defined:

In CMS

- Full screen mode: The invocation procedures will define a file named *fname* LIST A (where *fname* is the name specified on the FILE ID line of the invocation panel).
- Line mode and batch mode: The EXEC used to run VS FORTRAN programs should contain a FILEDEF statement for AFFPRINT.

```
FILEDEF AFFPRINT DISK progname LIST A
```

In TSO

- Full screen mode: The invocation procedures will define a data set named *userid.fname*.PRINT (where *fname* is the name specified on the MEMBER line or on the FILE ID FOR DEBUG FILES line on the invocation panel).
- Line mode and batch mode: The CLIST used to run VS FORTRAN programs should contain an ALLOCATE command for AFFPRINT.

```
ALLOCATE FI(AFFPRINT) DA(progname.PRINT)
```

Note: To prevent an existing AFFPRINT file from being overwritten, you must rename it before running a program with the same name.

Characteristics

AFFPRINT is created with a RECFM of VB and an LRECL of 137 (and a blocksize of 1733 for TSO).

Include File (AFFON)

The AFFON file is helpful in debugging large programs that are divided into separately compiled sections, and programs in which many sections are known to be free of errors. AFFON is optional in full screen mode, line mode, and batch mode debugging.

The AFFON file allows you to:

- Define separate listing data sets (TSO) or files (CMS) for different program units. In full screen mode, these listings are displayed in the source window during the debugging session, and are used when annotating listings with frequency or sampling data using the ANNOTATE command.
- Define separate program information data sets (TSO) or files (CMS) for different program units. Program information files are generated by the compiler when the VECTOR(IVA) option is specified, and are used by Interactive Debug to gather vector tuning information (page 67).
- Restrict debugging hooks to certain program units or sections of program units. Hooks are used to pass control from the program to Interactive Debug in order to monitor the execution of the program (tracing, timing, and frequency) and to provide breakpoint capabilities (AT, STEP, WHEN, HALT, NEXT).

  By default, Interactive Debug inserts hooks at all executable statements of all debuggable program units. This allows you to set a breakpoint at any

statement in the program. Because there is overhead associated with this ability, it is more efficient to exclude from debugging those program units that are known to be error free. You can also exclude a part of a program unit, which may be helpful if there are heavily run sections of that unit which are error free.

How to Define AFFON to Interactive Debug

AFFON must exist and be defined prior to execution of the VS FORTRAN Version 2 program:

In CMS

- Full screen mode: The invocation procedures must include a correct FILEDEF statement for AFFON. In full screen mode with ISPF/PDF, it is assumed to be *fname* INCLUDE (where *fname* is the name specified on the FILE ID line of the invocation panel).

  Using the RESTART command, you can change the allocation or the content of the existing file prior to restarting.

  Using the information in the AFFON file entries, Interactive Debug will attempt to automatically identify the data sets containing the program listings for use in the source window and with the ANNOTATE command.

- Line mode and batch mode: The EXEC used to run VS FORTRAN programs should contain a FILEDEF statement for the AFFON file, for example:

```
FILEDEF AFFON DISK progname INCLUDE A
```

In TSO

- Full screen mode: The invocation procedures must include a correct ALLOCATE statement for AFFON. In full screen mode with ISPF/PDF, it is assumed to be *userid.fname*.INCLUDE (where *fname* is the name specified on the MEMBER line or on the FILE ID FOR DEBUG FILES line on the invocation panel).

- Line mode and batch mode: The CLIST used to run VS FORTRAN programs, should contain an ALLOCATE command for the AFFON data set.

```
ALLOCATE FI(AFFON) DA(progname.INCLUDE)
```

  where *progname* is the name of the program you want to debug.

Characteristics

An AFFON file consists of records called AFFON entries. An AFFON entry can be:

- A comment entry, which begins with an asterisk.
- A program unit entry, which specifies debugging attributes for a program unit.
- An ALL entry, which is used to change the default attributes.

The attributes that can be specified for a program unit are:

- The name of a data set (TSO) or file (CMS) that contains the source listing for the unit.
- The name of a data set or file that contains the program information for the unit.

- A hook restriction list, describing how hooks should be placed in the program unit.

Syntax

For an explanation of syntax conventions, see page 100.

Syntax for an AFFON Entry

```
{ program-unit-name | (ALL) }
    [ 'listing-file-name' [ 'program-info-file-name' ] ]
    [ ENTRY | NONE | specification-list ]
```

program-unit-name
specifies the name, up to 31 characters long, of the program unit to be selected for debugging.

(ALL)
specifies that defaults are to be set for all program units. The defaults pertain to all program units following the ALL entry, whether they are specified in the AFFON file or not.

More than one ALL entry can be placed in the AFFON file. Subsequent ALL entries override the defaults set by previous ALL entries. If *listing-file-name* (and optional *prog-info-file-name*) or *specification-list* is not specified on an ALL entry, then the default for that field is unchanged.

'listing-file-name'
specifies the name of the file containing the source listing of the program unit. It must be enclosed with quotes.

'program-info-file-name'
specifies the name of the program information file for the program unit. It must be enclosed with quotes. If it is specified, *listing-file-name* must also be specified.

ENTRY
specifies that only ENTRY and EXIT hooks are to be placed in the program unit. This is helpful when you want to increase the accuracy of timing information.

NONE
allows file names to be specified, but bypasses the inclusion of debugging hooks.

specification-list
specifies a list of statement entries, separated by either blanks or commas. Each entry in *specification-list* has the following format:

{n[:n] | *} [DOLOOP | DONEST | DOVECT]

n[:n] specifies either a single or a range of statement numbers (ISNs or sequence numbers) that are to be selected for debugging. ISNs are the default, unless you specified SDUMP(SEQ) when you compiled the program unit.

***** specifies that hooks are to be placed at every statement in the program unit, unless DOLOOP, DONEST, or DOVECT follows the asterisk. If this is the case, only DO loop analysis hooks are to be placed in the program unit.

DOLOOP specifies that only DO loop analysis hooks are to be placed in DO loops only for the preceding ISN or ISN range. If a single ISN was specified, then it must be the ISN of a DO statement.

DONEST specifies that DO loop analysis hooks are to be placed only in the outermost DO loops in nested DO loops for the preceding ISN or ISN range. If a single ISN was specified, then it must be the ISN of a DO statement. If the DO statement is part of a nest of DO loops, it must be the DO statement of the outermost loop.

DOVECT specifies that DO loop analysis hooks are to be placed only in vectorized DO loops for the preceding ISN or ISN range. If a single ISN was specified, then it must be the ISN of the DO statement if it is a vectorized DO loop.

Usage Notes

1. There may be executable statements in the specified ranges that cannot have debugging hooks placed on them because of optimization or vectorization. To find out which statements have had hooks placed on them, you can use the LISTFREQ command.
2. Program units that do not meet the requirements for debugging will not have program hooks inserted, even though they may be in the AFFON list. For compiler and library restrictions that may make a program unit nondebuggable, see "Requirements" on page 5.
3. AFFON can have any record format. The logical record length can be any value up to 255 bytes. Entries can be in lower case.

   If the file exists but has incorrect attributes, you will receive an error message stating that the AFFON file cannot be read.

   AFFON can contain references to sequence numbers for VS FORTRAN programs compiled with SDUMP(SEQ), but AFFON itself cannot be sequenced.
4. The initial ALL entry settings are:

   - In line mode and batch mode, *listing-file-name* and *prog-info-file-name* are undefined:

     ```
     (ALL) '' '' *
     ```

     If you are debugging in MVS batch mode, however, Interactive Debug will try to get the *listing-file-name* and *prog-info-file-name* from the files AFFLST (page 48) and AFFPIF (page 49) respectively.

   - In ISPF/PDF on TSO:

     ```
     (ALL) 'userid.module-name.LIST' 'userid.module-name.PIF' *
     ```

     where `userid` is the prefix set with the PROFILE command and `module-name` is the name of the module being debugged.

   - In ISPF/PDF on CMS:

     ```
     (ALL) 'module-name LISTING *' 'module-name PIF *' *
     ```

     where `userid` is the user's logon id and `module-name` is the name of the module being debugged.

5. Migration: To migrate AFFON file entries for programs created prior to release 3 of VS FORTRAN Version 2, place the following entry at the end of the AFFON file:

   ```
   (ALL) none
   ```

6. If ISN range is followed by DOLOOP, DONEST, or DOVECT, then hooks are placed in the appropriate DO loops in the range. A DO loop is in the range if the ISN of the DO statement of the loop is in the range. The starting and ending ISNs do not have to be DO statements.

7. DO loop analysis hooks are placed in DO loops in all ranges where standard hooks are placed. For example, suppose the following AFFON entry is in the AFFON file:

   ```
   subl '' 'progl.pif' 10:40
   ```

   DO loop analysis hooks would be placed in all DO loops whose DO statement is within the range ISN 10 through 40, in addition to standard hooks at statement boundaries.

8. If any hooks are placed in the program unit, including DO loop analysis hooks, then hooks will also be placed at program unit entry and exit.

9. An ALL entry cannot have a statement restriction list that contains ISN's. This is because the entry is supposed to contain defaults, and particular ISN's are meaningful only for a single program unit.

Examples of AFFON Entries

1. In CMS, make an AFFON entry with debugging hooks placed between statements 6 and 16 and at statement 18 of SUB1.

   ```
   subl     'subl listing *'          6:16  18
   ```

   `subl listing *` is defined as the default listing. ENTRY and EXIT hooks are always placed in the program unit whenever any statement number ranges are specified.

2. In TSO, make an AFFON entry with debugging hooks placed between statements 6 and 16 and at statement 18 of SUB1.

   ```
   Subl     'userid.subl.list'        6:16  18
   ```

3. In MVS, set the default listing data set and program information data set for the program to 'state.fort.listings(prog1)' and 'state.fort.compinfo(prog1)' respectively. Also, by default, place hooks only at outermost nested DO loops, and program unit entry and exit.

   ```
   (all) 'state.fort.listings(progl)' 'state.fort.compinfo(progl)' * donest
   ```

4. In CMS, set the program information file for program unit SHENNA to SHENNA INFOFILE A. Also, only place hooks at entry and exit of the program unit and in the DO loop at ISN 35.

   ```
   shenna '' 'shenna infofile a' 35 doloop
   ```

5. For program unit SUBSRT, place DO loop analysis hooks at vectorized DO loop whose DO statement is at ISN 34, place standard statement boundary hooks in statements at ISNs 25 through 36, and place DO loop analysis hooks in all DO loops whose DO statement is in the range 32 through 65.

   ```
   subsrt 34 dovect 25:36 32:65 doloop
   ```

Examples of AFFON Files

1. The following is an example of a complete AFFON file.

```
* I want to do detailed debugging on these three routines.
(all) 'user33.vlop33.list' *
main
rachek
mcslip

* The following four routines are frequently used and I believe
* have no errors.  So, suppress all hooks.
(all) none
lrakd
lrjjb
lrral
lrsst

* For everything else, use a common listing, and hook entry and exit
* so subroutine tracing is possible.
(all) 'group8.vlop.list' entry
```

Figure 19. Sample AFFON File

2. The following is an example of a complete AFFON file used to gather vector tuning information (page 67).

```
* Define program information file and set hooks placement
* to only DO loop analysis hooks.
(all) " 'user33.vlop33.pif' * doloop

* Stop in RACHEK at ISNs 44 and 48 to temporarily
* turn off DO loop timing.
rachek 44 48 * doloop
```

Figure 20. Sample AFFON File for Vector Tuning

Listing File (AFFLST)

This file is used in MVS batch mode debugging only, and must be used if a listing-file-name is not defined in an AFFON file. Thus, AFFLST is the default ddname used when the listing-file-name is not defined.

Characteristics

AFFLST must have the same characteristics as the source listing generated by the compiler (ddname SYSPRINT).

Program Information File (AFFPIF)

This file is used in MVS batch mode debugging only, and must be used if a program-info-file-name is not defined in an AFFON file. Thus, AFFPIF is the default ddname used when the prog-info-filename is not defined.

Characteristics

AFFPIF must have the same characteristics as the program information file generated by the compiler (ddname VSF2PIF).

Chapter 6. Debugging Tasks

This section describes some of the common debugging tasks you can perform with Interactive Debug.

For complete descriptions of all commands, see "Part Two. Command Reference" on page 99.

Getting On-line Help about Interactive Debug

Command used: HELP (page 135)

While using Interactive Debug, you can get on-line information about:

- Interactive Debug commands, each description containing:
  - Function and syntax
  - Usage notes
  - Examples
- Common Interactive Debug tasks
- Vector report messages that are in the vector report source listing

Invoke help by entering HELP with:

- No operands (or press an equivalent PF key). If the most recently specified command was in error, the help panel for that command will appear.

  Otherwise, the main help menu will appear. From this menu, you can select help on commands, tasks, or a tutorial describing a basic debugging session.

  Figure 21 on page 52 shows the Interactive Debug main HELP menu in full screen mode.

  Figure 22 on page 52 shows the Interactive Debug task HELP menu in full screen mode. The task menu may be accessed from the main menu.

- A command name. This takes you directly to the help information for that command.

  Figure 23 on page 53 shows a sample first screen of a set of screens for an individual command in full screen mode.

- A message identifier. This displays help for a vector report message in the vector report source listing.

  Figure 24 on page 53 shows a sample first screen of a set of screens for a vector message in full screen mode.

```
SELECTION ===>                          VS FORTRAN VERSION 2 INTERACTIVE DEBUG

                                MAIN HELP MENU

Choose a topic by typing its number on the command line and pressing ENTER.
Return to this menu from any topic by typing TOP on the command line and
pressing ENTER.  Return to debugging from any topic by pressing PF3.  Toggle
between this menu and the task menu by pressing ENTER.

   1  Task Menu      16  fixup        31  next         46  right
   2  Tutorial       17  go           32  off          47  search
   3  *,*            18  halt         33  offwn        48  set
   4  annotate       19  help         34  position     49  size
   5  at             20  if           35  prevdisp     50  step
   6  autolist       21  left         36  profile      51  syscmd
   7  backspace      22  list         37  purge        52  termio
   8  close          23  listbrks     38  qualify      53  timer
   9  color          24  listfreq     39  quit         54  trace
  10  dbcs           25  listings     40  reconnect    55  up
  11  describe       26  listsamp     41  refresh      56  vecstat
  12  down           27  listsubs     42  restart      57  when
  13  enddebug       28  listtime     43  restore      58  where
  14  endfile        29  listvec      44  retrieve     59  window
  15  error          30  movecurs     45  rewind       60  zoom
```

Figure 21. Main HELP Menu (full screen mode)

```
SELECTION ===>                          VS FORTRAN VERSION 2 INTERACTIVE DEBUG

                                TASK HELP MENU

Choose a topic by typing its number on the command line and pressing ENTER.
Return to the main menu from any topic by typing TOP on the command line and
pressing ENTER.  Return to debugging from any topic by pressing PF3.  Toggle
between this menu and the main menu by pressing ENTER.

   1  Main Menu                        13  Display variable values
   2  Specify files to debug           14  Handle library errors
   3  Full screen animation            15  Enter program input
   4  Enter cmds in attention          16  Debug optimized code
   5  Debug in batch mode              17  Program sampling
   6  Command continuation             18  Process external files
   7  Command lists                    19  Listing information
   8  Control program execution        20  Set breakpoints
   9  Cursor-sensitive cmds            21  Execute system commands
  10  Display frequencies              22  Program timing
  11  Display pgm information          23  Trace program execution
  12  Display data types               24  Vector tuning assistance
```

Figure 22. Task HELP Menu (full screen mode)

Additional Features of Interactive Debug On-line Help in Line Mode

In either CMS or TSO line mode, you can not only specify a command for which you want help, but also whether you want the function, syntax, or keyword information of a particular command.

```
COMMAND ===>                              VS FORTRAN VERSION 2 INTERACTIVE DEBUG
CLOSE Command                                                      Panel 1 of 4

CLOSE disconnects a VS FORTRAN external file from an input or output unit.
Its usage is similar to that of the CLOSE statement in the VS FORTRAN Version
2 language.  This command allows you to close an external file, for example to
assign another file to the input or output unit, or to examine the contents of
the file.

Abbreviation: None

Syntax:
   CLOSE
     <|number | <qual.>integer-variable | <qual.>integer-array-element|>

number
    is the number of the I/O unit associated with the file that is to be
    closed.

TOP for main menu                                      hit ENTER for next page
```

Figure 23. Sample Command HELP Panel (full screen mode)

```
COMMAND ===>                              VS FORTRAN VERSION 2 INTERACTIVE DEBUG
Vector Report Message ILX0113I                                     Panel 1 of 5

Short Form:    RESTRICTED CONSTRUCT
Long Form:     THE LANGUAGE CONSTRUCT(S) <clist> ARE NOT ANALYZED FOR
               VECTORIZATION.

Explanation:    Indicates that a loop is rejected because it contains some
language construct that cannot be analyzed by the compiler.  These constructs
include assigned and computed GOTO statements and NAMELIST statements.

Supplemental Data:

 <clist>
    is a list consisting of the names of the language constructs
    responsible for the rejection along with the ISNs (Internal Statement
    Numbers) of the statements in which they are used.

                                                       hit ENTER for next page
```

Figure 24. Sample Vector Message HELP Panel (full screen mode)

For example, if you wished to get on-line help about the syntax of the LISTTIME command in CMS line mode, type:

```
help listtime (form
```

In TSO line mode, you would request the same information by typing:

```
help listtime syntax
```

You can request such specific help information in CMS full screen mode. To get syntax information about LISTTIME in full screen mode, type:

```
cms help aff listtime (form
```

Displaying Information about Debuggable Program Units

Command used: LISTSUBS (page 154)

Most debugging activities, such as displaying variables, can be performed only on program units that are considered debuggable by Interactive Debug. To be debuggable, a program unit must be compiled with the SDUMP option. In addition, it must be in storage at the time the VS FORTRAN Library is initialized.

Note: The reentrant part of a program unit compiled with the RENT option need not be in storage at this time.

If you want to see which program units are debuggable, you can use the LISTSUBS command.

The following is a sample of the output produced by LISTSUBS:

```
PROGRAM UNIT     COMPILER    OPT  HOOKED  TIMING
MAINLINE        VSF  2.3.0   V2    YES     ON
SUBBUILD        VSF  1.4.0    3     NO     OFF  RENT NOT LOADED
SUBDOWN         VSF  (TEST)   0    YES     OFF
```

In this example, we see that the program unit MAINLINE was compiled using VS FORTRAN Version 2, Release 3, identified as VSF 2.3.0, and SUBBUILD was compiled with Release 4 of VS FORTRAN Version 1, identified as VSF 1.4.0. VSF (TEST) tells us that SUBDOWN was compiled prior to VS FORTRAN Version 1 Release 4, and the TEST option was specified. In this case, it is not possible to determine the VS FORTRAN release level.

Notice that, for MAINLINE, the OPT column specifies V2. This indicates that vectorization was specified.

In the HOOKED column, "YES" means that hooks are installed at entry and exit points and possibly at some or all statement boundaries as well. You can set breakpoints only in program units that have hooks. The hook settings are controlled by the AFFON file. "NO" in the HOOKED column indicates that no hooks are installed in the program unit. The TIMING column indicates whether the TIMER command has been activated for each program unit listed. This column may also be followed by an indication of the load status for reentrant programs. In our example, SUBBUILD indicates RENT NOT LOADED, meaning that the program unit has not yet been called, and has not been located (although it may actually be in storage).

Referring to Statements or Variables in Other Program Units

Commands used: AT (page 108), LIST (page 141), LISTBRKS (page 145), QUALIFY (page 173)

Programs often contain more than one program unit. A program unit is defined as a main program, a function subprogram, or a subroutine subprogram. Each of these units has its own set of variables, but variables in different program units may have the same names.

A similar situation exists with statement identification; two statements in different program units may have the same statement label or be assigned the

same ISN by the VS FORTRAN compiler. On Interactive Debug commands that refer to statements or variables, you can specify the program unit as a qualifier (in the form of the program unit name followed by a period). If no qualifier is specified in a command that references statements or variables, Interactive Debug resolves these references using the current program qualification.

The current program qualification is normally the program unit that is running (or in which execution is suspended). However, you can change the current qualification by issuing the QUALIFY command.

Note: Each time execution is resumed, Interactive Debug will reset the qualification to the program unit currently running.

Displaying the Current Program Qualification

To see which program unit is currently qualified, enter the QUALIFY command with no parameters:

```
qualify
```

The response will be something like:

```
QUALIFICATION IS MAIN
```

where MAIN is the name of the program unit.

Changing the Current Program Qualification

If the current qualification is MAIN, unqualified statement identifiers and variable names will refer to statements and variables in MAIN. Issuing the command AT /10 will set a breakpoint at the statement labeled 10 in MAIN. If your program has a subroutine (or function) named SUB1 and you wanted to set a breakpoint in that subroutine, you could do so by using an explicit qualifier. For example:

```
AT sub1./10
```

You could also change the current qualification. For example:

```
qualify sub1
```

Now entering:

```
at /10
```

sets a breakpoint at the statement labeled 10 in your subroutine SUB1. To check current breakpoint settings, entering the LISTBRKS command:

```
listbrks
```

would produce output similar to that shown below.

```
CURRENT BREAKPOINTS:
  MAIN.25/10
  SUB1.32/10
CURRENT WHEN CONDITIONS:
  NONE
CURRENT HALT STATUS: OFF
```

Breakpoint settings are displayed for both the main program and the subroutine SUB1.

You can display all the variables in the currently qualified program unit with the LIST command by entering:

```
list *
```

To issue commands without explicit qualification that reference statements or variables in MAIN, change the currently qualified program unit back to the main program by entering:

```
qualify main
```

Now enter LIST * again to display all the variables in MAIN. If, on resuming execution, the breakpoint you set in SUB1 was reached, the currently qualified program unit would be set to SUB1. For example:

```
go
AT SUB1.32/10
list *
```

would list all the variables in SUB1.

Qualifying Individual Variables

You can qualify individual variables without a QUALIFY command by preceding the variable name with the name of the program unit that it belongs to and a period. For example, you can refer to variable x from MAIN as:

```
main.x
```

You can refer to an array element data(10) from sub2 as:

```
sub2.data(10)
```

When qualifying an array element with a symbolic subscript, remember to also qualify the subscript. For example, to display array element data(i) from sub2 while execution is suspended in MAIN, enter:

```
list sub2.data(sub2.i)
```

If you omit the second SUB2, and the current qualification is MAIN, Interactive Debug will look for a value of I in MAIN. If it finds one, it will give you that element of DATA instead of the one you want in SUB2. You must qualify each variable that is not in the currently qualified program unit.

You can reference variables outside the currently executing program unit in any command dealing with VS FORTRAN variables. For example:

```
list (a,b,sub1.alpha,sub2.beta,x,y,z)
set a=sub1.value
when over (sub1.rchg=5.)
```

Setting Breakpoints at Debugging Hooks

Command used: AT (page 108)

The AT command sets breakpoints at specific statements. Breakpoints can be set only at statements that have *debugging hooks*. A hook gives temporary control to Interactive Debug at a specific point within a program (usually at the beginning of an executable statement).

For VS FORTRAN code compiled with the TEST option, debugging hooks are placed in the object code by the compiler. For VS FORTRAN code compiled with the NOTEST and SDUMP options, hooks are inserted into the object code by Interactive Debug at debugging time.

When using the AT command, you can identify the statement either by its statement number or by the statement label, if it has one. Normally, the statement number refers to the Internal Statement Number (ISN) generated by the compiler (see "Statement Identifier Conventions" on page 101 for more details). However, you can specify at compile time that you want to use the sequence numbers in columns 73 through 80 as the statement numbers for your debugging session instead of the ISNs.

As an example, assume that your main program has a write statement labeled 10 and that the compiler has assigned an ISN of 6 to this statement. The listing might show:

```
ISN     6      10      WRITE(*,*)  'Example Program'
```

You can set a breakpoint at this statement by issuing either of the following two commands:

```
at 6
at /10
```

Statement labels are preceded with a slash to distinguish them from ISNs or sequence numbers. Remember that, if no qualifier is specified, Interactive Debug uses the current qualification to determine which program unit this statement is located in. (MAIN is assumed here.) When the statement is reached, execution is suspended and the following message is displayed:

```
AT: MAIN.6/10
```

You may specify a list of statements or a range of statements with the AT command. For example, the following command:

```
at ( 6 /15 14 3)
```

sets breakpoints at statement numbers 6, 14, and 3, and also at statement label 15, providing they are all executable statements and have debugging hooks.

```
at (6:/15)
```

sets breakpoints at every hooked executable statement between the statement whose ISN is 6 and the statement labeled 15. Both statements specified in a range must be executable, and the statement on the left must appear in the program before the statement on the right.

Breakpoints cannot be set if optimization or vectorization causes the statement to be *collapsed*. A collapsed statement is an executable source statement that occupies no object code because of the effects of optimization or vectorization. (The code was either moved to a new location, or eliminated.) For further explanation, see "Debugging Optimized and Vectorized Code" on page 86.

If the statement is non-executable, you cannot set a breakpoint.

If you have specified statement ranges for one or more program units in the AFFON control file, breakpoints can only be set in the specified statement ranges for those program units. Multiple units and ranges can be specified. See chapters on full screen mode, line mode, or batch mode for more information about specifying statement ranges in your environment.

You cannot suspend execution at the "trailer" statement following a logical IF under any of these conditions:

- The trailer statement is a GOTO statement.
- Sequence numbers were used instead of ISNs.

Controlling Program Execution

Commands used: HALT (page 133), NEXT (page 164), OFFWN (page 167), WHEN (page 202)

Note: The following section refers only to statements that have hooks. You cannot suspend execution at a statement that does not have a hook.

You can suspend execution:

- At every executable statement
- At the next executable statement (without knowing which it is)
- At every apparent program branch
- At every entry to and exit from a program unit
- Whenever a user-defined condition is met
- Whenever a specific variable is modified

The HALT command allows you to suspend execution under certain specified conditions. For example:

```
halt stmt
```

suspends execution at every executable statement. This allows you to single step through your program, which can be helpful in finding errors related to the processing flow.

```
halt goto
```

suspends execution at every apparent program branch. Halting can occur for several reasons, including a GOTO, a DO group, and an IF statement.

```
halt entry
```

suspends execution at every entry to or exit from a debuggable program unit.

The HALT command remains in effect until you cancel it with:

```
halt off
```

The NEXT command requests that execution be suspended at the next executable statement. It is similar to the HALT STMT command, except that the NEXT command is temporary and does not remain in effect after execution is suspended.

The WHEN command allows you to suspend execution every time a particular condition is met. You define the condition and supply its name. Later you can refer to the condition by name without redefining it. With WHEN, you can monitor:

- An arithmetic relationship between two variables or between a variable and a constant
- The status of a logical variable
- A change in the value of a variable

For example, to cause execution to be suspended when variable SMITH equals 30, define a condition, such as the one below named RDS. (The name can be one to four alphameric characters, the first character alphabetic.)

```
when rds (smith = 30)
```

Execution is suspended at the first possible statement following the point at which the condition becomes true. For example, if condition RDS was found to be satisfied at the beginning of statement 46 in program unit MAIN, you would receive the following:

```
WHEN:  "RDS" SATISFIED;
CURRENTLY AT MAIN.46
```

The first line tells you which condition was satisfied; the second tells you where execution is suspended. To detect when SMITH changes value, enter:

```
when rds smith
```

Notice that when you want to define a condition that monitors any change in the value of a variable, the variable name is not enclosed in parentheses.

If the value of SMITH is continually being changed, and SMITH changes initially from 2 to 3, you are notified. If SMITH changes to 4, you are notified again.

Examples:

```
when rds1 (smith = md)

when rds2 (smith .lt. 4.7)

when rds3 (rich)
```

In the final example, RICH is a logical variable. In this case, you get control when RICH is true. If the parentheses were omitted, you would get control whenever RICH was modified.

Note: Interactive Debug cannot tell you exactly which statement changed the variable being monitored. (It might have occurred in a section of code that has no debugging hooks.) However, you can get a list of the last ten branches known to the debugger by entering `WHERE FLOW`. This should help you deduce which statement actually caused the change.

To turn off WHEN condition monitoring, use the OFFWN command. For example, to turn off condition RDS, enter:

```
offwn rds
```

To turn off condition DJV along with condition RDS, enter:

```
offwn (djv,rds)
```

To stop all condition monitoring, enter:

```
offwn *
```

WHEN condition monitoring is not automatically turned off when a condition is satisfied. If you do not want to continue monitoring the same condition, you must issue the OFFWN command after the condition has been satisfied.

If you want to reactivate a condition after it has been deactivated by an OFFWN command, enter WHEN with the condition name. For example:

```
when rds
```

Using Command Lists

Command used: AT (page 108)

As part of an AT command, you can specify a list of commands to be run whenever a breakpoint is reached. This allows you to conditionally suspend execution at a specific statement or to specify a list of commands to be executed there. When you specify a command list, you can control whether Interactive Debug will wait for a command, or whether it will continue execution without the need for intervention.

In this example, the value of variable A will be displayed each time sequence number 10 is reached, and execution will then continue.

```
at 10 (list a %go)
```

This is useful for observing how the value of a variable changes in a loop. Here, sequence number 10 could be at the end of a loop and the value of variable A would be displayed at each iteration of the loop.

Note: The percent sign (%) is used to separate commands within the list.

To conditionally suspend execution at a particular statement, you can use the IF and HALT commands within an AT command list. The HALT command will suspend the execution of the command list. For example, entering:

```
at 10 (list (a,b) %if (a.lt.b) halt %go)
```

will cause execution to be suspended at sequence number 10 only if A is less than B. Otherwise, the GO command will cause execution to continue. In either case, the values of A and B will be displayed.

A command in an AT command list that causes execution to resume or halt will cause the remainder of the command list to be ignored. In this example:

```
at /200 (if (a=0) go /10 %if (b=0) go /10 %go /300)
```

if the value of A or B is equal to 0, execution will resume at the statement labeled 10; otherwise, execution will resume at the statement labeled 300.

The following commands cannot be used in a command list:

AUTOLIST	MOVECURS	RIGHT
COLOR	POSITION	SEARCH
DOWN	PREVDISP	SIZE
FIXUP	PROFILE	UP
HELP	REFRESH	WINDOW
LEFT	RESTART	ZOOM
LISTINGS	RESTORE	
LISTSAMP	RETRIEVE	

Displaying Data Types of Variables and Arrays

Command used: DESCRIBE (page 121)

To display the data types of variables and arrays, use the DESCRIBE command. DESCRIBE also displays dimension information for arrays. This command can be useful for checking the attributes of variables and arrays when it is inconvenient to search the source listing for their declarations. It is particularly useful for displaying the dimensions that were passed for dummy array arguments.

For example, to see the data type of the variable "a" in program unit "sub1," enter this command:

```
describe sub1.a
```

By entering an asterisk (*), you can request a display of the type of every variable in the currently qualified program unit. To see a list of all the names in the current program unit, with their data types, enter this:

```
describe *
```

Let's say you have a program that uses a mixture of variables and arrays, and you would like to display the type of a specific group of them. You can specify the variables and arrays in a name list, for example:

```
describe (i,k,sub2.dumchr,r8ary,sub2.r4dumy,llaymn)
```

Possible output:

```
SUB3.I:                          INTEGER*4
SUB3.K:                          INTEGER*4              DUMMY
SUB2.DUMCHR:                     CHARACTER*(*)          DUMMY
SUB3.R8ARY:                      REAL*8
   RANK = 2,  SIZE = 49 ELEMENTS
   DIM 1:     EXTENT = 7, LBOUND = (1), UBOUND = (7)
   DIM 2:     EXTENT = 7, LBOUND = (1), UBOUND = (7)
SUB2.R4DUMY:                     REAL*4                 DUMMY
   RANK = 3;  DUMMY ARRAY ARGUMENT OF INACTIVE SUBPROGRAM OR
              ALTERNATE ENTRY POINT;
              DIMENSION INFORMATION NOT AVAILABLE
SUB3.LLAYMN:                      LOGICAL*1             DUMMY
   RANK = 2,  SIZE = *  ELEMENTS
   DIM 1:     EXTENT = 5, LBOUND =(-2), UBOUND = (2)
   DIM 2:     EXTENT = *, LBOUND = (1), UBOUND = (*)
```

Figure 25. Sample DESCRIBE Output

Dummy arguments are identified by DUMMY in the right-hand column. The length of DUMCHR could not be determined because it is a dummy argument in an inactive program unit, so the length is displayed as CHARACTER*(*).

Dimension information is not available for R4DUMY. This information is never displayed for arrays that are dummy arguments in an inactive program unit, or that are defined only when entered by some other entry point.

L1AYMN is an assumed-size array. The upper bound for the last dimension of such arrays is displayed as an asterisk (*), and the size is indicated as follows:

```
* ELEMENTS
```

In full screen mode, DESCRIBE can be used as a cursor-sensitive command. For example, if the variable "A1" is displayed on the main debugging panel, you could type DESCRIBE on the command line, then position the cursor on "A1" and press ENTER.

Determining Statement Execution Frequency

Command used: LISTFREQ (page 146)

The LISTFREQ command displays the number of times each statement has been run. For example, to see how often the statement whose statement number is 100 has been run, enter:

```
listfreq 100
```

You can specify a statement label or a statement number (an ISN or sequence number), a list of statement labels or statement numbers (in parentheses), or a range of statements. For example, to see how often each statement between sequence numbers 45 and 52 has been run, enter:

```
listfreq 45:52
```

To list the number of times every statement in the currently qualified program unit has been run, enter:

```
listfreq
```

Possible output:

```
STATEMENT            FREQUENCY
MAIN.ENTRY            NO HOOK
MAIN.EXIT             NO HOOK
MAIN.14/80            6
MAIN.15               6
MAIN.16               90
MAIN.17               90
MAIN.18               12
MAIN.19               90
MAIN.20/50            90
MAIN.21               6
MAIN.22               5
```

Figure 26. Sample LISTFREQ Output

The LISTFREQ output tells you which statements have debugging hooks. Those that do will have an execution count in the FREQUENCY column; those that do not will be identified as either COLLAPSED STMT or NO HOOK. COLLAPSED STMT indicates that, because of optimization or vectorization, there is no code at this location. NO HOOK indicates that the statement was not in the AFFON statement range list, or that it is an ENTRY/EXIT to the main program unit.

You can also use LISTFREQ to list statements that have never been run. The ZEROFREQ keyword is used to create such a list. The PRINT keyword can be added to obtain a listing file or a print data set.

```
listfreq 11:74 zerofreq print
```

Interactive Debug will list to the print file all the statements between statement numbers 11 and 74 that have never been run.

Program Sampling

Program sampling can help you identify the portions of your program that are using the most CPU time, without using the resources required when using debugging hooks. The information developed by program sampling can be displayed at your terminal using the LISTSAMP command, or reported as printed output using the ANNOTATE command.

Initiating Program Sampling

Command used: ENDDEBUG (page 124)

You can initiate program sampling by issuing the ENDDEBUG command with the SAMPLE option. This will cause Interactive Debug to interrupt your program's execution periodically to collect sampling data. Data is collected for each statement of every debuggable program unit and for each entry point in every nondebuggable program unit. The sampling data is recorded in two counters for each of the statements and entry points, as follows:

- DIRECT counter

  If the interruption occurs in the code of a debuggable program unit, the DIRECT counter for the interrupted statement is incremented. If the interruption occurs while the VS FORTRAN library is active, the DIRECT counter for the library entry point is also incremented.

- CALLED counter

  If sampling is initiated with the CALLED option of ENDDEBUG, the CALLED counter is incremented for each statement (or entry) included in the sequence of calling program units that lead to the interrupted statement. In addition, if an I/O operation is in process when an interruption occurs, the CALLED counter is incremented for the statement (or entry) that requested the operation.

All interruptions that cannot be associated with any statement or entry are recorded in the *UNKNOWN DIRECT counter. (In order to be properly identified, nondebuggable modules must follow standard MVS linkage conventions.) This count is incremented if an interruption occurs in a program that has no entry identifier or in system code servicing an asynchronous interrupt. The *UNKNOWN CALLED counter is incremented when the save area chain cannot be successfully traced back to the main program.

An additional counter, *LIBRARY DIRECT, shows the sampling count for all VS FORTRAN library modules, other than the mathematical functions and the Error Monitor. This includes lower-level calls to system services. The *LIBRARY CALLED counter is never incremented.

Displaying Program Sampling Information

Commands used: ANNOTATE (page 104), LISTSAMP (page 149)

When your program has completed, you can view the program sampling statistics in three ways:

1. **Terminal display**: Using LISTSAMP, you can display sampling counts either by statement or program unit. Additionally, LISTSAMP allows you to list only those statements or program units having the highest sampling counts, using the TOP(n) option.
2. **Printed output**: Using ANNOTATE, you can copy the source listings to AFFPRINT. Sampling data is added to the right of each statement and is summarized by program unit. All program units and nondebuggable entries that were encountered are included in the summary. Page number references are shown for program units whose listings were annotated.
3. **Bar chart**: In full screen mode, you can overlay the source window with a bar chart that displays the frequency or sampling data for each statement in the listing. You control this feature by your choice of options for the ANNOTATE command, specifying whether you want the bar chart to be in terms of "frequency" (total executions) or "sample" (timer interrupts). You can also overlay the frequency or sampling data with short vector REPORT(SLIST) messages in the source window.

   The bar chart adjusts to the size of the window so that 100% would cover the full width. On a seven color terminal, the bar charts are shown by simply changing the color (reverse video is assumed). Otherwise, asterisks (*) are displayed.

The ANNOTATE and LISTSAMP commands provide options to let you display the DIRECT counts, the CALLED counts, or the sum of the two (ALL) in your terminal or printed output.

The following examples illustrate some forms of the ANNOTATE and LISTSAMP commands. For more examples, please see pages 106 and 152.

1. Copy the source listing for SUB2 to AFFPRINT, annotating it with sampling information.

   ```
   annotate sub2
   ```

2. Overlay the source window with a bar chart showing the sum of the DIRECT and CALLED counters.

   ```
   annotate on all
   ```

3. Display a summary of the sampling counts for all program units.

   ```
   listsamp * summary
   ```

   Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 10 MS; TOTAL NUMBER OF SAMPLES
       WAS 2698.
DIRECT SAMPLES:
 PROGRAM UNIT     SAMPLES %TOTAL
 MAIN                  90   3.34 *
 INIT                   0   0.00
 FUN1                 177   6.56 *
 SUB1                 752  27.87 ******
 S#QRT                 61   2.26
 S#IN                   4   0.15
 A#LOG                  5   0.19
 *LIBRARY            1609  59.64 ************
```

Figure 27. Sample LISTSAMP Output for Programs

(MAIN, INIT, FUN1, and SUB1 are VS FORTRAN program units; S#QRT, S#IN, and A#LOG are VS FORTRAN math library entry points; *LIBRARY contains the counts of sampling occurrences in VS FORTRAN non-math library routines.)

4. Display the sampling counts for SUB1, including the CALLED counts. (Sampling counts will include interruptions which occurred in the code of a statement as well as in any lower-level routines called by the statement.)

   ```
   listsamp sub1.* all
   ```

   Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 10 MS; TOTAL NUMBER OF SAMPLES
       WAS 2698.
SUM OF DIRECT AND CALLED SAMPLES:
STATEMENT          SAMPLES     %UNIT   %TOTAL
 SUB1.ENTRY/EXIT        52      6.47     1.93
 SUB1.8                  7      0.87     0.26
 SUB1.9                 34      4.23     1.26
 SUB1.10               561     69.78    20.79 ****
 SUB1.11/10            146     18.16     5.41 *
 SUB1.12                 4      0.50     0.15
 SUB1.13                 0      0.00     0.00
```

Figure 28. Sample LISTSAMP Output for Programs

Limitations of Program Sampling

1. Use of sampling will cancel any active timer interval at the start of execution (one millisecond per timer interruption).
2. In CMS, the BLIP will be turned off during sampling.
3. Program performance will be slightly degraded due to the execution of sampling code at each interrupt, one millisecond per timer interruption. Use of the CALLED option executes additional code in order to trace back through the call chain.
4. Interruptions that occur in math library routines called by program units compiled with NOSDUMP will be attributed to the program unit instead of the library routine. You can eliminate this inaccuracy by recompiling with the SDUMP option.

5. The accuracy of the STIMER macro, used to provide periodic interruptions on VM and MVS, is sensitive to system activity. Thus, sampling may occur less often than the interval you specified in ENDDEBUG. If your CPU has the virtual interval timer assist facility, and you are using CMS, you may be able to improve the timing values. You can turn on this facility by issuing:

   ```
   SYSCHD CP SET ASSIST ON THR
   ```

6. If you use the STIMER macro in your program, or a system service that uses STIMER, sampling will be discontinued. For example, SVC 99 on MVS uses STIMER in performing dynamic file allocation. (If you use VS FORTRAN dynamic file allocation, however, sampling will *not* be discontinued.)

Program Unit Timing

If you want to time one or more program units and then see the timing information produced by Interactive Debug, use the TIMER and LISTTIME commands.

Initiating Program Unit Timing

Command used: TIMER (page 194)

The TIMER command turns timing on and off, or resets the activation count and time to zero for a specified program unit. When you want to turn timing on for a program unit named LR63, for example, you must issue the command:

```
TIHER LR63
```

Displaying Program Unit Timing Information

Command used: LISTTIME (page 156)

After your program has run, you can use the LISTTIME command to display timing information. The following information is provided for each program unit and entry point:

- Total time: total execution time.
- % total: percentage of total execution time.
- Invocations: number of invocations.
- Average time: total time divided by the number of invocations.

The timing information is presented by entry point, although timing is controlled by the program unit name. LISTTIME shows values only for those program units for which timing has been turned on (using TIMER). To get a printed listing of the LISTTIME information, enter the command:

```
LISTTIHE PRINT
```

To see the information at your terminal, omit the PRINT keyword.

If you want to increase the accuracy of timing information for subroutines, you must minimize the overhead caused by debugging hooks in your program. To do this, use the AFFON file (page 43) to specify hooks only on entry and exit points. Your AFFON file might look like this:

```
(all) entry
```

which will place hooks at entry and exit points of all program units.

Next use the TIMER command to turn timing on for the subroutine you are interested in. If the subroutine calls any other routines, be sure to turn timing on for them also. If you do not, the time spent in any called routines is included in the measurement for the calling routine.

When you ask for your LISTTIME display, you should see a fairly accurate execution time for that subroutine.

Vector Tuning Assistance

Interactive Debug can help you analyze and tune vectorized programs by:

- Gathering vector length and stride information at run time
- Summarizing sample counts by DO loop
- Timing analyzable DO loops
- Providing the capability of displaying a vector report source listing in the source window (full screen mode only)
- Providing help on the vector report messages in the vector report source listing

Program Information File

Before Interactive Debug can gather vector tuning information, your program must have been compiled with the VECTOR(IVA) option. This causes the compiler to generate a program information file, which is needed by Interactive Debug to gather vector tuning information. The file contains information on DO loops and how they are vectorized, locations where hooks need to be placed to determine vector length and stride and DO loop timing, and compiler estimates of vector lengths and strides.

Specify the program information file as an entry in the AFFON file (page 43).

If a program information file is not specified for a program unit, or if a program information file exists but does not contain the necessary specifications, vector tuning information cannot be gathered on the program unit and an error will occur.

DO Loop Analysis Hooks

Interactive Debug places special hooks, called DO loop analysis hooks, into the code of your program to perform vector analysis. They differ from regular debugging hooks in that breakpoints cannot be set, and execution cannot be suspended at these hooks.

By default, DO loop analysis hooks will be placed in every DO loop of every program unit compiled with the VECTOR(IVA) option, in addition to regular debugging hooks. The AFFON file can be used to restrict hooks to only DO loop analysis hooks, for improved execution performance and more accurate DO loop times.

For example, if your program contains a subroutine called MFPDIV, and you want to time a single DO loop at ISN 23, you can limit hooks to only DO loop analysis hooks for this loop by placing this in the AFFON file:

```
mfpdiv 23 doloop
(all) none
```

If you restrict hooks to only DO loop analysis hooks, entry and exit debugging hooks will still be placed in the program unit. These hooks are required for Interactive Debug to function properly, but you may take advantage of them by setting breakpoints to display vector analysis results partially through the execution of the program, and, perhaps, to turn on and off the analysis of certain DO loops.

For example, the following commands will set breakpoints at the entry and exit of subroutine MFPDIV:

```
at mfpdiv.entry (timer * doloop reset% go)
at mfpdiv.exit (listtime * doloop% go)
```

The breakpoint at the entry of the subroutine will reset DO loop times for the subroutine. The breakpoint at the exit of the subroutine will display DO loop times for that execution of the subroutine.

Gathering Vector Length and Stride Information

Commands used: LISTVEC (page 159), VECSTAT (page 200)

Interactive Debug can record the average length and stride of vectors in a program at run-time, and display the averages as well as compiler estimates for length and stride. Averages are calculated over all executions of the loop where vector length and stride recording is active.

The length of a vector is equivalent to the iteration count of the DO loop. Thus, the lengths of all vectors in one DO loop are equal. The stride of a vector is the distance between successive elements of a vector, in units of the array element size.

Length and stride statistics can be gathered for all DO loops that were analyzable by the compiler. Refer to the *VS FORTRAN Version 2 Programming Guide* for a list of conditions that causes an unanalyzable loop.

Run-time vector length and stride information can be used in several ways:

- The average iteration count can be used in the ASSUME directive for DO loops whose iteration count could not be accurately determined at compile-time.
- Average stride information can be used to restructure a DO loop to decrease vector strides. This improves data cache usage and thus the performance of the loop.
- Length and stride information can be used to determine if a DO loop should or should not be vectorized. If the average iteration count of the loop is small, or the average stride of vectors in the loop is large, it may not be profitable to vectorize the loop. A PREFER directive can then be used to force or inhibit vectorization of the DO loop.

- Length and stride information can be compared to compiler estimates to verify that the compiler is using reasonable estimates for vector cost analysis.

The VECSTAT command is used to activate, deactivate, or reset vector length and stride recording. The LISTVEC command is used to display average length and stride for vectors, recorded during execution, as well as compiler estimates for length and stride.

Examples

1. Activate vector length and stride recording for all loops in a program:

   ```
   vecstat *.* on
   ```

2. List vector lengths and strides for all loops in program unit COPY_MATRIX:

   ```
   listvec copy_matrix.*
   ```

   Possible output:

```
COPY_MATRIX.4:
  STATUS = ON
  TOTAL NUMBER OF EXECUTIONS = 33
  AVERAGE ITERATION COUNT = 10
  ESTIMATED ITERATION COUNT = 10
    STATEMENT              ARRAY          AVG STRIDE  EST STRIDE
  S COPY_MATRIX.6          M1                      1           1
  S COPY_MATRIX.6          M2                      1           1
COPY_MATRIX.5:
  STATUS = ON
  TOTAL NUMBER OF EXECUTIONS = 330
  AVERAGE ITERATION COUNT = 10
  ESTIMATED ITERATION COUNT = 10
    STATEMENT              ARRAY          AVG STRIDE  EST STRIDE
  S COPY_MATRIX.6          M1                     10          10
  S COPY_MATRIX.6          M2                     10          10
```

Figure 29. Sample LISTVEC Output

DO Loop Sampling

Commands used: ENDDEBUG (page 124), LISTSAMP (page 149)

Interactive Debug allows you to summarize sampling statistics by DO loop:

- To initiate sampling, use the ENDDEBUG command with the SAMPLE option. Interactive Debug will remove DO loop analysis hooks as well as standard statement hooks before execution under ENDDEBUG.
- To list sampling information, use the LISTSAMP command for DO loops.

Example

Display sampling statistics by DO loop for program unit DISPLAY_MATRIX:

```
listsamp display_matrix.* doloop
```

Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 4 MS; TOTAL NUMBER OF SAMPLES WAS 269.
DIRECT SAMPLES:
DO LOOP                  SAMPLES  %UNIT %TOTAL
DISPLAY_MATRIX.6              84 100.00  31.23 ******
DISPLAY_MATRIX.7              48  57.14  17.84 ****
DISPLAY_MATRIX.15              7   8.33   2.60 *
```

Figure 30. Sample LISTSAMP Output for DO Loops

DO Loop Timing

Commands used: LISTTIME (page 156), TIMER (page 194)

DO loop timing allows you to measure the actual performance improvement gained from vectorizing a loop. You can time an analyzable DO loop before and after vectorization, and then use the PREFER compiler directive to force compilation of the most efficient form of the loop. (For information on the PREFER compiler directive, see the vectorization chapter in *VS FORTRAN Version 2 Programming Guide*.)

Here is a suggested method:

1. Place PREFER SCALAR directives before the loops being analyzed, in the source file of the program.
2. Compile the program with the VEC option.
3. Create an AFFON file with an entry that will restrict hooks to only those needed to accurately time DO loops:

   ```
   (all) * doloop
   ```

4. Invoke the program with Interactive Debug using the AFFON file created.
5. Issue the following commands:

   ```
   timer * doloop
   go
   listtime *.* doloop print
   quit
   ```

   This will run the program, time all DO loops, write the times in the print file, and quit the debugging session.
6. Rename the print file (in order to run the program again without it being erased).
7. Place PREFER VECTOR directives before the loops being analyzed in the source file of the program.
8. Recompile the program.
9. Invoke the program again with Interactive Debug
10. Type in the same commands as in step 5 above.
11. Compare the DO loop timings in the two print files.

Interactive Debug run time is included in the times for DO loops. It is proportional to the number of times a loop is executed, and so should be identical for two runs of the program with the same input data. Thus, times can and should be used for comparison only, not for absolute measures of performance.

DO loop timing cannot be performed on unanalyzable loops.

Vector Report Source Listing

The compiler generates a new source listing when REPORT(SLIST) is specified with the VECTOR compiler option. Interactive Debug uses this listing, when available, instead of the standard listing in the source window and in listing annotation with the ANNOTATE command. In full screen mode, the source window is overlaid with vector messages.

To avoid getting duplicate information on the source listing, compile your program with:

```
NOSOURCE VECTOR(REPORT(SLIST))
```

On-line Help for Vector Messages

Command used: HELP (page 135)

Interactive Debug provides on-line help for vector messages contained in the vector report source listing.

For example, to display information on vector message ILX0109I, type either of the following:

```
help ILX0109I
help 09
```

In full screen mode, you can also get information on vector messages by using HELP as a cursor-sensitive command. Simply type help on the command line, and point the cursor on the line displaying the vector message number, and press ENTER. Or, if you have set HELP to a PF key, point the cursor on the line displaying the vector message number, and press the specified PF key.

Tracing Program Execution

Commands used: TRACE (page 197), WHERE (page 205)

The TRACE command traces control transfers within your program as it executes. To trace each entry to and exit from any subprogram as it occurs, enter:

```
trace entry
```

This produces output similar to the following:

```
TRACE:  FROM MAIN.14 TO SUB1.ENTRY
```

To trace the origin and destination of every apparent branch within the program (including entry to and exit from subprograms) listed by statement identifier, enter:

```
trace goto
```

This produces output similar to the following:

```
TRACE:  FROM SUB1.150/20 TO SUB1.210/40
```

If you don't need to examine your TRACE output right away, you can add PRINT to the TRACE command and send the output to the print data set.

```
trace goto print
```

To stop tracing, enter:

```
trace off
```

The WHERE command shows you the number of the statement at which execution is suspended. This statement will normally be the one run next. For example, if MAIN calls subroutine TAD at sequence number 150, but a breakpoint is set at sequence number 20 in TAD, a WHERE command produces:

```
WHERE:  TAD.20
```

WHERE has a TRBACK keyword that gives you a trace of the calls that got you to your current location. For example, if your program MAIN calls subroutine TAD and execution is suspended, entering:

```
where trback
```

might produce the following:

```
WHERE:  TAD.20
TAD CALLED AT MAIN.150
```

TRBACK output is limited to the transfers between debuggable program units.

WHERE also has a FLOW keyword that gives you a trace of the last ten program transfers run. For example, if you specify the FLOW keyword:

```
where flow
```

you will receive output similar to that in Figure 31.

```
WHERE: MAIN.92                (WHERE response)
TO: MAIN.80     FROM: MAIN.85 (FLOW response, prev. branch)
TO: MAIN.65     FROM: MAIN.70 (Next most recent branch)
TO: MAIN.51     FROM: MAIN.53
TO: MAIN.49     FROM: MAIN.53 (Loop in MAIN)
TO: MAIN.49     FROM: MAIN.53
TO: MAIN.47     FROM: MAIN.40
TO: MAIN.38     FROM: MAIN.20
TO: MAIN.15     FROM: MAIN.10
TO: MAIN.3      FROM: MAIN.10 (Loop in MAIN)
TO: MAIN.3      FROM: MAIN.10
```

Figure 31. Sample WHERE Output

Note that Interactive Debug can only keep track of statements that have debugging hooks. If there is a block of code that has no hooks, it appears to Interactive Debug as if there was a branch from the statement before the block to the statement after it.

The PRINT keyword can be used to send WHERE information to the print data set. For example, you can record the contents of a 100-element array, "ar," at several points in your program by issuing the following sequence of commands each time you want "ar" recorded:

```
where print
list ar(1):ar(100) print
```

This produces a record on your print data set both of the array "ar" and of the exact program location where "ar" had that particular content. Issuing the GO command will resume execution until the next break.

Animating the Execution of Your Program

Command used: STEP (page 189)

When you use the STEP command in full screen mode, Interactive Debug "animates" the execution of your program so you can watch the execution progress. The currently executing line is highlighted in the source window.

The source, log, and monitor (if the AUTOLIST command was specified) windows are refreshed after each step of the program has been run. When the STEP command terminates, or when execution is halted by some other means (such as breakpoints), animation ends.

Controlling the Pace of Program Animation

Command used: PROFILE (page 170)

You can control the timing of animation by modifying the STEP DELAY field in the profile panel. To change the value, enter the command PROFILE.

Displaying Formatted Variable and Array Values

Commands used: AUTOLIST (page 111), LIST (page 141)

Interactive Debug allows you to display the values of variables and arrays through the LIST command. In full screen mode, you can use either the LIST command or the AUTOLIST command. The difference between the two is that LIST displays values in the log window, whereas AUTOLIST displays values in the monitor window. Also, as a program runs and the value of a variable changes, AUTOLIST will reflect the change, whereas LIST will not.

LIST can be used as a cursor-sensitive command in full screen mode.

To display the value of variable A, enter:

```
list a
```

To display the values of variables A, B, C, D, enter:

```
list (a,b,c,d)
```

Similarly, to display array elements ARY(1,1) through ARY(3,4), enter:

```
list ary(1,1):ary(3,4)
```

Output from the LIST command can be very long and you may not want it displayed at the terminal. The LIST command has a PRINT keyword that sends its output to a print data set.

```
list a(1):a(100) print
```

The value of each variable in a LIST command is normally displayed in its correct VS FORTRAN data type and precision: integer, real, complex, and so on. If you want to display the values in a different format, you can use the FORMAT or DUMP keyword. To display the hexadecimal values of some variables, enter:

```
list (a,i,n,r) format(x)
```

To display the values as if they were character strings, enter:

```
list (a,i,n,r) format(a)
```

The DUMP keyword is similar to the FORMAT keyword, but shows the hexadecimal storage location instead of the name. For example, to display the storage location and the hexadecimal value of variable A, enter:

```
list (a) dump(x)
```

The FORMAT and DUMP keywords and codes are described in the reference section in Figure 38 on page 112.

Note: For the LIST command and other commands that allow array element references, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
A(I), A(3), ARY(I+3) or ARY(I-3)
```

Handling Run-Time Errors

Commands used: ERROR (page 128), FIXUP (page 130)

Interactive Debug allows you to control the action taken when you encounter run-time errors. The ERROR command allows you to specify whether to take corrective action or to suspend execution when an error occurs. If you choose the latter, corrective action can be specified by the FIXUP command. It also allows you to suppress the VS FORTRAN library run-time error messages for specific errors.

Identifying Errors

Initially, whenever a run-time error occurs, execution is suspended and VS FORTRAN library run-time error messages are displayed. The ERROR command allows you to change these initial error settings. ERROR uses the identification numbers from the VS FORTRAN library to identify run-time errors. (You can find these error numbers in *VS FORTRAN Version 2 Language and Library Reference*.)

Examples

1. Cause corrective action to be taken and a full diagnostic message to be displayed for error AFB215I.

   ```
   error 215 noexit msg
   ```

2. Cause corrective action to be taken and suppress the diagnostic message.

   ```
   error 215 noexit nomsg
   ```

   After entering this command you will not be notified if error AFB215I occurs again. You can change the error settings back to their original settings by entering any one of the following commands:

   ```
   error 215
   error 215 exit
   error 215 msg
   error 215 msg exit
   ```

   All these commands have the same effect because of keyword defaults.

3. Cause corrective action to be taken, and display full diagnostic messages for errors AFB215I, AFB243I, AFB247I, and AFB289I.

```
error (215 243 247 289) noexit
```

4. Cause execution to be suspended and suppress diagnostic messages for errors AFB215I, AFB216I, AFB217I, AFB218I, and AFB290I.

```
error (215:218 290) nomsg
```

Note: When executing in Interactive Debug (except in batch mode), the VS FORTRAN library does not update the run-time error occurrence counts; therefore, settings in the VS FORTRAN Version 2 error option table that depend on these counts have no effect. This permits unlimited occurrences of errors and messages regardless of the settings in the error option table.

In batch mode, the error counts are updated and tested just as if running without Interactive Debug.

Performing Corrective Action

When execution is suspended because of an error, Interactive Debug displays a message like this:

```
ERROR EXIT: ERROR 243 AT MAIN.6/15
```

and waits for you to enter a command. You may not enter a GO command with a statement identifier. Entering the FIXUP command with no arguments, or a GO command with no statement identification, will cause standard corrective action to be taken and execution to resume. If the error is caused by an incorrect value passed to a VS FORTRAN library mathematical routine, you may use the FIXUP command to specify corrected values to be used to recalculate the function.

With FIXUP, you can assign values to the first, second, or both arguments of a function. The following examples illustrate some of the uses of the FIXUP command:

1. The function has been reevaluated using both arguments, and execution continues.

```
ERRMSG=> AFB241I FIXPI : INTEGER BASE=0, INTEGER EXPONENT=0, LESS THAN
ERRMSG=> OR EQUAL TO ZERO
ERRMSG=>         FIXPI : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM
ERRMSG=> MAIN AT ISN 44 (OFFSET 000954).
INFMSG=>  USER ERROR CORRECTIVE ROUTINE ENTERED.
ERROR EXIT: ERROR 241 AT MAIN.44
IAD/E
fixup arg1(2) arg2(2)
INFMSG=> USER CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
```

Figure 32. Sample Use of FIXUP

2. A new value is given for the first argument (base) only; the second argument (exponent) remains unchanged.

```
ERRMSG=> AFB242I FRXPI : REAL*4 BASE=0.0, INTEGER EXPONENT = 0, LESS
ERRMSG=> THAN OR EQUAL TO ZERO
ERRMSG=>         FRXPI : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM
ERRMSG=> MAIN AT ISN 51 (OFFSET 00099E).
INFMSG=>  USER ERROR CORRECTIVE ROUTINE ENTERED.
ERROR EXIT: ERROR 242 AT MAIN.51
IAD/E
fixup arg1(2.0)
INFMSG=> USER CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
```

Figure 33. Sample Use of FIXUP

3. The function has been reevaluated by changing the second argument; the first argument is unchanged.

```
ERRMSG=> AFB244I FRXPR : REAL*4 BASE=0.0, REAL*4 EXPONENT= 0.0000000E
ERRMSG=> +00,LESS THAN OR EQUAL TO ZERO
ERRMSG=>         FRXPR : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM MAIN
ERRMSG=> AT ISN 61 (OFFSET 000A0E).
INFMSG=> USER ERROR CORRECTIVE ROUTINE ENTERED.
ERROR EXIT: ERROR 244 AT MAIN.61
IAD/E
fixup arg2(1.0e)
INFMSG=> USER CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
```

Figure 34. Sample Use of FIXUP

4. The abbreviation F for FIXUP is used with no arguments. Standard corrective action is taken. The same action would have been taken if GO had been entered instead.

```
ERRMSG=> AFB243I FDXPI : REAL*8 BASE=0.0, INTEGER EXPONENT = 0, LESS
ERRMSG=> THAN OR EQUAL TO ZERO
ERRMSG=>         FDXPI : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM
ERRMSG=> MAIN AT ISN 56 (OFFSET 0009D6).
INFMSG=> USER ERROR CORRECTIVE ROUTINE ENTERED.
ERROR EXIT: ERROR 243 AT MAIN.56
IAD/E
f
INFMSG=> STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
```

Figure 35. Sample Use of FIXUP

To determine what standard corrective action is applied for a particular error or for more information about mathematical functions and their arguments, see *VS FORTRAN Version 2 Language and Library Reference*.

Processing External Files

Commands used: BACKSPACE (page 115), CLOSE (page 116), ENDFILE (page 127), GO (page 131), NEXT (page 164), RECONNECT (page 175), REWIND (page 180)

Interactive Debug allows you to manipulate external files used by a VS FORTRAN program through a set of commands that are similar to corresponding VS FORTRAN statements.

The BACKSPACE command positions a sequentially accessed external file at the beginning of the previous record. For example:

```
backspace 8
```

positions the file connected to I/O unit 8 at the beginning of the last record written or read, allowing it to be written or read again.

The REWIND command positions a sequentially accessed external file at the beginning of the first record of the file. For example:

```
rewind 4
```

positions the file connected to I/O unit 4 at the beginning. This permits you to perform I/O operations as though the file had just been opened. VS FORTRAN supports multiple files under the same I/O unit. The REWIND command sets the VS FORTRAN file name to the first in the sequence of files for the specified I/O unit. For example, if you were currently processing file FT08F003 on I/O unit 8 and entered:

```
rewind 8
```

I/O unit 8 would be connected to file FT08F001, which would be positioned at the beginning of the first record.

The ENDFILE command writes an end-of-file record on a sequentially accessed external file. This causes subsequent I/O operations to be performed on the next file for the specified I/O unit. For example, if you are currently processing file FT05F001 on I/O unit 5, entering:

```
endfile 5
```

causes subsequent I/O operations to be performed, using file FT05F002. If REWIND were issued for I/O unit 5, the filename would be set back to FT05F001.

The CLOSE command disconnects an external file from an I/O unit. For example:

```
close 1
```

disconnects the VS FORTRAN file connected to I/O unit 1, allowing you to associate a different file or data set with that I/O unit, if desired, or to examine the file contents using an editor or browser.

The RECONNECT command resets a file to its original (preconnected) condition. For example, if unit 8 has been closed, you can make it possible for the program to perform additional I/O on unit 8 (without executing an OPEN) by issuing

```
reconnect 8
```

to reconnect unit 8 to file FT08F001. This is necessary only if the OCSTATUS run-time option is in effect.

If you neglected to allocate a file that is needed by your program, you will receive an error message when the program attempts to access that file. If the

program is debuggable and ERROR EXIT is in effect, you can recover from this condition by issuing the following sequence:

```
NEXT
GO
SYSCMD ALLOCATE...(or FILEDEF...)
GO n
```

where n is the statement identifier for the I/O statement. The ALLOCATE or FILEDEF command must be completed as appropriate for allocating the required file. This procedure will work whether or not the OCSTATUS run-time option is in effect.

Using System Commands

Command used: SYSCMD (page 191)

Use SYSCMD to issue a system command while debugging.

CMS Examples:

1. Begin processing a different file using I/O unit 8:

   ```
   close 8
   syscmd filedef 8 disk example data a
   ```

2. In line mode, view your VS FORTRAN source or listing file:

   ```
   syscmd xedit example fortran a
   syscmd type example listing a
   ```

3. Show which CMS files are currently defined:

   ```
   syscmd filedef
   ```

TSO Examples:

1. Begin processing a different data set using I/O unit 8:

   ```
   close 8
   syscmd allocate file (ft08f001) da(example.data) reuse
   ```

2. In line mode, view your VS FORTRAN source or listing file:

   ```
   syscmd edit example.fort
   syscmd list example.list
   ```

3. Show which TSO data sets are currently allocated:

   ```
   syscmd listalc status
   ```

Entering Terminal Input

Command used: TERMIO (page 192)

When a VS FORTRAN program attempts to perform any I/O to or from the terminal, either the Interactive Debug I/O routines or the VS FORTRAN library I/O routines can be used. The TERMIO setting determines which set of routines is used. Using the Interactive Debug routines gives you the advantage of being told which unit is being read, and you have the ability to issue Interactive Debug commands while the read is pending. In addition, when operating in full screen mode, the Interactive Debug routines remain in full screen mode for the read or write.

When running in Interactive Debug, the initial TERMIO setting is IAD, specifying that the Interactive Debug routines should be used for terminal I/O. The TERMIO command allows you to change the setting to either IAD or LIBRARY or to display the current TERMIO setting. When the option selected is LIBRARY, all terminal I/O is performed in line mode in the same manner as when Interactive Debug is not active.

When running in batch mode on MVS, no actual terminal is available. However, the DEBUNIT run-time option may be used to specify that certain I/O units are to be treated as if they were allocated to the terminal. These units then come in the control of TERMIO. For details about using DEBUNIT, see *VS FORTRAN Version 2 Programming Guide*.

When the TERMIO setting is IAD and your program attempts to read from the terminal, the following message is displayed:

```
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
```

where FT05F001 indicates the VS FORTRAN file being read. (In full screen mode, "IAD/R" will also be displayed, in the upper left corner of the main debugging panel.)

When this message is displayed, you may either:

- Issue a debugging command (other than STEP, GO, or ENDDEBUG).
- Enter the requested input, prefaced with a percent sign (%).

The percent sign is not passed to the VS FORTRAN program. For example:

```
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
%743
```

inputs the number 743 to your program. Alternatively, you could have first determined which statement in your program was issuing the read, as follows:

```
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
where
WHERE: MAIN.10
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
%743
```

When the TERMIO setting is IAD:

- The leading and trailing percent signs are removed before the input is sent to the VS FORTRAN program (the trailing percent sign is not required). Whenever you want to include a percent sign as part of your input, you must include a trailing percent sign. For example, to input the string 55% to a VS FORTRAN program, enter:

  ```
  %55%%
  ```

  To enter two percent signs (%%), enter:

  ```
  %%%%
  ```

- To signify an end-of-file from the terminal, use two percent signs. For example:

  ```
  %%
  ```

- All terminal input is padded on the right with blanks to the LRECL of the terminal file.

- All terminal input is converted to uppercase. To avoid this in environments that support mixed-case input, use the TERMIO LIBRARY setting.
- Both Interactive Debug commands and terminal input can be continued on succeeding lines by ending each continued line with a hyphen (-). For example:

```
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
% hello, this input line is -
broken across two lines.
```

- In ISPF, leading blanks are stripped from the beginning of all continuation lines. To avoid this, precede the continuation line with a single quotation mark ("). The quotation mark will not be passed to the program. For example:

```
FT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
% hello, this input line is -
" broken across two lines.
```

- The maximum length of an input line is 151 characters. In CMS and TSO line mode, any input longer than this is truncated.
- When this limit is exceeded in full screen mode, a message is displayed and the input is ignored.
- All output written from your program to the terminal is shown in the log, preceded by the name of the VS FORTRAN file it was written to, and is broken into lines that are 60 characters long.

Continuing Execution without Further Debugging

Command used: ENDDEBUG (page 124)

When you complete debugging and want to finish execution of the program, use the ENDDEBUG command. This command discontinues communication between the program and Interactive Debug until the program terminates. Use ENDDEBUG only when you are certain that further debugging is not required.

Using ENDDEBUG causes the program to run as if the DEBUG run-time option had never been specified, except that attention interrupts are possible, and Interactive Debug will be reentered at termination.

After ENDDEBUG has been issued, all terminal I/O is handled by the VS FORTRAN library I/O routines (as if TERMIO LIBRARY had been issued).

In addition, the VS FORTRAN library begins updating the occurrence count for run-time errors. All error handling is determined by settings in the VS FORTRAN Version 2 error option table. The error summary displayed when the program terminates reflects only errors occurring after ENDDEBUG was issued. WHERE information is not available.

When the program terminates after issuing an ENDDEBUG command, you may issue Interactive Debug commands. However, commands such as LISTFREQ and LISTTIME will display only the information that was current when ENDDEBUG was issued.

Attention interrupts (page 85) can still be used to interrupt the program after ENDDEBUG is issued; however, no program information is available at that time. To resume execution after an attention interrupt, enter a null line. The only command allowed while in an attention exit after ENDDEBUG has been issued is QUIT, which terminates the program execution. You can then list the values of variables or issue other commands that do not require program execution.

Chapter 7. Special Considerations When Using Interactive Debug

This section discusses debugging tasks that are more specialized than those discussed in the previous chapter:

- Recognizing some common errors when setting up a debugging session
- Issuing commands after a program has run
- Handling loops in nondebuggable program units
- Specifying the default run-time options
- Monitoring floating-point equalities
- Referring to unused FORTRAN variables
- Entering commands in an attention-interrupt exit
- Debugging optimized and vectorized code
- Improving program performance while debugging
- Using Interactive Debug double-byte data support

For conventions when using any Interactive Debug command with statement labels or statement numbers, see "Statement Identifier Conventions" on page 101.

Recognizing Common Errors in Setting up a Debugging Session

If you try to begin a debugging session without success, or if you try to debug a program unit that Interactive Debug considers nondebuggable, look through the following list of common errors to find a possible solution:

Problem	Solution
All program units are nondebuggable.	Check your AFFON file (page 43) for incorrectly coded AFFON entries. Insure that the AFFON file matches the program being run. Make the necessary corrections and try again.
One or more program units are nondebuggable.	Check to see whether the program unit was compiled without SDUMP, or was compiled with a FORTRAN compiler level that is not supported. If so, recompile the program unit.

Problem	Solution
Interactive Debug is not loaded and a debugging session is not initialized.	Be sure you have specified the DEBUG run-time option, or have taken steps to override the default option. Be sure you are running your program with a level of the VS FORTRAN library that supports Interactive Debug. If a program was link-edited with an older library, re-link-edit it with the current library.
In ISPF, you receive the message "LOG FILE NOT FOUND," and Interactive Debug never runs.	Be sure that the VS FORTRAN Version 2 libraries are accessible and that you have allocated the FT06F001 file, then rerun the program. Also be sure you have allocated enough storage.

Issuing Commands after a Program Runs

The following commands cannot be issued after your program runs:

AT	HALT	RECONNECT
ENDDEBUG	IF	STEP
ERROR	NEXT	TIMER
FIXUP	OFF	TRACE
GO	OFFWN	WHEN

Handling Loops in Nondebuggable Program Units

Many programs may contain units that cannot be debugged. These may be subroutines coded in some language other than VS FORTRAN Version 1 or VS FORTRAN Version 2, a VS FORTRAN module that is not debuggable, and so forth. It is possible for execution to get caught in a loop in such a nondebuggable program unit. When this happens, the only way to suspend the program is by issuing an attention interrupt, and the only accepted command that will affect program execution is QUIT. When QUIT is entered following an attention interrupt, Interactive Debug terminates the program and allows you to list the values of variables or issue other Interactive Debug commands that do not require further execution. When QUIT is issued again, the debugging session is terminated.

Specifying Default Run-Time Options

You may occasionally need to use a program compiled with a compiler other than VS FORTRAN (COBOL, for example), which then invokes a VS FORTRAN program. However, when you have no FORTRAN main program to accept a parameter, there is no way to explicitly specify DEBUG or any other run-time option. One way to solve this problem is to override the normal VS FORTRAN default options by including a local parameter specification module (AFBVLPRM) when you link-edit your module, or by using a customized library containing a global parameter specification module (AFBVGPRM). You can

create either a local or a global parameter specification program by assembling a small program that invokes a macro supplied with the VS FORTRAN Version 2 library. For details, see *VS FORTRAN Version 2 Programming Guide.*

Monitoring Floating-Point Equalities

Requesting Interactive Debug to monitor equal conditions for floating-point numbers requires caution. Equality comparisons are performed on a bit-by-bit basis. Numbers may appear to be equal in a program designed to be insensitive to minor differences, but may differ by a single bit and not be considered equal when compared by Interactive Debug. Because only simple relations are supported, there is no way to monitor equality to a given precision. See "WHEN" on page 202 for more information.

Referring to Unused VS FORTRAN Variables

If you define a variable in your VS FORTRAN program without assigning an initial value, and never refer to it in your program, no storage is allocated for the variable and it does not appear in the symbol table used by Interactive Debug. If you try to refer to the variable in your debug session, you will get an error message stating that the variable does not exist.

Therefore, if there are VS FORTRAN variables that you may want to reference in Interactive Debug but which are not referenced in the VS FORTRAN program, you should assign initial values to them.

Entering Commands in an Attention-Interrupt Exit

You can interrupt processing in full screen or line mode by issuing an attention interrupt signal. You can use this signal to gain control if your program appears to be looping, or if an Interactive Debug command is producing excessive output.

The way you issue an attention interrupt signal varies with the operating system and with the type of terminal. A line mode terminal typically has a BREAK key, or one marked ATTN. On a 3270-type terminal, you can use the PA1 key. In VM, pressing ENTER may signal attention.

When you enter an attention interrupt, Interactive Debug issues the *attention prompt* (IAD/A), and temporarily suspends your program. Your program is now in an *attention exit*. You can either:

- Enter a null line, which will cause execution to continue. (You will leave the attention exit.)
- Enter an Interactive Debug command.
- Enter QUIT, which will terminate your program. (However, you will still have a chance to issue any Interactive Debug command that is valid after program termination.)

If you issue any of the following commands while in an attention exit, the command will be run and you will remain in the attention exit:

PURGE	To terminate excessive output, such as from a LIST command.
WHERE	To identify the last statement that was begun in a debuggable program unit (parameters are not honored).
NEXT	To request a pause at the next executable statement in a debuggable program unit. However, if your program is looping in a nondebuggable program unit, you may never get back to a debuggable unit.
*** or " (comment)**	To enter a comment. (However, the comment will not be logged.)

If you issue a QUIT command while the program is executing, the program will be terminated and Interactive Debug will accept debugging commands that are not related to execution.

If you issue any other non-fullscreen Interactive Debug command while the program is executing, the command will be saved, the attention routine will be exited, and execution will continue. The saved command is deferred until an executable statement in a debuggable program unit is reached (which may not happen). Note that if the program is looping in a nondebuggable routine, the saved command will never be run. Full screen commands cannot be issued from an attention exit.

Debugging Optimized and Vectorized Code

Interactive Debug allows you to debug VS FORTRAN Version 2 programs compiled with

- Optimization: compiler option OPTIMIZE(0), OPTIMIZE(1), OPTIMIZE(2), or OPTIMIZE(3)
- Vectorization: compiler option VECTOR (or VEC)

Vectorization requires either optimization level 2 or 3, so debugging vectorized code always involves debugging optimized code.

Debugging is least complicated for programs compiled at optimization level 0 with no vectorization. Optimization level 0 provides object code that most closely follows the source code, so a bug found in the executing object code can be most easily and directly traced to its corresponding source statement.

Debugging optimized or vectorized programs may be necessary, but it requires careful, informed interpretation of the results. You must recognize certain actions taken by the compiler for vectorization, and for the different optimization levels. These actions are:

Register optimization: Retaining values in registers instead of in storage.

Common expression elimination: Eliminating duplicated instructions by retaining subexpression values for later use.

Strength reduction: Replacing an operation by a faster one to improve execution of DO-loops..

Code motion: Altering the placement of calculations, usually by moving instructions from inside a loop to outside

Vectorization: Running certain computations in DO-loops with vector instructions rather than scalar instructions.

Interactive Debug will not be able to counter all the effects of optimization or vectorization, but will issue a warning message to inform you that optimized or vectorized code is being debugged. These messages are issued at your first attempt to reference a variable in an optimized or vectorized program unit. Rather than appearing every time an affected command is used, such messages appear only once for each program unit.

If you must debug optimized or vectorized code, you can determine how optimization or vectorization has affected your program by looking at the listing and vector report.

Optimization Examples

Constant Propagation: Constant propagation is a calculation done at compile time. The pre-calculated results are used at run-time. For example:

```
II = 10
JJ = II        is replaced by        JJ = 10
```

Results may be unexpected when you change the value of II at a breakpoint and later look at the value of JJ.

Common Expression Elimination: Common expressions are those in which the result of a calculation is available because of a previous calculation. In the example below, the calculation J + K is performed twice:

```
20      H  = J + K
21      II = 10
22      JJ = J + K     can be replaced by      JJ = H
```

Note the opportunity for unexpected results when you change the value of J or K while at ISN 21. Because the recalculation of the common expression J+K has been eliminated, JJ will be equal to M, not the new sum of J + K. If there is a breakpoint set at ISN 21 and the value of K is altered, it will have no effect on the value assigned to JJ.

Backward Movement: If the variables involved in an expression are not changed in a loop, it may be possible to move an expression outside of the loop. In the example below, A = B could be moved out of the loop:

```
20      X = Y
21      DO 10 I = 1,10
22      ARR(I) = ARR (I) + I
23      A = B
24  10  CONTINUE
25      WRITE(6,*) X,A
```

The program then becomes:

```
20        X = Y
21        A = B
          DO 10 I = 1,10
22        ARR(I) = ARR (I) + I
23
24  10    CONTINUE
25        WRITE(6,*) X,A
```

The optimizer recognizes that both A and B are loop invariant, and the computation can therefore be moved outside the loop. Note that ISN 23 is still there, but is empty, or "collapsed." An attempt to set a breakpoint at ISN 23 would result in an Interactive Debug error message.

Strength Reduction: Strength reduction replaces one operation by a faster one to improve execution of DO loops.

Let's assume that elements of an array such as ARR, in the example below, are 4 bytes long.

```
21        DO 10 J = 1,10
22        ARR(J) = ARR(J) + B
23  10    CONTINUE
```

Since each element is 4 bytes long, the subscript J must be internally multiplied by 4 to obtain the proper offset from the start of the array. The addresses of elements in this array actually increase in steps of 4, rather than in the steps of 1 implied by the DO loop. Thus, the offset takes on values of 4, 8, 12, and so on, as the subscript J takes on values of 1, 2, 3, and so on.

Strength reduction replaces one operation (in this case, the internal multiplication) by a faster one (in this case, addition). The loop code generated for the loop would contain instructions to add 4 to each iteration of an internally maintained offset, rather than keeping the subscript J and multiplying it by 4 on each iteration.

If strength reduction occurred in the example above and you set a breakpoint at the statement following CONTINUE to display the value of J, the value would be 1. This is a misleading value. J is never really used in the optimized loop. A compiler-generated temporary is used instead to calculate the offsets.

Global Register Assignment: Global register assignment is another way to improve the generated code. The example below illustrates the principle involved.

```
21        DO 10 I = 1,10
22        J = J + I
23  10    CONTINUE
```

results in machine code that performs these actions:

```
          load 1 into a register
          store register at I
        * add register to J
          add 1 to register
          compare register contents with 10
          branch (to *) if compare is less than or equal
          store final value at J
```

This is not an exact representation of the machine code, but illustrates the principle of register assignment. Instructions that use registers are faster than instructions that reference storage. For the duration of this loop, I is assigned to a register.

If you were to set a breakpoint at ISN 22 and display the variable I, the value would be 1.

If a program works correctly at OPT(0) and fails when optimized at a higher level, check for uninitialized variables. Frequently, a variable kept in storage at OPT(0) is assigned to a register under OPT(3). This is only one of the effects of debugging an optimized program.

Vectorization Examples

Grouping of Data Elements: When a DO-loop is vectorized, individual loops that operate on single elements are modified to produce less-frequently run loops that operate on *groups* of elements. When you try to set a breakpoint at a particular statement in a program unit that has been vectorized, you may find that the loop index and/or the contents of storage have unexpected values. For example, examine the following code:

```
1         REAL A(100), B(100)
2         DO 2 K = 1, N
3    2      A(K) = B(K)
```

The single loop in the above example may actually look like this in the vectorized code:

```
     DO 2 K  = 1, N, Z
     DO 2 KK = K, K+MIN(N-K,Z-1)
2      A(KK) = B(KK)
```

where Z is the size of the group, and the inner "loop" is actually performed by vector instructions.

After vectorization, the single loop over individual elements becomes a loop over groups of elements. This loop contains a second "conceptual" loop, run in vector instructions, over elements in the group.

The original statement labeled "2" (ISN 3) no longer exists. It has been placed under ISN 2 in a different form. If you try to set a breakpoint there, Interactive Debug may issue a message saying that it cannot find the statement.

Statement Reordering: To vectorize certain loops, the order of execution of statements may have to be modified. When you later try to debug a statement in this loop, you may find that it doesn't appear where you expected to find it.

For example, you may have coded the following:

```
1         DIMENSION A(100),B(100),C(100)
2         DO 500 I = 2,100
3              A(I) = B(I-1) * 3.0
4              B(I) = C(I) * 3.0
5    500  CONTINUE
```

But your vector report listing after vectorization may look like this:

```
ISN  FLAG NESTING *....*...1.........2.........3.........4....
___  ___ _______
0001                    DIMENSION A(100),B(100),C(100)
0002 VECT +-------      DO 500 I = 2,100
0004      |                 B(I) = C(I) * 3.0
0003      |_____            A(I) = B(I-1) * 3.0
```

In this case, ISNs 3 and 4 have swapped positions. They have been placed under ISN 2 in a different form. You will not be able to set breakpoints at these statements.

Loop Distribution: During vectorization, statements coded within a single DO loop are sometimes distributed to separate loops during vectorization.

For example, if you code this loop:

```
1          REAL A(200), B(200)
2          DO 20 I = 2,100,2
3            A(I) = A(I) + 2
4            B(I+2) = B(I) + 2
5   20     CONTINUE
```

your vector report listing might look like this after vectorization:

```
ISN  FLAG NESTING *....*...1.........2.........3.........4....
+___  ___ _______
0001                    REAL A(200),B(200)
0002 VECT +-------      DO 20 I = 2,100,2
0003      |_____            A(I) = A(I) + 2

0002 SCAL +-------      DO 20 I = 2,100,2
0004      |_____            B(I+2) = B(I) + 2
0006                    END
```

You will not be able to set breakpoints inside the original loop. If you attempt to debug statements within the original loop, you will find that ISNs 3, 4, and 5 have disappeared. They have been placed under ISN 2 in a different form.

Commands Affected by Optimization and Vectorization

The following Interactive Debug commands may produce unexpected or inaccurate results when used in debugging optimized or vectorized code.

AT: If you use AT on a statement that has been moved or completely eliminated due to vectorization or optimization, Interactive Debug will issue a message telling you that no breakpoint can be established at that statement.

AUTOLIST: If you use AUTOLIST on an optimized or vectorized program, it is possible that the current value of a variable, or of an array element, is only in a register. As a result, what is displayed by the AUTOLIST command may not be the current value of the variable or variables.

GO: If you use GO on a statement identifier in an optimized program, you will receive a message indicating that results are unpredictable, and requires your confirmation before proceeding. The VS FORTRAN Version 2 optimizer produces code assuming that the possible paths through a module are known—for example, that a sequence of ten assignment statements will always be run in order. Based on this assumption, a register may be loaded once at the beginning, and used by subsequent statements without being reloaded. If you issue a GO command referring to a statement in the middle of that

sequence, you may be bypassing code that causes an important register to be loaded. Results are unpredictable when a statement using that register is subsequently run.

IF: If IF is used to examine the value assigned to a variable, results may be unpredictable if the value is being kept in a register.

LIST: If LIST is used on an optimized or vectorized program, it is possible that the current value of a variable, or of an array element, is only in a register. As a result, what is displayed by the LIST command may not be the current value of the variable or variables.

LISTFREQ: The LISTFREQ command lists statements even though they may have been removed or relocated due to optimization or vectorization, but indicates that these are "collapsed statements" in the "frequency" field. Execution counts cannot be maintained for these statements.

SET: The SET command alters the contents of a storage location assigned to a variable by the compiler. However, optimized or vectorized code may not be using this storage location to contain the current value. Even if the optimized or vectorized code has stored the value in this location, VS FORTRAN Version 2 may use a copy of the register value or values (without reloading it from storage) for the next use of the variable. Furthermore, subsequent instructions may store the values from the registers into storage, thereby overwriting the value just stored by the SET command. Under either of these circumstances, changing the value in storage will not have the desired effect.

WHEN: The WHEN command uses the storage value of a variable, and may not produce correct results if the value is being kept in a register.

Warning Messages

On the first AUTOLIST, IF, LIST, SET, or WHEN command in any program unit compiled with OPT(1), OPT(2), or OPT(3), or with VECTOR, a warning message will appear, stating that the program unit contains optimized or vectorized code. Subsequent commands involving the same unit will not result in further messages.

Improving Program Performance while Debugging

When you debug a program with Interactive Debug, the program runs slower. This is because Interactive Debug checks every statement boundary for breakpoints or other conditions.

This is especially true when you are debugging a program that runs many simple VS FORTRAN statements, with debugging hooks at all (or most) of them.

Ways to Improve Performance

If you find performance seriously affected when running in Interactive Debug, there are a number of things you can do to improve it.

- Limit the amount of input to be processed.
- Limit the number of program units being debugged at one time. The include file (AFFON) can be used to specify which program units you want to debug, allowing the rest to run at full speed. This is especially important

if some subroutines are called often. You can debug some of the program units in one debugging session, and others in another debugging session. For more information on AFFON, see page 43.

- Insert only entry/exit hooks in heavily run subroutines. This may be all you need to decide whether the subroutine is incorrectly changing a variable, for example. If you find that such a subroutine is producing errors, you may be able to use the AT command with a command list to temporarily generate correct values while you are debugging other subroutines. Later, you can restart the debugging session with a different AFFON file and concentrate on the other subroutines that are generating incorrect results.
- Avoid putting hooks in heavily run code, especially if it consists of many simple statements. Use the include file (AFFON) to specify which parts of each program unit are to have hooks inserted in them.

Using Interactive Debug Double-byte Data Support

After you specify the command DBCS YES (page 120), you can:

- **Specify double-byte data in Interactive Debug commands** that contain symbolic names or character data as parameters.

  Mixed character data, or data that contains single-byte and/or double-byte data, is supported. Double-byte data is enclosed by shift-in (X'0F') and shift-out (X'0E') characters.

  If a command containing double-byte data cannot fit on one input line, it can be continued on additional lines in the usual way; specify a single-byte dash (-) as the last non-blank character in the line. If the character immediately preceding the dash is a shift-in character and the first character of the next line is a shift-out character, then the shift-in/shift-out pair will be removed.

- **Display a source listing containing double-byte data** in the source window (full screen mode only)

If the DBCS YES command is not specified, double-byte data will not be correctly interpreted and displayed.

A valid double-byte character consists of two bytes, each of which is in the range of X'41' to X'FE', except for the double-byte character blank, which is represented by X'4040'.

Chapter 8. A Sample Debugging Session

This section will acquaint you with some of the basic concepts and commands of Interactive Debug through a sample line mode debugging session.

1. Type in the following program:

```
@PROCESS
      PROGRAM SAMPLE
      INTEGER A1(10),A2(10),A3(10)
      DO 20 I=1,10                          ! Assign values to elements
        A1(I)=I+1                           ! in arrays A1 and A2
        A2(I)=I-1
20    CONTINUE
      CALL DIVIDE (A1,A2,A3)
      DO 40 J=1,10                          ! Write out contents of
        WRITE (6,30) J,A3(J)                ! array A3
30      FORMAT (5X,'A3(',I2,')=',I5)
40    CONTINUE
      STOP
      END
@PROCESS
      SUBROUTINE DIVIDE (DIVEND,DIV,RES)    ! Divide every element in
      INTEGER DIVEND(10),DIV(10),RES(10)    ! A1 by the corresponding
      DO 10 I=1,10                          ! element in A2, put result
        RES(I)=DIVEND(I)/DIV(I)             ! in array A3
10    CONTINUE
      RETURN
      END
```

Figure 36. Sample VS FORTRAN Version 2 Program

2. Compile the program with the SDUMP and OPT(0) options.
3. Run the program with the DEBUG option.

   The following figure shows how the compiler listing for the SAMPLE program might appear. During the debugging session, this listing will be useful for determining statement identifiers for breakpoints. A statement identifier can be an ISN, a sequence number in columns 73 through 80, or a statement label.

   The compiler assigns a number known as the *internal statement number* (ISN) to each statement in the program. For example, the first executable statement in Figure 37 has an ISN of 3. To reference that statement in an Interactive Debug command, you would normally use this ISN. You can use a qualifier to distinguish ISNs with the same number in different program units.

   However, if you specify SDUMP(SEQ) when you compile your program, refer to the sequence numbers in columns 73 through 80 instead of the compiler-generated ISNs.

   Statements that have a user-specified statement label in columns 1 through 5, such as the CONTINUE statement in program SAMPLE, can also be referenced by that statement label. When a statement label is used, the number must be preceded by a slash (/). The CONTINUE statement in SAMPLE may be referenced as /20 as well as 6.

```
LEVEL 2.3.0 (MAR  1988)        VS FORTRAN           OCT 16, 1987  13:21:50                          PAGE:    1
REQUESTED OPTIONS (EXECUTE):  SDUMP OPT(0)
REQUESTED OPTIONS (PROCESS):
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) NOSXM NOVECTOR IL(DIM) NOTEST NODC NOICA NODIRECTIVE NODBCS NOSAA
                   OPT(0) LANGLVL(77) NOFIPS FLAG(I) AUTODBL(NONE) NAME(MAIN) LINECOUNT(60) CHARLEN(500)
 IF DO   ISN   *....*...1.....2.....3.....4.....5.....6.....7.*.....8
           1         PROGRAM SAMPLE
           2         INTEGER A1(10),A2(10),A3(10)
           3         DO 20 I=1,10                    ! Assign values to elements
     1     4           A1(I)=I+1                     ! in arrays A1 and A2
     1     5           A2(I)=I-1
     1     6   20    CONTINUE
           7         CALL DIVIDE (A1,A2,A3)
           8         DO 40 J=1,10                    ! Write out contents of
     1     9           WRITE (6,30) J,A3(J)          ! array A3
     1    10   30      FORMAT (5X,'A3(',I2,')=',I5)
     1    11   40    CONTINUE
          12         STOP
          13         END
*STATISTICS*   SOURCE STATEMENTS = 13, PROGRAM SIZE = 956 BYTES, PROGRAM NAME = SAMPLE    PAGE:    1.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**SAMPLE** END OF COMPILATION 1 ******                                          TIME STAMP: 87.28913.21.50
LEVEL 2.3.0 (MAR  1988)        VS FORTRAN           OCT 16, 1987  13:21:50                          PAGE:    2
REQUESTED OPTIONS (PROCESS):
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) NOSXM NOVECTOR IL(DIM) NOTEST NODC NOICA NODIRECTIVE NODBCS NOSAA
                   OPT(0) LANGLVL(77) NOFIPS FLAG(I) AUTODBL(NONE) NAME(MAIN) LINECOUNT(60) CHARLEN(500)
 IF DO   ISN   *....*...1.....2.....3.....4.....5.....6.....7.*.....8
           1         SUBROUTINE DIVIDE (DIVEND,DIV,RES)   ! Divide every element in
           2         INTEGER DIVEND(10),DIV(10),RES(10)   ! A1 by the corresponding
           3         DO 10 I=1,10                         ! element in A2, put result
     1     4           RES(I)=DIVEND(I)/DIV(I)            ! in array A3
     1     5   10    CONTINUE
           6         RETURN
           7         END
*STATISTICS*   SOURCE STATEMENTS = 7, PROGRAM SIZE = 736 BYTES, PROGRAM NAME = DIVIDE    PAGE:    2.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**DIVIDE** END OF COMPILATION 2 ******                                          TIME STAMP: 87.28913.21.50
LEVEL 2.3.0 (MAR  1988)        VS FORTRAN           OCT 16, 1987  13:21 :50                         PAGE:    3
SUMMARY OF MESSAGES AND STATISTICS FOR ALL COMPILATIONS
*STATISTICS*   SOURCE STATEMENTS = 13, PROGRAM SIZE = 956 BYTES, PROGRAM NAME = SAMPLE    PAGE:    1.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**SAMPLE** END OF COMPILATION 1 ******                                          TIME STAMP: 87.28913.21.50
*STATISTICS*   SOURCE STATEMENTS = 7, PROGRAM SIZE = 736 BYTES, PROGRAM NAME = DIVIDE    PAGE:    2.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**DIVIDE** END OF COMPILATION 2 ******                                          TIME STAMP: 87.28913.21.50
******* SUMMARY STATISTICS ******* 0 DIAGNOSTICS GENERATED. HIGHEST SEVERITY CODE IS 0.
```

Figure 37. Sample VS FORTRAN Version 2 Program Listing

Before running the first statement, Interactive Debug suspends execution and allows you to enter commands. The log tells you that execution is suspended at ISN 3:

```
WHERE: SAMPLE.3
```

Interactive Debug will prompt you for commands wherever execution is suspended. You can enter commands in upper- or lowercase letters, but all system responses will appear in uppercase letters.

To begin our debugging session, let's list the program units available for debugging.

4. List the program units:

```
listsubs
```

The output to LISTSUBS should look like this:

```
PROGRAM UNIT    COMPILER   OPT   HOOKED   TIMING
SAMPLE         VSF 2.3.0    0      YES      OFF
DIVIDE         VSF 2.3.0    0      YES      OFF
```

Now let's try running the program. The GO command begins execution at the next executable statement. Because we are not aware of any errors in the program, let's try running it without setting any breakpoints.

5. Run the program:

```
go
```

Unfortunately, there *is* an error in the program. You should receive an error message that looks like this:

```
ERRMSG=> AFB209I VFNTH : PROGRAM INTERRUPT - FIXED-POINT DIVIDE EXCEP
ERRMSG=> TION
ERRMSG=>         VFNTH : PSW FFE40009A2020620
ERRMSG=>         VFNTH : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM D
ERRMSG=> IVIDE AT ISN 4 (OFFSET 000210).
INFMSG=> USER ERROR CORRECTIVE ROUTINE ENTERED.
ERROR EXIT: ERROR 209 AT DIVIDE.4
```

Now we know that there is a problem at ISN 4 in program unit DIVIDE. Because ISN 4 contains several variables, it might help to look at the values of the variables there.

6. Display variable information:

```
list (i,divend(i),div(i),res(i))
```

The output to LIST should look like this:

```
DIVIDE.I          =       1
DIVIDE.DIVEND(1)
                  =       2
DIVIDE.DIV(1)     =       0
DIVIDE.RES(1)     =       0
```

Look at the value of our divisor, DIV. At this point in the program, DIV is 0, an invalid value for a division operation. But how did the value become 0? To find out, let's set a breakpoint at the first statement in main program SAMPLE after returning from the call to DIVIDE, using the AT command. Looking at our listing, we see that we will set a breakpoint at ISN 8. Since we are currently in DIVIDE, we must *qualify* the ISN with the name of the main program.

7. Set the breakpoint:

```
at sample.8
```

8. Now resume execution:

   ```
   go
   ```

   Execution is suspended at ISN 8 in main program SAMPLE. You should receive the following messages:

   ```
   INFMSG=> STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
   AT: SAMPLE.8
   ```

   The error in DIVIDE has been temporarily corrected, and execution has resumed up to ISN 8. The AT message indicates where execution is now suspended by displaying the ISN of the statement and the name of its program unit (in this case, SAMPLE, the main program).

   By using the GO command with a statement identifier, the initialization loop can be run again. We can check the values in the three arrays just before DIVIDE is called. To do this, we should set a breakpoint at the statement labeled 20, and use a command list to list the values of the variables at this point:

9. Set another breakpoint:

   ```
   at /20 (list (i,a1(i),a2(i),a3(i)))
   ```

   Before resuming execution, we can verify that we set the correct breakpoints by listing them:

10. List the breakpoints:

    ```
    listbrks
    ```

    The output to LISTBRKS should look like this:

    ```
    CURRENT BREAKPOINTS:
      SAMPLE.6/20
      SAMPLE.8
    CURRENT WHEN CONDITIONS:
      NONE
    CURRENT HALT STATUS: OFF
    ```

    We see that a breakpoint is indeed set at ISN 6, which is also statement label 20, in SAMPLE. So we are ready to resume execution at the beginning of the array initialization loop, ISN 3.

11. Execute SAMPLE at ISN 3:

    ```
    go 3
    ```

    Execution will be suspended at ISN 6, and the values of the variables listed:

    ```
    AT: SAMPLE.6/20
    SAMPLE.I     =          1
    SAMPLE.A1(1) =          2
    SAMPLE.A2(1) =          0
    SAMPLE.A3(1) =          2
    ```

    Again, we can see that the value in the second array, A2, which is referenced indirectly by subroutine DIVIDE as DIV, is 0. For now, let's temporarily correct this value with the SET command.

12. Change the value of A2(1), for the current debugging session only, with:

    ```
    set a2(1) = 1
    ```

13. Remove all the breakpoints:

    ```
    off *
    ```

14. Now run the program:

    ```
    go
    ```

    You should now get this output:

    ```
    FT06F001       A3( 1)=    2
    FT06F001       A3( 2)=    3
    FT06F001       A3( 3)=    2
    FT06F001       A3( 4)=    1
    FT06F001       A3( 5)=    1
    FT06F001       A3( 6)=    1
    FT06F001       A3( 7)=    1
    FT06F001       A3( 8)=    1
    FT06F001       A3( 9)=    1
    FT06F001       A3(10)=    1
    PROGRAH HAS TERHINATED; RC = (0)
    ```

    The last message indicates that the program has completed execution, in this case as the result of the STOP statement. Because the arrays are integer, the values are truncated. The output is correct up to the truncated value, so you can assume that the value of A2 was the bug in the program.

15. End the debugging session:

    ```
    quit
    ```

To permanently correct the problem in this program, you must:

1. Edit the source program so that no value of A2 in DIV is ever 0.
2. Save the new version of the program.
3. Re-compile and run the program.

To edit the source listing without ending the debugging session, use the SYSCMD command:

`syscmd edit sample.for` (TSO)

`syscmd xedit sample fortran a` (CMS)

Part Two. Command Reference

Syntax Notation

The following items explain how to interpret the syntax used in this section:

- UPPERCASE letters and special characters (such as commas and parentheses) are to be coded exactly as shown, except where otherwise noted. You can, however, mix lowercase and uppercase letters; lowercase letters are equivalent to their uppercase counterparts.
- *Italicized* lowercase letters or words indicate variables, such as array names or data types, and are to be substituted.
- Underlined letters or words indicate IBM-supplied defaults.
- Ellipses (...) indicate that the preceding optional items may appear one or more times in succession.
- Braces ({ }) group items from which you must choose one.
- Square brackets ([ ]) group optional items from which you may choose none, one, or more.
- OR signs (|) indicate you may choose only one of the items they separate.
- A command can be abbreviated only with the abbreviation shown.
- Keywords within a command, except those for full screen commands, can be abbreviated to any unique shortened form.

Sample Syntax

```
POUR_CUP
     [ ONE | TWO | THREE ]
     { ESPRESSO | CAFE_DELUX | CAFE_AU_LAIT }
```

The syntax above shows how to specify the imaginary command POUR_CUP. If no number is specified with CUP, ONE will be assumed. A choice must be made between ESPRESSO, CAFE_DELUX, and CAFE_AU_LAIT. According to the syntax rules, all the following are valid:

```
pour_cup espresso
pour_cup three cafe_au_lait
pour_cup one cafe_d
pour_cup cap
pour_cup e
```

The following are *not* valid:

```
pour_cup
pour_cup ca
pour_cup one two
```

Statement Identifier Conventions

Follow these conventions when using any Interactive Debug command with VS FORTRAN statement labels, ISNs, or sequence numbers:

- All statement labels (numbers in columns 1 through 5 of the source program) must be preceded with a slash (for example, /100).
- The type of statement number is determined by the compiler and the options used when the program was compiled. A statement number is either an ISN or a sequence number.
  - If the program was compiled with VS FORTRAN Version 2, you can use either the ISN or the sequence number. ISNs are the default; to use sequence numbers, specify the compiler option SDUMP(SEQ).
  - If the program was compiled with VS FORTRAN Version 1, and if both TEST and NOSDUMP were in effect or if VS FORTRAN Release 2 was used, and if the first record has a number in columns 73 through 80, use the sequence numbers. If the first record does not have a number in positions 73 through 80, use the internal statement number (ISN) provided in the compiler listing. VS FORTRAN Version 1 sequence numbers are not supported in full screen mode.
  - For all other combinations of VS FORTRAN Version 1 options, use the ISN.

Commands Summarized by Function

Controlling Program Execution	AT ENDDEBUG GO HALT LISTBRKS NEXT OFF OFFWN RESTART STEP WHEN
Monitoring and Modifying Variables	AUTOLIST DESCRIBE IF LIST QUALIFY SET
Processing Sequential Files	BACKSPACE CLOSE ENDFILE RECONNECT REWIND
Controlling Full Screen Display	COLOR DOWN LEFT LISTINGS MOVECURS POSITION PREVDISP PROFILE REFRESH RESTORE RETRIEVE RIGHT SEARCH SIZE UP WINDOW ZOOM
Handling Run-Time Errors	ERROR FIXUP
Gathering Vector Tuning Information	LISTVEC VECSTAT
Tracing and Timing	ANNOTATE LISTFREQ LISTSAMP LISTSUBS LISTTIME TIMER TRACE WHERE
General	* or " (comment) DBCS HELP PURGE QUIT SYSCMD TERMIO

* or " (Comments)

Function: A comment may be inserted into the Interactive Debug log by preceding it with either an asterisk (*) or a quotation mark ("). Comments are useful in helping you identify items for later examination.

Abbreviation: None

Syntax

```
{* | "}
     [comment]
```

comment
is any character string. This character string will appear in the log of the debugging session.

Usage Notes

1. In CMS, the use of a quotation mark (") to identify a comment may be inhibited because the (") mark is normally assigned to be the CMS escape character. In this case, a double quotation mark is required to enter a comment. For example:

   ```
   "" This is a comment.
   ```

2. Entered comments are ignored:
   - in a command list as part of an AT command
   - in an attention exit
   - as the command portion on the IF command

Examples

```
*   This is how comments are inserted into the log.

"   A quotation mark also works.
```

ANNOTATE

Tasks:

Specifying Output to a Print File (page 42)
Program Sampling (page 63)

Function: ANNOTATE provides program sampling or frequency data in either:

- A source listing to the AFFPRINT file
- A bar chart overlay in the source window (full screen mode only)

Abbreviation: AN

Format 1

Syntax for Copying Source Listings to a Print File

```
ANNOTATE
     {unit | (unit-list) | * }
     [SAMPLING [DIRECT | CALLED | ALL] | FREQUENCY]
```

Format 2

Syntax for Providing a Bar Chart in the Source Window

```
ANNOTATE
     {ON | OFF | TOGGLE}
     [SAMPLING [DIRECT | CALLED | ALL] | FREQUENCY | MESSAGE]
```

Format 3

Syntax for Querying ANNOTATE Settings

```
ANNOTATE
```

unit
specifies the name of a program unit whose listing is to be copied to a print file (AFFPRINT) with sampling or frequency data added. The program unit must be a VS FORTRAN program unit compiled with the SDUMP option. Listing files must be identified either by specification in AFFON or on the Interactive Debug Listings Panel.

unit-list
specifies a list of program units whose listings are to be copied to AFFPRINT with sampling or frequency data added. (See restrictions under *unit*, above.)

*
specifies that all available program unit listings are to be copied to AFFPRINT with sampling or frequency data added. (See restrictions under *unit*, above.)

ON | OFF | TOGGLE
ON, OFF, and TOGGLE are used in full screen mode only.

ON specifies that the source listing window is to be shown with overlaid sampling or frequency bar charts. ON cannot be abbreviated.

OFF specifies that overlaid sampling or frequency bar charts are *not* to be shown in the source window. OFF cannot be abbreviated.

TOGGLE specifies that overlaying of the source window with sampling or frequency bar charts is to be changed to ON if currently OFF, or to OFF if currently ON. This form is intended for assignment to a PF key. TOGGLE cannot be abbreviated.

SAMPLING
specifies sampling information annotation. For Format 1, SAMPLING is the default when neither SAMPLING or FREQUENCY is specified.

DIRECT | CALLED | ALL
DIRECT counts interruptions occurring in the code.

CALLED counts interruptions occurring in lower-level routines. This option is valid only if sampling was initiated with CALLED.

ALL specifies that sampling counts are to be the sum of the DIRECT and CALLED counts.

FREQUENCY
specifies statement frequency annotation information.

MESSAGE
(Format 2 only) specifies that short vector REPORT(SLIST) messages are to replace sampling or frequency annotation (whichever was previously specified) in the source window, if the program is compiled with REPORT(SLIST).

Usage Notes

1. Annotated information listings show a summary of all program units and entries for which sampling counts have been accumulated. This summary includes non-FORTRAN units, FORTRAN units compiled with NOSDUMP, and program units not specified in the unit list operand of the command.
2. When SAMPLING is specified along with the ON, OFF, or TOGGLE options, the current ANNOTATE settings are unchanged if FREQUENCY, DIRECT, CALLED, or ALL are not specified. The initial ANNOTATE setting is

   ```
   ON MESSAGE
   ```

   If sampling is done, the setting is automatically changed to

   ```
   ON SAMPLING DIRECT
   ```
3. Only *unit* | (*unit-list*) | * may be used in a restart file or in batch mode. Annotation is then limited to listings identified in the AFFON or AFFLST (for MVS batch mode debugging) file.
4. Histograms are scaled at the subroutine level such that the most CPU-intensive statement within any subroutine receives the maximum histogram width.
5. With full screen message annotation, messages are truncated on the right so that the message overlays at most 50% of the width of the source window.

Examples

1. Overlay the source window with bar charts indicating the sum of both DIRECT and CALLED sampling.

   ```
   annotate on all
   ```

2. Remove the bar chart overlays from the source window.

   ```
   annotate off
   ```

3. Query the ANNOTATE settings.

   ```
   annotate
   ```

4. Copy the VS FORTRAN source listings (if known) for all program units to AFFPRINT, annotating them with DIRECT sampling information.

   ```
   annotate *
   ```

5. Copy the VS FORTRAN source listing for program unit SUB2 to AFFPRINT, annotating it with frequency information.

   ```
   annotate sub2 frequency
   ```

   Possible output:

```
AFF563I VS FORTRAN INTERACTIVE DEBUG V2 R3 ANNOTATED LISTINGS:
AFF568I FREQUENCIES:
LEVEL 2.3.0 (MAR  1988)        VS FORTRAN           DEC 15, 1987  14:34:49                                    PAGE     1
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED  TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) NOSXM NOVECTOR IL(DIM) NOTEST NODC NOICA NODIRECT IVE NODBCS NOSAA
                      OPT(3) LANGLVL(77) NOFIPS   FLAG(I)  AUTODBL(NONE)  NAME( MAIN)    LINECOUNT(60)      CHARLEN(500)
 IF DO   ISN   *....*...1.....2.....3.....4.....5.....6.... .....7.*.......8 FREQUENCY DISTRIBUTION
          1        subroutine sub2                                                          14
          2          integer i, j, k
          3          real a(10,10,10)
          4          do 10 i = 1, 10                                                        14
     1    5            call sub3 (2, 80, 4)                                                132
     1    6   10       continue                                                            131
          7          do 20 i = 1, 10                                                        13
     1    8            do 20 j = 1, 10                                                     130
     2    9              do 20 k = 1, 2                                                   1300 ******
     3   10   20           a(i,j,k) = a(k,j,i)                                            2600 **********
         11          return                                                                 13
         12        end                                                                       0
**SUB2** END OF COMPILATION 3 ******                                                TIME STAMP: 87.34914.34.49

AFF568I FREQUENCIES:
AFF347I PROGRAM UNIT                          PAGE FREQUENCY DISTRIBUTION
AFF348I   CTAFF1                                    191
AFF348I   SUB1                                       43
AFF348I   SUB2                                1     4360 ******
AFF348I   SUB3                                      9165 **********
```

6. In full screen mode, request that the source window be overlaid with statement frequency annotation. (Note that if you are using an extended color terminal, your output will look a little different.)

   ```
   annotate on freq
   ```

   Possible output:

```
IAD/F Q: SUB1                          W: SUB1.11
COMMAND ===>                                                SCROLL ===> HALF
SOURCE  0----+----1----+----2----+----3----+----4----+----5----+- LINE: 1 OF 14
        C ************************************************************** ...
        C * SUBROUTINE SUB1                                              ...
        C ************************************************************** ...
       1       SUBROUTINE SUB1                                          ...
       2       INTEGER I                                                ...
       3       DO 10 I = 1, 2                                         *  17
       4         IF (I.EQ.1) THEN                                    ***  39
       5           CALL SUB3                                ********** 296
       6         ENDIF                                                  6
       7         IF (I.EQ.2) THEN                                 ****  56
       8           CALL SUB2                                  ********** 218
       9         ENDIF                                                *  17
      10 10    CONTINUE                                             *****  73
      11       END                                                   **  34
****************************** BOTTOM OF SOURCE ******************************
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6 LINE: 8 OF 12
0008 * q sub3
0009 * go
0010 PROGRAM HAS TERMINATED; RC ( 0)
0011 * q sub1
0012 * annotate on freq
```

7. In full screen mode, request vector report message annotation.

   ```
   annotate on message
   ```

   Possible output:

```
IAD/F Q: VREP23                        W: VREP23.16
COMMAND ===>                                                SCROLL ===> HALF
SOURCE  0----+----1----+----2----+----3----+----4----+----5----+ LINE: 12 OF 34
      10 U        DO 140 I = 1, 7              USER FUNCTION OR SUBROUTINE  1
      11 S1         DO 130 J = 1, 128         NESTED NON-CONSTANT INDUCTION  1
      12 V2           DO 130 K = 1, 128                                    1
      13 s1             CC(K,J,I) = BB(K,J,I)            EQUIVALENCE USED 12
      14 s1             CC(K,J,I) = BB(K,J,I) * 23       EQUIVALENCE USED 12
      15 v2             CC(K,J,I) = I + J + K                            12
      16 v2             DD(K,J,I) = I * J * K                            11
      17 s1 130         AA(K,J,I) = DD(K,J,I)            EQUIVALENCE USED 11
      18    140     CALL SUB1(AA,CC)                                      0
            C
            C                                                            0
            C SUM REDUCTION                                              0
            C
            C                                                            0
      19          S4 = 0.0                                               0
      20 V1       DO 290 I = 1, 100                                      0
      21 v          S4 = A(I) + S4                  VECTOR SUM REDUCTION  0
      22 v  290   CONTINUE                                               0
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6 LINE: 8 OF 12
0008 * enddebug sample
0009 * PROGRAM HAS TERMINATED; RC ( 0)
0010 * annotate on message
```

AT

Tasks:

Using Cursor-Sensitive Commands (page 20)
Referring to Statements or Variables in Other Program Units (page 54)
Setting Breakpoints at Debugging Hooks (page 56)
Using Command Lists (page 60)

Function: AT sets breakpoints in a VS FORTRAN program at executable statements, entry points, or exit points. When defining a breakpoint, you may specify a list of Interactive Debug commands to be run whenever the breakpoint is reached. Execution will be suspended *before* the specified statement is run.

Abbreviation: None

Syntax

```
AT
     { [qual.]{number [:[qual.]number] | ENTRY
     | EXIT} | (number/ENTRY/EXIT list)}
     [(command-list)]
     [COUNT(n)]
     [NOTIFY | NONOTIFY]
```

qual
specifies a program unit name prefix to temporarily override the current qualifier for the prefixed operand only.

number
specifies the statement label, ISN, or sequence number of an executable VS FORTRAN statement. Precede a statement label with a slash to distinguish it from an ISN or sequence number. The type of statement number used (ISN or sequence number) is determined by the compiler and the options used. See "Statement Identifier Conventions" on page 101.

number:[*qual.*]*number*
specifies a range of statement labels and/or statement numbers (either ISNs or sequence numbers). A breakpoint is set at each executable statement within the range. Statement labels can be combined with ISNs or sequence numbers in the range, but the first and last must both be executable statements. Precede each statement label with a slash.

Statement identifiers can be qualified with a program unit name. The default program unit for the first identifier is the current qualifier. The default program unit for the second identifier in a range is the program unit specified or defaulted for the first identifier. Both identifiers must have the same program unit in effect.

For example, `A.5:6` is the same as `A.5:A.6`. However, `A.5:B.6` is invalid.

ENTRY
specifies that an entry breakpoint is to be set. The breakpoint occurs in the prolog of the program unit. The subroutine or function is not yet active, so dummy arguments are not accessible, and a GO with a statement identifier cannot be issued. The breakpoint occurs regardless of which entry point of the program unit is entered.

EXIT
specifies that an exit breakpoint is to be set. The breakpoint occurs after the last FORTRAN statement in the program unit has been executed.

(*number*/**ENTRY**/**EXIT** *list*)
specifies a list of statement labels, ISNs, sequence numbers, entry breakpoints, exit breakpoints, and/or ranges of numbers. A breakpoint is set at each executable statement specified. Enclose the list in parentheses, and separate entries with commas or blanks. Precede each statement label with a slash.

Note: If a statement that does not have a debugging hook is specified in the number list, an error message is issued but breakpoints are still set at the remaining statements.

(*command-list*)
specifies a list of commands to be run at the specified statement. If more than one statement is listed, the command list is run at every statement listed. Enclose the command list in parentheses, and separate commands with percent signs (%).

The qualification in effect when AT is issued is used as the default for any variables and statements referenced in the command list.

COUNT(*n***)**
specifies that a breakpoint occurs only every *n*th time the specified statement is reached. Specify *n* as an integer. If more than one statement is specified, the iteration count is applied independently to each listed statement.

NOTIFY | NONOTIFY
NOTIFY specifies that the location of every breakpoint is to be displayed when it is reached. This includes breakpoints that cause the program to resume without user intervention.

NONOTIFY specifies that no notification is given when the AT command list contains a command, such as GO, that causes execution to resume.

NOTIFY and NONOTIFY have no effect on the notification that is done when a breakpoint is reached and execution is suspended. You cannot turn off notification when execution is suspended.

Usage Notes

1. In full screen mode, AT can be used as a cursor-sensitive command. Place the cursor in the prefix area of the source window, and press the PF key assigned to AT. You can type over multiple ISN or sequence numbers with the AT command before pressing ENTER.

   Instead of assigning a PF key to the AT command, you can type the AT command over the statement number, or you can type AT on the command line and move the cursor to the target statement number before pressing ENTER.

2. Unless you have attached a command list containing GO, ENDDEBUG, STEP, RESTART, or QUIT, execution is always suspended when a breakpoint is reached.
3. The following commands cannot be used in a command list: AUTLOLIST, COLOR, DOWN, FIXUP, HELP, LEFT, LISTINGS, LISTSAMP, MOVECURS, POSITION, PREVDISP, PROFILE, REFRESH, RESTART, RESTORE, RETRIEVE, RIGHT, SEARCH, SIZE, UP, WINDOW, ZOOM.
4. An AT statement specified for a statement that already contains a breakpoint will.replace the old breakpoint with a new breakpoint.
5. HALT, NEXT, WHEN conditions, and AT breakpoints all cause execution to be suspended. When execution is suspended for multiple reasons, messages are issued for all the reasons.
6. At ENTRY, the following restrictions apply:
   - GO with a statement ID is not permitted.
   - Variables in a dynamic common cannot be referenced.
   - Dummy arguments are not accessible.
7. You can display a comment with a breakpoint by using LIST (page 141) to display a quoted string as part of the command list for the breakpoint.
8. You can set a breakpoint only on statements that have a *debugging hook*. You cannot set a breakpoint on statements outside the AFFON statement list, on statements that are collapsed, on the ENTRY of a VS FORTRAN main program, or on the EXIT of a VS FORTRAN main program. For more information on debugging hooks, see "Setting Breakpoints at Debugging Hooks" on page 56.
9. A command in an AT command list that causes execution to resume will cause the remainder of the command list to be ignored. These commands are GO, STEP, ENDDEBUG, and RESTART. QUIT and HALT IMMED also cause the remainder of the command list to be ignored.
10. AT is not permitted after the VS FORTRAN program has terminated.

Examples

1. Stop execution every 10th time the program reaches the beginning of a loop that begins at statement label 65.

```
at /65 count(10)
```

2. Set breakpoints at ISNs 180 and 220 in the currently qualified program unit, at every executable statement in program unit SUB1 between ISN 10 and statement label 50, and at entry to program unit CHECK. Execution is suspended at each of these points.

```
at (180 220 sub1.10:/50 check.entry)
```

3. At ISN 140, list the value of variable A, set variable I to 10, and continue execution. Except for listing the value of A, no notification is given.

```
at 140 (list a%set i=10%go) nonotify
```

4. At ISN 5 in program unit STUB, set variable ANSWER to 100 and exit the subroutine.

```
at stub.5 (set stub.answer=100%go stub.exit)
```

AUTOLIST (full screen mode only)

Function: AUTOLIST is used to define a group of variables and array elements for which values are to be displayed in the monitor window of the main debugging panel. The values are continuously displayed during execution.

The containing program unit name is shown with all variable or array names. Array elements may be displayed outside the defined dimensions. If AUTOLIST is specified with no operands, the monitor window will become empty.

Abbreviation: AL

Syntax

AUTOLIST
[{[*qual.***]*name*[:[*qual.*]*name*] | * | '*string*' | *number* | (*list*)}**
[FORMAT [(*code*)] | DUMP [(*code*)]]]

[*qual.***]*name***
specifies the name of a variable, array, or array element used in the program. If *qual.* is specified, it overrides the current qualifier for the specified name. Substring notation may be used with string variables.

[*qual.***]*name*:[*qual.*]*name***
specifies a range of variable, array, or array element names used in the program. If *qual.* is specified, it overrides the current qualifier for the specified name.

AUTOLIST displays all storage locations between the two variables. Unless FORMAT or DUMP is specified, the format of the displayed storage is the same as the type of the first variable.

*
specifies that a list of all the variables and arrays in the currently qualified program unit is desired. Unless FORMAT or DUMP is specified, each is displayed according to its own type.

'string'
specifies a character string to be displayed as a remark. You can use this operand to help identify the AUTOLIST display.

number
specifies an integer or real numeric constant to be displayed as a remark. This function can be useful in converting numbers, when used in conjunction with the FORMAT option.

list
identifies a list of individual specifications. The list must be enclosed in parentheses. Individual entries must be separated by commas or blanks.

FORMAT [(*code*)] | DUMP [(*code*)]
specifies a particular data format:

- FORMAT displays the names listed and their values in the specified format.

- DUMP displays the address in storage of the names listed and their values in the specified format.
- *code* specifies the format or dump code for the names to be listed. The default format code is X. The default dump code is Z. FORMAT and DUMP codes for the AUTOLIST command are shown in Figure 38.
- FORMAT and DUMP are mutually exclusive.

Code	Output
L1	Logical*1
L4	Logical*4
I2	Integer*2
I4	Integer*4
R4	Real*4
R8	Real*8
R16	Real*16
C8	Complex*8
C16	Complex*16
C32	Complex*32
L	Logical with size closest to internal data size
I	Integer with size closest to internal data size
R	Real with size closest to internal data size
C	Complex with size closest to internal data size
X[nnn]	Hexadecimal with nnn bytes per data item (default to internal data size)
Z[nnn]	Hexadecimal with nnn bytes per data item (default to Z4)
A[nnn]	Character with nnn bytes per data item (default to internal data size)
H[nnn]	Character with nnn bytes per data item (default to continuous full line output)

Figure 38. DUMP and FORMAT Codes for the AUTOLIST Command

Usage Notes

1. Each time you issue AUTOLIST, the set of variables to be listed is redefined. The lists are not cumulative.

   For example, if you type `AL (X,Y)`, then X and Y are displayed. If you later type `AL Z`, only Z will be displayed.

2. The output is displayed in the monitor window. No AUTOLIST output is shown in the log window. If more than 1000 lines of output are requested, the excess is not displayed.

3. Dummy arguments can only be displayed when the program unit in which they are defined is active. For example, when executing in MAIN before calling SUB1, entering:

   ```
   autolist subl.a
   ```

   will cause a message to be produced if a is a parameter. Results are unpredictable if you display a dummy argument that is not defined at the entry point called. (Note that a program unit is not yet active when suspended at entry.)

4. Variables in dynamic commons can only be displayed if the program unit used to qualify the variable has been activated at least once. (If not, you may receive an error message. However, if a variable has a large displacement in its dynamic common, Interactive Debug cannot detect that it is not initialized.)

5. When you request an individual name or list of names, the default format is determined by the type of each variable being displayed. When you request a range of names, the formatting of the values is determined by the format of the first name in the range. The locations of the listed names are not identified in the output. You may, however, specify a different format using FORMAT or DUMP.

6. VS FORTRAN defines storage layout only within arrays, variables in a common block (defined in a COMMON statement), and variables in equivalence groups (defined in an EQUIVALENCE statement). The relative positions of any other names in storage cannot be predicted. Names that you may expect to be adjacent in storage may be widely separated by other data. Therefore, a range specification for names other than array, equivalence, or common variables may produce unexpected results.

7. The DUMP option is not permitted with constant operands, including strings.

8. The length specification in a FORMAT or DUMP code may be entered with 1 through 3 digits. Thus, I4, I04, and I004 are equivalent.

9. A length specification of 0 in character and hexadecimal FORMAT and DUMP codes (for example, A0 or Z0) causes the data to be displayed as a continuous string, rather than split into pieces of some specified length.

10. If a FORMAT or DUMP code with no length specification is given for a range of variables or array elements, each variable or array element is displayed separately in the specified format. However, if a length specification is given, Interactive Debug will consider the entire storage area occupied by all the range of variables or array elements, or occupied by the entire array, as if it were broken into pieces, each with a length equal to that specified in the DUMP or FORMAT code, and will display each piece according to the specified format. For example, if PRIMES is a 2 x 3 array of INTEGER*4 values, then:

    ```
    autolist primes format(x)
    ```

    will cause a display of 6 values, each corresponding to an element of the array. However:

    ```
    autolist primes format(x2)
    ```

    will cause a display of 12 values, each displaying the contents of successive 2-byte storage areas within the array.

11. An assumed size array cannot be listed by just specifying the array name; the specific element or range of elements must be specified. (An assumed size array is an array with the last upper bound specified as an asterisk (*).) This restriction does not apply to arrays whose last dimension is "1" even though such arrays are otherwise treated as assumed size arrays. However, only the elements whose last subscript is "1" will be displayed if no subscripts are specified.

12. For array elements, subscripts may have values beyond the range of the corresponding array dimensions. A warning message will be issued except in the special case where the last dimension is "1" or "*" and only the last subscript is out of range.

13. Array subscripts may contain simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid forms:

```
autolist ARY(I)        autolist ARY(3)
autolist ARY(I + 3)    autolist ARY(I - 3)
```

Examples

1. Continuously display the value of variable NCOUNT.

```
autolist ncount
```

2. Continuously display a hexadecimal dump (FORMAT(X)) showing values of array variables A(1,1) through A(2,3).

```
autolist a(1,1):a(2,3) format
```

3. Continuously display several variables in hexadecimal, each with as many bytes as are appropriate for its data type.

```
autolist (i,j,p,q,r) format
```

4. Continuously display an entire array (CHARAY) containing a series of 30-character alphabetic strings so that each character string is separated from the others. (If the array is declared in the program to be a CHARACTER*30 array, then the elements of the array will be separated from each other when the array is listed.)

```
autolist charay format(a30)
```

5. Continuously display the value of the variable REAL1 in SUB1, ARRAY(I,J) in SUB2, and STRING, I, and J in the currently qualified program unit.

```
autolist (sub1.real1,sub2.array(i,j),string,i,j)
```

BACKSPACE

Tasks: Processing External Files (page 76)

Function: BACKSPACE positions a sequentially accessed external file to the beginning of the previous record. It is similar to the BACKSPACE statement in the VS FORTRAN Version 2 language. This command allows you to move backward in the file, for example to rewrite or reread a record.

Abbreviation: BACKSPAC, BACKS

Syntax

BACKSPACE
{*number* | [*qual.*]*integer-variable* | [*qual.*]*integer-array-element*}

number
is the number of the I/O unit associated with the sequential file on which the backspace is to be performed.

[*qual.*]*integer-variable*
is the name of an integer variable in the VS FORTRAN program. This variable specifies the number of the I/O unit associated with the sequential file on which the backspace is to be performed.

[*qual.*]*integer-array-element*
is the name of an element of an integer array in the VS FORTRAN program. This element specifies the number of the I/O unit associated with the sequential file on which the backspace is to be performed.

Usage Notes

1. *number*, *integer-variable*, or *integer-array-element* must be specified; there is no default number.
2. This command may not be issued when I/O is currently active.
3. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
A(I), A(3), ARY(I+3), ARY(I-3)
```

Example

Backspace the sequentially accessed external file associated with I/O unit 8 so that the last record written can be rewritten.

```
backspace 8
```

CLOSE

Tasks: Processing External Files (page 76)

Function: CLOSE disconnects a VS FORTRAN external file from an input or output unit. Its usage is similar to that of the CLOSE statement in the VS FORTRAN Version 2 language. This command allows you to close an external file, for example to assign another file to the input or output unit, or to examine the contents of the file.

Abbreviation: None

Syntax

```
CLOSE
     {number | [qual.]integer-variable | [qual.]integer-array-element}
```

number
is the number of the I/O unit associated with the file that is to be closed.

[qual.]integer-variable
is the name of an integer-variable in the VS FORTRAN program. This variable specifies the number of the I/O unit associated with the file that is to be closed.

[qual.]integer-array-element
is the name of an element of an integer array in the VS FORTRAN program. This element specifies the number of the I/O unit associated with the file that is to be closed.

Usage Notes

1. *Number*, *integer-variable*, or *integer-array-element* must be specified; there is no default number.
2. This command may not be issued when I/O is currently active.
3. Files used in the program but not explicitly closed will still be open when Interactive Debug gives you control at termination. If you want to examine such a file, you must CLOSE it first.
4. Under certain conditions, use of the CLOSE command may make it necessary for you to use the RECONNECT command (page 175) before your program can perform additional I/O operations on the file. This situation occurs when the OCSTATUS run-time option is in effect, and the program cannot be made to run an OPEN statement before doing more I/O to the file.
5. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
A(I), A(3), ARY(I+3), ARY(I-3)
```

Examples

1. In CMS, reallocate an external output file for I/O unit 8.

```
close 8
sys filedef 8 disk output file a
```

2. In TSO, reallocate an external output file for I/O unit 8.

```
close 8
sys allocate file(ft08f001) dataset(output.file)
```

COLOR (full screen mode only)

Tasks: Changing the Display Color (page 22)

Function: COLOR allows you to customize the color, highlighting and intensity of the various parts of the main debugging panel.

Abbreviation: None

Syntax

```
COLOR
```

The color panel is shown in Figure 39.

```
        VS FORTRAN INTERACTIVE DEBUG COLOR PANEL
COMMAND ===>
                          COLOR   HIGHLIGHT  INTENSITY
TITLE  : FIELD HEADERS    WHITE    NONE       HIGH      ENTER:
         OUTPUT FIELDS    BLUE     NONE       LOW         SAVE    TO SAVE YOUR
MONITOR: CONTENTS         TURQ     REVERSE    LOW         COLOR SETTINGS.
         LINE NUMBERS     TURQ     REVERSE    HIGH        RESET  TO RESTORE
SOURCE : LISTING AREA     WHITE    REVERSE    LOW         PROFILE SETTINGS.
         PREFIX AREA      TURQ     REVERSE    LOW         END TO RETURN TO
         SUFFIX AREA      BLUE     REVERSE    LOW         DEBUG SESSION WITH
         CURRENT LINE     YELLOW   REVERSE    HIGH        CURRENT SETTINGS IN
         BREAKPOINT       RED      REVERSE    HIGH        EFFECT.
         BAR GRAPHS       GREEN    REVERSE    HIGH
LOG    : OUTPUT LINES     GREEN    NONE       LOW
         INPUT LINES      YELLOW   NONE       HIGH
         LINE NUMBERS     TURQ     NONE       HIGH
 COMMAND LINE             YELLOW   NONE       LOW       COLOR and HIGHLIGHT
 AREA HEADERS             GREEN    REVERSE    HIGH      ARE VALID ONLY WITH
 TOF & EOF DELIMITER      BLUE     REVERSE    LOW       COLOR TERMINALS.
 SEARCH TARGET            WHITE    NONE       HIGH

 VALID COLOR     :  WHITE YELLOW BLUE TURQ GREEN PINK RED
 VALID INTENSITY :  HIGH LOW
 VALID HIGHLIGHT :  USCORE BLINK REVERSE NONE
```

Figure 39. Interactive Debug Color Panel

The following commands can be entered on the command line:

SAVE: saves current color settings in the ISPF profile for future debugging sessions.

RESET: restores color settings to the profile settings.

END: returns to main debugging panel with current settings in effect. (Unless a SAVE has been issued, these settings will remain in effect for only the duration of the debugging session.)

Usage Notes

1. COLOR cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. COLOR will operate as usual with a parameter list, but the panel will include the message "PARAMETERS IGNORED."

DBCS

Function: DBCS specifies whether X'0E' and X'0F' are interpreted as the DBCS shift-out and shift-in characters in input and output.

Abbreviation: None

Syntax

```
DBCS
     [YES | NO]
```

YES
specifies that X'0E' and X'0F' are interpreted as DBCS shift codes.

NO
specifies that X'0E' and X'0F' are not interpreted as DBCS shift codes. NO is the initial setting.

Usage Note

To query the current status of DBCS, specify DBCS with no operands.

Example

Query DBCS status.

```
dbcs
```

DESCRIBE

Tasks:

Using Cursor-Sensitive Commands (page 20)
Specifying Output to a Print File (page 42)
Displaying Data Types of Variables and Arrays (page 61)

Function: DESCRIBE displays the data type of variables or arrays, and also supplies dimension information for arrays.

Abbreviation: DE

Syntax

```
DESCRIBE
    {[qual.]name | * | (name-list)}
    [PRINT]
```

[qual.]name
specifies the name of a variable, or array used in the program. The name can be qualified by the name of a program unit.

*
specifies a list of all the names in the currently qualified program unit. The type of each is displayed.

(name-list)
specifies a list of names. Enclose the list in parentheses and separate individual names with commas or blanks.

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. In full screen mode, DESCRIBE can be used as a cursor-sensitive command. If the DESCRIBE command is already assigned to a PF key, place the cursor at a variable name in the source window, and press the PF key for DESCRIBE. The variable may include either subscript or substring notation. If both are present, only the subscript will be included in the command.
2. In your output, dummy arguments are identified in the last column with the word DUMMY.
3. Interactive Debug cannot determine the length of character variables that are dummy arguments in an inactive program unit. The length is displayed as CHARACTER*(*). (Note that at ENTRY, the program unit is not yet active.)
4. Dimension information is not displayed for arrays that are dummy arguments in an inactive program unit, or that are defined only when entered by some other entry point. (Note that at ENTRY, the program unit is not yet active.)

5. The upper bound for the last dimension of an assumed-size array is displayed as an asterisk, and the size is indicated as * ELEMENTS.

Example

Display the data type information for the variables I and R4, for array CM8ARY, for the dummy array R8ASAR, for the dummy variable WHERE in program unit SUB1, and for dummy array L1AYMN in program unit SUB1.

```
describe (i,r4,cm8ary,r8asar,sub1.where,sub1.l1aymn)
```

The output from this command should look something like this:

```
SUB2.I:                          INTEGER*4
SUB2.R4:                         REAL*4
SUB2.CH8ARY:                     COMPLEX*8
   RANK = 2, SIZE = 21 ELEMENTS
   DIM 1:    EXTENT = 3,  LBOUND = (1),  UBOUND = (3)
   DIM 2:    EXTENT = 7,  LBOUND = (1),  UBOUND = (7)
SUB2.R8ASAR:                     REAL*8             DUMMY
   RANK = 1, SIZE = * ELEMENTS
   DIM 1:    EXTENT = *,  LBOUND = (-3),  UBOUND = (*)
SUB1.WHERE:                      CHARACTER*(*)      DUMMY
SUB1.L1AYMN:                     LOGICAL*1          DUMMY
   RANK = 1; DUMMY ARRAY ARGUMENT OF INACTIVE SUBPROGRAM OR
             ALTERNATE ENTRY POINT
             DIMENSION INFORMATION NOT AVAILABLE
```

DOWN (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Using Cursor-Sensitive Commands (page 20)

Function: DOWN scrolls the contents of a window so that lines below those currently displayed in the window can be seen.

Abbreviation: None

Syntax

DOWN
[*number* **| PAGE | HALF | CSR | DATA | MAX]**

number
is the number of lines to scroll down, from 1 to 9999.

PAGE (or P)
scrolls down by the number of lines in the window.

HALF (or H)
scrolls down by half the number of lines in the window.

CSR (or C)
scrolls down by PAGE, unless the cursor is in the window, in which case the window is scrolled down by the appropriate number of lines to place the cursor at the edge of the window.

DATA (or D)
scrolls down by PAGE-1 number of lines. If only one line is visible, the scrolling is equivalent to that of PAGE.

MAX (or M)
scrolls down so that the last line of the window will contain the bottom-of-data marker (see page 15).

Usage Notes

1. DOWN is cursor-sensitive. The window that is scrolled is determined from the cursor position and the windows currently open.
2. DOWN cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
3. If an operand is not specified with DOWN, the scrolling amount is taken from the SCROLL ===> field on the main debugging panel.

Example

Scroll down ten lines in the listing displayed in the source window, assuming that the cursor is on the command line and the source window is open.

```
down 10
```

ENDDEBUG

Tasks:

Program Sampling (page 63)
Continuing Execution without Further Debugging (page 80)

Function: ENDDEBUG allows you to:

1. Discontinue debugging and run the program at full speed. Except for entering limited commands after the program has terminated, no debugging is available after using the ENDDEBUG command.
2. Initiate program sampling to obtain an approximation of relative CPU time, using the SAMPLE option.

Abbreviation: None

Syntax

```
ENDDEBUG
    [SAMPLE[(msecs)] [MAXSAMP(n[,STOP])] [CALLED ]]
```

SAMPLE [(*msecs*)]
indicates that sampling is to occur during subsequent execution. *msecs* is the time interval in milliseconds between sampling interruptions; the default is 10 milliseconds.

MAXSAMP(*n*[,STOP])
specifies the maximum number of sampling interruptions (*n*) that can occur. If STOP is specified, the program will be terminated when the specified number of samples is reached. Otherwise, the program will continue execution without sampling interruptions after the specified number of samples is reached. If MAXSAMP is not specified, sampling interruptions will continue until the program terminates. MAXSAMP is valid only if SAMPLE is specified.

CALLED
specifies that each sampling interruption is to be counted for each caller in the save chain. This option may cause an appreciable increase in overhead for the sampling function; however, it makes it much easier to determine when CPU usage is primarily due to called subroutines. CALLED is valid only if SAMPLE is specified.

If CALLED is not specified in the ENDDEBUG command, the CALLED option will not be permitted with the ANNOTATE and LISTSAMP commands.

Usage Notes

1. All breakpoints and WHEN conditions are removed and the HALT command status is set to OFF.
2. TERMIO is set to LIBRARY.
3. Timing is turned off for all program units.
4. The VS FORTRAN Library will begin updating the occurrence count for run-time errors. All error handling actions will be determined by the settings in the VS FORTRAN error option table.
5. The attention exit will no longer allow entry of Interactive Debug commands after ENDDEBUG is called. The only Interactive Debug commands allowed from an attention exit are PURGE and QUIT, which will terminate the VS FORTRAN program. You will then be prompted for further Interactive Debug commands. Entering QUIT a second time will terminate the debugging session. To resume execution following an attention interrupt, enter a null line.
6. At the end of program execution, Interactive Debug will prompt for commands. LISTTIME and LISTFREQ will show information that was current when the ENDDEBUG command was issued. WHERE information cannot be determined after ENDDEBUG is issued.
7. Unless the program was compiled with the TEST option, or SAMPLE was specified, execution will proceed at the same speed at which it would run if the DEBUG run-time option had not been specified.

   If a program unit was compiled with the TEST option, Interactive Debug will still be called for each VS FORTRAN statement in that program unit. While this activity will not be apparent, there will be a slight increase in the time required to run your program.
8. If SAMPLE was specified, program speed will be reduced due to the overhead involved in recording the sampling information at each interrupt. Sampling will cancel any active timer interval at the start of execution (one millisecond per timer interruption).
9. If SAMPLE is specified without CALLED, the ANNOTATE settings (for the source window) are changed to ON. If SAMPLE and CALLED are both specified, the settings are changed to ON and CALLED.
10. ENDDEBUG is not permitted after the VS FORTRAN program has terminated or while a read is pending. If issued in an error exit, standard corrective action is taken.
11. The BLIP is turned off when sampling is in operation and restored when sampling completes.
12. The service routine CPUTIME cannot be used while sampling is in progress. For more information on CPUTIME, see *VS FORTRAN Version 2 Language and Library Reference*.

Examples

1. Continue running a program from the current statement to the end of the program with no further debugging activity.

```
enddebug
```

2. End debugging, but continue running the program with sampling interruptions every 10 milliseconds.

```
enddebug sample
```

3. End debugging, but continue running the program with sampling interruptions every 20 milliseconds and CALLED counts accumulated.

```
enddebug sample(20) called
```

4. End debugging, but continue running the program with sampling interruptions occurring every 40 milliseconds. Continue running the program without interruptions if the number of interruptions exceeds 10,000.

```
enddebug sample(40) maxsamp(10000)
```

ENDFILE

Tasks: Processing External Files (page 76)

Function: ENDFILE writes an end-of-file record on a sequentially accessed external file. Its usage is similar to that of the ENDFILE statement in the VS FORTRAN Version 2 language.

Abbreviation: ENDF

Syntax

ENDFILE
{number | [qual.]integer-variable | [qual.]integer-array-element}

number
is the number of the I/O unit associated with the sequential file on which the end-of-file record is to be written.

[qual.]integer-variable
is the name of an integer-variable in the VS FORTRAN program. This variable specifies the number of the I/O unit associated with the sequential file on which the end-of-file record is to be written.

[qual.]integer-array-element
is the name of an element of an integer array in the VS FORTRAN program. This element specifies the number of the I/O unit associated with the sequential file on which the end-of-file record is to be written.

Usage Notes

1. *number*, *integer-variable*, or *integer-array-element* must be specified; there is no default number.
2. This command may not be issued when I/O is currently active.
3. Writing an end-of-file record erases all records that may follow.
4. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
ARY(I), ARY(3), ARY(I+3), ARY(I-3)
```

Example

Write an end-of-file record on the sequentially accessed external file associated with I/O unit 8.

```
endfile 8
```

ERROR

Tasks: Handling Run-Time Errors (page 74)

Function: ERROR specifies whether corrective action is to be performed or execution is to be suspended, and whether messages are to be received for run-time errors. If execution is suspended, you may specify the corrective action to be taken by issuing the FIXUP command. For more information on library error messages, see *VS FORTRAN Version 2 Language and Library Reference.*

Abbreviation: ER

Syntax

```
ERROR
     {error | error:error | (error-list)}
     [MSG|NOMSG]
     [EXIT|NOEXIT]
```

error
specifies the identification number of a single error as defined by the VS FORTRAN Version 2 or VS FORTRAN Version 1 Library, or as defined by an auxiliary product.

error:error
specifies a range of error identification numbers to which all specified keywords apply.

(error-list)
specifies a list of error numbers or ranges to which all specified keywords apply. Enclose the list in parentheses, and separate entries with commas or blanks.

MSG | NOMSG
specifies whether or not a diagnostic message is to be displayed if a run-time error occurs among those specified.

EXIT | NOEXIT
EXIT specifies that program execution is to be suspended if any of the listed errors occur.

NOEXIT specifies that the VS FORTRAN library is to take corrective action and execution is to continue if any of the listed errors occur.

Usage Notes

1. The default options at the start of a debugging session are to provide all diagnostic messages (MSG) and to suspend execution of the program if any run-time error occurs (EXIT). If your program calls ERRSET, this will change the settings for the specified errors to MSG and NOEXIT.
2. If the EXIT keyword is in effect for a particular error, you are notified of the location and error number when the error occurs. You can then trace the sequence of control that led to the error location by issuing the WHERE command with the TRBACK or FLOW keyword. If NOEXIT and NOMSG are both in effect, no notification of error location or error number is given. If MSG is in effect, the error number will be contained in the error message.
3. If you specify a large range of error numbers, the ERROR command may generate a large number of diagnostic messages. You can use the PURGE command in an attention exit to terminate ERROR if the messages are excessive.
4. If NOEXIT has been specified, standard corrective action will be taken unless you have defined user corrective action by calling ERRSET from your VS FORTRAN program. Execution will not be suspended in either case.
5. Not all VS FORTRAN library error numbers may be specified by the ERROR command. If you specify one that may not, you will receive the following message:

   ```
   AFB198I VIADI : ATTEMPT TO CHANGE UNMODIFIABLE MESSAGE TABLE ENTRY,
   MESSAGE NUMBER 240
   ```

6. ERROR is not permitted after the VS FORTRAN program has terminated.

Examples

1. You have set a variable to a negative value to test a condition but then realize that the square root of the variable will be taken later. To avoid halting execution when this occurs, you want the library to perform standard corrective action and continue with no notification of the error. The error number for the square root of a negative number is 251.

   ```
   error 251 nomsg noexit
   ```

2. If any single or double precision arithmetic run-time errors (logs, trigonometric functions, exponents) occur, you want Interactive Debug to provide full diagnostics and to also take standard corrective action. The error numbers are 241 through 285.

   ```
   error 241:285 noexit
   ```

FIXUP

Tasks: Handling Run-Time Errors (page 74)

Function: FIXUP specifies corrected argument values when execution has been suspended because of errors in a VS FORTRAN Library mathematical function. This command causes the function to be evaluated with new arguments and execution to be continued. A FIXUP command with no arguments causes standard corrective action to be taken.

Abbreviation: F

Syntax

```
FIXUP
     [ARG1(value)]
     [ARG2(value)]
```

ARG1(*value*)
specifies the value of the first argument of the function. The value can be a variable, an array element, or a constant.

ARG2(*value*)
specifies the value of the second argument of the function. The value can be a variable, an array element, or a constant.

Usage Notes

1. FIXUP is permitted only when execution is suspended for a library function error.
2. If the library function for which FIXUP is to be performed has two arguments, you can specify a single value for either of the two arguments, or you can specify values for both arguments.
3. If you issue GO or FIXUP without arguments after an run-time error, standard corrective action is performed and execution is resumed.
4. You cannot change the actual value of the variable in storage by using FIXUP.
5. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

   ```
   A(I), A(3), ARY(I+3), ARY(I-3)
   ```

Examples

1. Your program stops because of an arithmetic error. You want the library to perform standard corrective action and to resume execution.

   ```
   fixup
   ```

2. Your program attempts to take the square root of a variable that is set to a negative value. Instead of taking standard corrective action, you reset the negative variable to a positive number and resume execution. (The value of the variable in storage is not changed.)

   ```
   fixup arg1(36)
   ```

GO

Tasks:

Animating the Execution of Your Program (page 73)
Processing External Files (page 76)

Function: GO resumes program execution.

Abbreviation: None

Syntax

```
GO
    [[qual.]{number | EXIT}]
```

qual.
specifies a program unit name to temporarily override the current qualifier for this command. The qualifier must be the program unit that will be in execution when the GO is run.

number
specifies the statement identifier of an executable VS FORTRAN statement at which execution is to be resumed. The statement identifier is a statement label, an ISN, or a sequence number in columns 73 through 80. The type of statement number used (ISN or sequence number) is determined by the compiler and the options used. (See "Statement Identifier Conventions" on page 101.) Precede a statement label with a slash. The statement must be an executable statement with a debugging hook.

EXIT
specifies that Interactive Debug will resume execution of the program unit at the exit point that corresponds to the entry point used.

Usage Notes

1. The statement identifier or EXIT keyword is optional, and is used if execution is to resume at a specific point outside of the normal execution sequence. GO without an operand causes execution to be resumed at the currently suspended statement.
2. If execution has stopped because of an run-time error, resuming it with GO produces standard corrective action to fix the error. (GO with an operand is not allowed in an error exit.)
3. Execution of the GO command is subject to the same language restrictions as the GOTO statement in VS FORTRAN. Branches into DO-loops and inner DO-loops (except from the extended range of the DO loop (LANGLVL(66) only)) will produce unpredictable results.
4. The number or EXIT operand is not allowed with the GO command:
   - In an error exit
   - When execution is suspended at the ENTRY point of a program unit
   - When execution is suspended for output
   - When execution is suspended during a terminal read

5. It is not advisable to specify a statement label or statement identifier when running in a program unit that has been compiled with an optimization level higher than zero. (This is because optimized code is highly dependent on register contents.) Interactive Debug allows the operand, but will warn you (except in batch mode) that the program unit is optimized and ask if you wish to continue. If you type YES, the GO will be run.
6. GO is not permitted after the VS FORTRAN program has terminated. It is also not permitted if execution is suspended during a terminal read.

Examples

1. Your program suspends execution at a breakpoint for debugging, and you now want to resume execution at the currently suspended statement.

   ```
   go
   ```

2. Your program suspends execution at a breakpoint for debugging, and you now want to skip ahead and resume execution at sequence number 410.

   ```
   go 410
   ```

3. Your program suspends execution at a breakpoint for debugging, and you have changed the qualification to display variables in another program unit. You now want to skip ahead and resume execution at the exit point of the program unit currently executing, which is SUB1.

   ```
   go sub1.exit
   ```

4. You know that program unit BUGGY produces incorrect results past the statement labeled 100. For now, you want to set RESULT to the value A, and exit whenever statement 100 is reached.

   ```
   at buggy./100 (set result = a & go buggy.exit) nonotify
   ```

HALT

Tasks:

Animating the Execution of Your Program (page 73)
Controlling Program Execution (page 58)

Function: HALT causes execution to be suspended for every statement of a given class, or at a specific point in a command list. The classes of statements are: at the start of every statement, or after every branch, or at entry to and exit from a debuggable routine.

Abbreviation: None

Syntax

```
HALT
     [OFF |STMT | GOTO | ENTRY |IMMED]
```

OFF
specifies that the HALT setting in effect is to be terminated. This is the setting when Interactive Debug execution begins.

STMT
indicates that execution is to be suspended before every executable statement that has a debugging hook.

GOTO
indicates that execution is to be suspended whenever two consecutively executed debugging hooks are not on consecutively stored statements. This could occur for several reasons, including a GOTO, a DO group, a CALL, or an IF statement. It also might occur if one or more statements have been collapsed by optimization or vectorization, or because of hook restrictions specified for ranges in AFFON, or when debug packets occur in the code.

ENTRY
specifies that execution is to be suspended whenever any debuggable program unit is entered or exited. This could be as a result of a subroutine or function call or return from a subroutine or function.

HALT ENTRY causes suspension of execution at the same points as AT EXIT or AT ENTRY. At an entry breakpoint, the program unit is not yet active. Thus, dummy arguments or variables in a dynamic common cannot be accessed, and GO with an operand is not permitted.

IMMED
indicates that execution of an AT command list should be suspended immediately and a prompt should be issued. This is the default action when HALT is issued with no operand. In an attention exit, this will terminate the current AT command list if one is running.

Usage Notes

1. In full screen mode, HALT can be used as a cursor-sensitive command.
2. You can issue a HALT IMMED command to terminate a command list. You can also use HALT IMMED in an attention exit to terminate command loops. For example: `AT 5 (GO 5)`.
3. If a statement at which execution would normally be suspended by the current HALT setting has an AT breakpoint with a command list that causes execution to resume, the HALT setting will not suspend that statement.

   For example, if HALT STMT is in effect but you have issued the command `AT 5 (SET A=10%GO)`, execution will not be suspended at statement 5.
4. Any statement for which execution would be suspended by HALT GOTO will also have execution suspended by HALT STMT.

   Any statement for which execution would be suspended by HALT ENTRY will also have execution suspended by HALT GOTO and HALT STMT.
5. HALT is not permitted after the VS FORTRAN program has terminated.
6. To see the current HALT setting, type `LISTBRKS`.

Examples

1. Suspend execution at entry to or exit from all debuggable units.

   ```
   halt entry
   ```
2. Suspend execution prior to executing each statement with a debugging hook in the program.

   ```
   halt stmt
   ```
3. At statement identifier 10, halt execution if A is greater than B; otherwise, continue executing with no notification of the breakpoint.

   ```
   at 10 (if (a .gt. b) halt%go) nonotify
   ```

HELP

Tasks:

Using Cursor-Sensitive Commands (page 20)
Getting On-line Help about Interactive Debug (page 51)

HELP provides on-line information about Interactive Debug commands, common tasks, and vector messages contained in the vector report source listing, as well as a task-oriented tutorial.

Abbreviation: H

Syntax in CMS or TSO Full Screen Mode

HELP
[*command* **|** *vecmsg-id***]**

Syntax in CMS Line Mode

HELP
[*command* **[(ALL | (DESC | (PARM | (FORM] | *vecmsg-id*]**

Syntax In TSO Line Mode

HELP
[*command* **[ALL | FUNCTION | SYNTAX | OPERANDS [(***keyword-list***)]]**
| *vecmsg-id***]**

command
specifies the name of a command, or one of the keywords TASK or MENU (IADMENU in TSO line mode). If no command is specified, a HELP menu is displayed for further selection. You can specify a command abbreviation in full screen mode, but not in line mode.

ALL
requests the function, syntax, keywords, usage notes, and examples of the specified command.

DESC
requests the function of the specified command.

FORM
requests the syntax of the specified command.

PARM
requests the keywords, usage notes, and examples of the specified command.

FUNCTION
requests the function of the specified command.

SYNTAX
requests the syntax of the specified command.

OPERANDS [(*keyword-list*)]
requests the keywords of the specified command. Enclose the list in parentheses and separate the keywords in the list with commas or blanks. If the list is omitted, all keywords, usage notes, and examples of the specified command are displayed.

vecmsg-id
specifies the number of a vector message on the vector report source listing. It can be in either of two forms:

```
ILX01nnI
nn
```

where nn is the two-digit number that uniquely identifies the message.

Usage Notes

1. HELP cannot be issued in: an AT command list, an IF command, or a restart file.
2. In TSO line mode, you can request that usage notes and examples be shown by specifying NOTES or EXAMPLES as a keyword with OPERANDS.
3. In full screen mode, vector message information can be displayed by using the HELP command as a cursor-sensitive command:
   - If the source window displays a vector report source listing, the cursor is in the source window, and the user hits PF1 (normally HELP), the help for the vector report message on that line of the listing will be displayed.

     The cursor may be in the prefix area, the source listing area, or the prefix area of the source window corresponding to the line of the listing. If PF1 is not set to HELP, the user can type HELP on the command line, place the cursor on the desired line in the source window, and press ENTER for the same effect.
   - Placing the cursor on a line in the vector report source listing which does not have a vector report message results in the following message being displayed:

     ```
     NO VEC(REP) MESSAGE
     ```
   - If the cursor is outside the source window or if the listing being displayed is not a vector report source listing, then HELP will not function as a cursor-sensitive command.

Examples

1. Display a list of all available HELP topics.

   ```
   help
   ```
2. Display a list of all available HELP task topics.

   ```
   help task
   ```
3. Display all available information on the IF command.

   ```
   help if
   ```
4. In CMS line mode, display the syntax for the ERROR command.

   ```
   help error (form
   ```

5. In TSO line mode, display the syntax for the ERROR command.

```
help error syntax
```

6. In TSO line mode, display the syntax and the EXIT keyword for the ERROR command.

```
help error syntax operands(exit)
```

7. In TSO line mode, display the syntax, usage notes, and examples for the GO command.

```
help go syntax operands(notes, examples)
```

8. Display information on the vector message ILX0113I.

```
help 13
```

9. In full screen mode, display information for vector message ILX0109I on ISN 423 of the vector report listing for LBTAB:

   a. Qualify to program unit LBTAB by typing `q lbtab`

   b. Scroll down to ISN 423 in the source window by typing `position 423` on the command line, putting the cursor in the source window, and pressing ENTER.

   c. Place cursor on the statement at ISN 423 in the source window.

   d. Press the HELP PF key (normally PF1).

IF

Function: IF is used, usually within an AT command list, to test a relational or a logical condition when the specified breakpoint is reached. If the condition is true, the command specified within the IF command is run.

Abbreviation: None

Syntax

```
IF
    (condition) command
```

(condition)
is the condition to be tested. It can be either a relational or a logical condition:

- Relational condition: a signed or unsigned variable or array element or constant, followed by a *relational operator*, followed by another signed or unsigned variable or array element or constant. There are six relational operators that can be used to test relational conditions.

  = or .EQ.
  ¬ = or .NE.
  > or .GT.
  < or .LT.
  > = or .GE.
  < = or .LE.

- Logical condition: a logical variable or a logical array element, optionally preceded by the negation operator (¬ or .NOT.). No other operators are permitted.

command
is a single Interactive Debug command that is run only if the specified condition is true.

Usage Notes

1. The following is an example of the relational condition form of the IF command:

   ```
   IF (A .GT. B) HALT
   ```

   With the relational condition form of the IF command, the following is true:

   - When either variable or constant is a logical or character type, both must be of that same type, and no sign is permitted preceding either variable or constant.
   - When either variable or constant is a logical or complex type, only the relational operators .EQ. and .NE. (or = and ¬ =) may be used.

2. Substring notation is permitted for string variables.
3. For array elements, subscripts may have values beyond the range of the corresponding array dimensions. A warning message will be issued except in the special case where the last dimension is "1" or "*" and only the last subscript is out of range.

4. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

   ```
   ARY(I), ARY(3), ARY(I+3), ARY(I-3)
   ```

5. You cannot reference variables that are not currently defined, such as dummy arguments in an inactive subroutine.
6. Logical variables must have been set to VS FORTRAN logical constants for condition testing to produce predictable results.
7. Variables in different program units can be referenced by qualifying the variable names with program unit names:

   ```
   IF (HAIN.A .LT. SUB1.B) SET SUB1.B = MAIN.A
   ```

8. The command specified with IF cannot be HELP, QUALIFY, or FIXUP. You also cannot specify any of the full screen display commands.
9. When character variables or constants of unequal length are compared, the shorter is considered to be extended with blanks during the comparison.
10. IF is not permitted after the VS FORTRAN program has terminated.

Examples

1. At the statement labeled 100, determine whether logical variable OVER is true, and if it is true, reinitialize counter i and resume processing; if OVER is false, continue processing without resetting i.

   ```
   at /100 (if (over) set i=0%go)
   ```

2. At statement number 10, test if variable A in subroutine SUB is zero; if it is, suspend execution; if A is not zero, continue processing.

   ```
   at 10 (if (sub.a = 0.0) halt%go)
   ```

3. At statement 5, test the following set of logical conditions: if either A or B is true, go to the statement labeled 10 and continue processing; if neither is true, continue processing at the next executable statement.

   ```
   at 5 (if (a) go /10%if (b) go /10 %go)
   ```

4. At statement 7, if the first five characters of the variable SOLAR are ABCDE, set counter J to 2

   ```
   at 7 (if (solar(1:5)='ABCDE') set J=2)
   ```

LEFT (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Using Cursor-Sensitive Commands (page 20)

Function: LEFT scrolls the contents of a window so that columns to the left of those currently displayed in the window can be seen.

Abbreviation: None

Syntax

LEFT
[*number* **| PAGE | HALF | CSR | DATA | MAX]**

number
is the number of columns to scroll left, from 1 to 9999.

PAGE (or P)
scrolls left by the number of columns in the window.

HALF (or H)
scrolls left by half the number of columns in the window.

CSR (or C)
scrolls left by PAGE, unless the cursor is in the window, in which case the window is scrolled left by the appropriate number of columns to place the cursor at the edge of the window.

DATA (or D)
scrolls left by PAGE-1 number of columns. If only one column is visible, the scrolling is equivalent to that of PAGE.

MAX (or M)
scrolls left so that the leftmost column of the window will contain the left-most column of the contents for the window.

Usage Notes

1. LEFT is cursor-sensitive. The window that is scrolled is determined from the cursor position and the windows currently open.
2. LEFT cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
3. If an operand is not specified with LEFT, the scrolling amount is taken from the SCROLL ===> field on the main debugging panel.

Example

Scroll left so that the farthest left column of the window will contain the farthest left column of the contents for the window.

```
left max
```

LIST

Tasks:

Using Cursor-Sensitive Commands (page 20)
Specifying Output to a Print File (page 42)
Referring to Statements or Variables in Other Program Units (page 54)
Displaying Formatted Variable and Array Values (page 73)

Function: LIST displays the values of specified scalar variables, arrays, array elements, or string constants at the terminal or in a print file (AFFPRINT). Values can be displayed in a variety of formats. The specified or implied qualifier is shown with all variable or array names and the array elements may be displayed outside the defined dimensions.

Abbreviation: L

Syntax

LIST
{[*qual.*]*name*[:[*qual.*]*name*] | * | '*string*' | *number* | (*list*)}
[PRINT]
[FORMAT [(*code*)] | DUMP [(*code*)]]

[*qual.*]*name*
specifies the name of a variable, array, or array element used in the program. If a qualifier is specified, it overrides the current qualifier for the specified name.

[*qual.*]*name*:[*qual.*]*name*
specifies a range of variable, array, or array element names used in the program. If a qualifier is specified, it overrides the current qualifier for the specified name.

LIST displays all storage locations between the two variables. Unless FORMAT or DUMP is specified, the format of the displayed variables is the same as the type of the first variable.

*
specifies that a list of all the names in the currently qualified program unit is desired. Unless FORMAT or DUMP is specified, each is displayed according to its own type.

'*string*'
specifies a character string to be displayed as a remark. You can use this operand to help identify breakpoints.

number
specifies an integer or real numeric constant to be displayed as a remark. This function is useful for converting numbers, in conjunction with the FORMAT option.

list
specifies a list of individual specifications. Enclose the list in parentheses and separate entries with commas or blanks.

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

FORMAT [(*code*)] | DUMP [(*code*)]
specifies a particular data format:

- FORMAT displays the names listed and their values in the specified format.
- DUMP displays the address in storage of the names listed and their values in the specified format.
- (code) specifies the format or dump code for the names to be listed. The default format code is X. The default dump code is Z.
- FORMAT and DUMP are mutually exclusive.

FORMAT and DUMP codes for the LIST command are the same as for the AUTOLIST command, and are shown again in Figure 40.

Code	Output
L1	Logical*1
L4	Logical*4
I2	Integer*2
I4	Integer*4
R4	Real*4
R8	Real*8
R16	Real*16
C8	Complex*8
C16	Complex*16
C32	Complex*32
L	Logical with size closest to internal data size
I	Integer with size closest to internal data size
R	Real with size closest to internal data size
C	Complex with size closest to internal data size
X[nnn]	Hexadecimal with nnn bytes per data item (default to internal data size)
Z[nnn]	Hexadecimal with nnn bytes per data item (default to Z4)
A[nnn]	Character with nnn bytes per data item (default to internal data size)
H[nnn]	Character with nnn bytes per data item (default to continuous full line output)

Figure 40. DUMP and FORMAT Codes for the LIST Command

Usage Notes

1. In full screen mode, LIST can be used as a cursor-sensitive command. If the LIST command is already assigned to a PF key, place the cursor at a variable name in the source window, and press the PF key for LIST. The variable may include either subscript or substring notation. If both are present, only the subscript will be included in the command.

   Instead of assigning a PF key to the LIST command, you can type LIST on the command line and move the cursor to a variable name before pressing ENTER.

2. When you request an individual name or list of names, the default formatting of values is determined by the type of each name being displayed. When you request a range of names, the formatting of the values is determined by the format of the first name in the range. You may, however, specify a different format using the FORMAT or DUMP keyword. The locations of the listed names are identified in the output only if DUMP is specified.

3. VS FORTRAN defines storage layout only for arrays, variables in a common block (defined in a COMMON statement), and variables in equivalence groups (defined in an EQUIVALENCE statement). The relative positions of any other names in storage cannot be predicted. Names that you may expect to be adjacent in storage may be widely separated by other data. Therefore, a range specification for names other than array, equivalence, or common variables may produce unexpected results.

4. The length specification in a FORMAT or DUMP code may be entered with 1 through 3 digits. Thus, I4, I04, and I004 are equivalent.

5. A length specification of 0 in character and hexadecimal FORMAT and DUMP codes (for example, A0 or Z0) causes the data to be displayed as a continuous string, rather than split into pieces of some specified length.

6. If a FORMAT or DUMP code with no length specification is given for a range of variables or array elements, each variable or array element is displayed separately in the specified format. However, if a length specification is given, Interactive Debug will consider the entire storage area occupied by all the range of variables or array elements, or occupied by the entire array, as if it were broken into pieces, each with a length equal to that specified in the DUMP or FORMAT code, and will display each piece according to the specified format. For example, if PRIMES is a 2 x 3 array of INTEGER*4 values, then:

   ```
   list primes format(x)
   ```

   will cause a display of 6 values, each corresponding to an element of the array. However:

   ```
   list primes format(x2)
   ```

   will cause a display of 12 values, each displaying the contents of successive 2-byte storage areas within the array.

7. The DUMP option is not permitted with constant operands, including strings; using it will produce an error message.

8. An assumed size array cannot be listed by just specifying the array name; the specific element or range of elements must be specified. (An assumed size array is an array with the last upper bound declarator specified as an asterisk (*).) This restriction does not apply to arrays whose last dimension is "1," even though such arrays are otherwise treated as assumed size arrays. However, only the elements whose last subscript is "1" will be displayed if no subscripts are specified.

9. For array elements, subscripts may have values beyond the range of the corresponding array dimensions. A warning message will be issued except in the special case where the last dimension is "1" or "*" and only the last subscript is out of range.

10. Array subscripts must consist of simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
list ARY(J), list ARY(3), list ARY(J + 3), list ARY(J - 3)
```

11. Dummy arguments can only be displayed when the program unit in which they are defined is active. If not, an error message is issued. Results are unpredictable if you display a dummy argument that is not defined at the entry point called. Note that a program unit is not yet active when suspended at entry.

12. Variables in dynamic commons can only be displayed if the program unit used to qualify the variable has been activated at least once. (If not, an error message may be issued. However, if a variable has a large displacement in its dynamic common, Interactive Debug cannot detect that it is not initialized.)

13. Although a quoted string can be used as an operand on the LIST command, you cannot point the cursor at a quoted string in the source listing window (when using LIST as a cursor-sensitive command).

Examples

1. Display at the terminal the value of the variable named NCOUNT.

```
list ncount
```

2. Obtain a hexadecimal dump (FORMAT(X)) showing values of array variables A(1,1) through A(7,10). Have the display sent to the print data set.

```
list a(1,1):a(7,10) print format
```

3. List the decimal number 12345 in hex.

```
list 12345 f(x)
```

4. Display an entire array (CHARAY) containing a series of 30-character alphabetic strings so that each character string is separated from the others. (If the array is declared in the program to be a CHARACTER*30 array, the elements of the array will be separated from each other when the array is listed.)

```
list charay format(a30)
```

5. Display the value of the variable named CTR in subroutine SUB1.

```
list sub1.ctr
```

6. Display the message "inside loop" whenever ISN 100 is run.

```
at 100 (list 'inside loop'%go)
```

7. Display variable LONG_NAME_VAR in program unit SUB1, and variable SHORT in program unit SUB2. This illustrates the LIST format in support of long (31 character) names. Note that long names cause a line break to allow alignment of the "="s without excessive horizontal spread.

```
list (sub1.long_name_var,sub2.short)
SUB1.LONG_NAME_VAR
                =          10
SUB2.SHORT      =          25
```

8. Display values of the second through sixth characters in the character variable PORT

```
list PORT(2:6)
```

LISTBRKS

Tasks:

Specifying Output to a Print File (page 42)
Referring to Statements or Variables in Other Program Units (page 54)

Function: LISTBRKS provides the following information:

- All breakpoints that are currently set, including entry, exit, and statement breakpoints
- All WHEN conditions (both on and off) that are currently defined, and the condition being tested
- The current HALT status (OFF, STMT, GOTO, or ENTRY)

Abbreviation: LB

Syntax

```
LISTBRKS
     [PRINT]
```

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

Example

Possible output:

```
CURRENT BREAKPOINTS:
  MAIN.15/30
  SUB.ENTRY
  SUB.8/10
CURRENT WHEN CONDITIONS:
  ABCD ON (SUB.X > 5)
  EFGH ON (CH(1:2)='AB')
CURRENT HALT STATUS:  OFF
```

LISTFREQ

Tasks:

Specifying Output to a Print File (page 42)
Determining Statement Execution Frequency (page 62)

Function: LISTFREQ displays the number of times statements in the currently qualified program unit have been run. This command can also be used to list the statements that have not been run.

Abbreviation: LF

Syntax

LISTFREQ
[[*qual.***]{***number***[:[***qual.***]***number***] | ENTRY | EXIT} |**
(*number***/ENTRY/EXIT** *list***)]**
[ZEROFREQ]
[PRINT]

qual.
specifies a program unit name prefix to temporarily override the current qualifier for the prefixed operand only.

number
specifies the statement label, ISN, or sequence number of an executable VS FORTRAN statement whose execution counts are to be listed or checked for zero. Precede a statement label with a slash to distinguish it from an ISN or sequence number.

number:[*qual.*]*number*
specifies a range of statement labels or statement numbers (ISNs or sequence numbers) whose execution counts are to be listed or checked for zero. Statement labels can be combined with ISNs or sequence numbers in the range specification, but the first and last must be executable statements. Precede each statement label with a slash.

Statement identifiers can be qualified with a program unit name. The default program unit for the first identifier is the current qualifier. The default program unit for the second identifier is the program unit specified or defaulted for the first identifier. Both identifiers must have the same program unit in effect.

(*number*/**ENTRY/EXIT** *list*)
requests a list of statement labels, ISNs or sequence numbers, entry points, exit points, and ranges. (Note that ENTRY and EXIT are not permitted in a range.) The frequency for each specified statement is listed. Enclose the list in parentheses, with individual entries separated by commas or blanks. Precede each statement label with a slash.

ZEROFREQ
requests a list of statements that have not been run. All statements may be tested, or specific statements may be specified using the options discussed above.

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. If no operand is specified, the counts are displayed for ENTRY, EXIT, and all executable statements with debugging hooks in the currently qualified program unit.
2. If LISTFREQ is issued after an ENDDEBUG command, the execution counts displayed are those that existed when ENDDEBUG was issued.
3. Execution counts of unhooked or collapsed statements are not displayed. Instead, you will see the phrase "NO HOOK" or "COLLAPSED STMT."
4. Before a RENT program unit is first entered, all statements are considered to have no hook. Statements in a reentrant program unit that are excluded in the AFFON file will show "COLLAPSED STMT" on the LISTFREQ display.
5. After ENDDEBUG is issued, LISTFREQ displays counts for all non-collapsed statements.
6. Statements in a debug packet will be treated as collapsed in VS FORTRAN programs compiled prior to Version 1 Release 4.0. If the program is compiled with VS FORTRAN Version 1 Release 4.0 or later, the debug statements are inserted directly into the code and LISTFREQ will show duplicate statements in addition to the DEBUG packet code.
7. If you request ZEROFREQ, Interactive Debug displays only hooked statements that have *not* been run.

Examples

1. On the print data set, list how many times each executable statement in the currently qualified program unit has been run.

```
listfreq print
```

   Your output might look something like this:

```
STATEMENT              FREQUENCY
MAIN.ENTRY              NO HOOK
MAIN.EXIT               NO HOOK
MAIN.6                  1
MAIN.7                  COLLAPSED STMT
MAIN.8                  COLLAPSED STMT
MAIN.9                  NO HOOK
MAIN.10                 3
MAIN.11                 3
MAIN.12                 NO HOOK
MAIN.13/10014           NO HOOK
```

2. List the statements that have *not* been run in the currently qualified program unit.

```
listfreq zerofreq
```

3. List how many times some specific statements have been run in the currently qualified program unit.

```
listfreq (10:/80,300,/95,/105,ENTRY)

listfreq (20:130 ENTRY 250 /1000)
```

4. List how many times the executable statements 12 through 15 in subroutine SUB1 have been run.

```
listfreq sub1.12:sub1.15
```

LISTINGS (full screen mode only)

Tasks: Changing Listing Information (page 22)

Function: LISTINGS displays the Interactive Debug listings panel.

Abbreviation: None

Syntax

LISTINGS

Usage Notes

1. LISTINGS cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. LISTINGS will operate as usual with a parameter list, but the panel will indicate the message "THE PARAMETERS ARE INVALID AND HAVE NO EFFECT WITH THIS COMMAND."

Sample listings panel in CMS:

```
                    INTERACTIVE DEBUG LISTINGS PANEL            ROW  1 OF 5
COMMAND ===>                                                  SCROLL ===> PAGE

PROGRAM UNIT NAME              CMS LISTING FILE      DISPLAY

MAIN_____________________   XYZPROG  LISTING  A1    Y

THCOM____________________   THCOM    LISTING  E1    N
AFFA756E LISTING FILE COULD NOT BE FOUND ON ACCESSED DISKS.
THE281J__________________   THE281J  LISTING  A1    Y

THY1066__________________   THY1066  LISTING  *_    N

****************************** BOTTOM OF DATA ********************************
```

The column labelled "PROGRAM UNIT NAME" identifies the debuggable VS FORTRAN program units. The second column, "CMS LISTING FILE," indicates the names of files where the listings are to be found. In the column labelled "DISPLAY," YES indicates that the listing will be displayed in the source window.

A message area is located under each program unit name. In the sample panel above, the CMS message LISTING FILE COULD NOT BE FOUND ON ACCESSED DISKS indicates that the specified file was not found on any of your accessible disks. The equivalent TSO message is LISTING DATASET COULD NOT BE FOUND IN ALLOCATED DATASETS.

After filling in the listings panel, return to the execution panel by entering END, usually PF key 3.

LISTSAMP

Tasks:

Specifying Output to a Print File (page 42)
Program Sampling (page 63)
Vector Tuning Assistance (page 67)

Function: LISTSAMP lists sampling counts by statement, program unit, or DO loop. Percentage of program unit samples and percentage of total number of samples are also listed, along with a bar chart of the sampling counts.

Abbreviation: None

Format 1

Syntax for Listing Sampling Counts by Statement

```
LISTSAMP
     {[qual.]number[:[qual.]number] | [qual.]ENTRY | [qual.]* | (list) | *.*}
     [DIRECT | CALLED | ALL]
     [TOP[(n)]]
     [PRINT]
```

Format 2

Syntax for Listing Sampling Counts by Program Unit

```
LISTSAMP
     {unit-name | (unit-name-list) | *}
     SUMMARY
     [DIRECT | CALLED | ALL]
     [TOP[(n)]]
     [PRINT]
```

Format 3

Syntax for Listing Sampling Counts by DO Loop

```
LISTSAMP
     {[qual.]number[:[qual.]number] | [qual.]* | (list) | *.*}
     DOLOOP
     [DIRECT | CALLED | ALL]
     [TOP[(n)]]
     [PRINT]
```

[qual.]
specifies a program unit name to temporarily override the current qualifier for the prefixed operand only.

[qual.]number
is the statement label, ISN, or sequence number of an executable statement (formats 1 and 2) or DO statement for a DO loop (format 3) for which sampling information is to be listed. Qualification is optional. A statement label must be prefixed with a slash (/).

[qual.]number:*[qual.]number*
specifies a range of statements in the program (formats 1 and 2) or a range of statements in the program for all DO loops in the range (format 3) for which sampling counts is to be displayed. (A DO loop is in the range if the DO statement of the loop is in the range. All the statements of the loop do not have to be in the range. The starting and ending statement identifiers do not have to be DO statements.) Qualification is optional. If the second qualifier is specified, it must be the same as that specified for the first qualifier.

*[qual.]***ENTRY**
indicates that the sampling count for the entry and exit code of the specified or currently qualified program unit is to be listed. There is only one sampling count to cover both entry and exit code.

*[qual.]****
indicates that sampling counts for all statements (formats 1 and 2) or DO loops (format 3) in the specified or currently qualified program unit are to be displayed.

list
specifies a list of individual statement (formats 1 and 2) or DO loop (format 3) specifications. Enclose the list in parentheses and separate entries using commas or blanks.

.
indicates that sampling counts for all statements in all programming units (formats 1 and 2) or all DO loops in the program (format 3) are to be displayed.

unit-name
specifies the name of a program unit whose sampling summary is to be listed. This must be a VS FORTRAN unit compiled with SDUMP.

unit-name-list
specifies a list of program unit names separated by commas or blanks.

*
indicates that all program units, debuggable or not, are to be included. In addition, there are two special names that are reported when "*" is specified:

*LIBRARY shows the sampling count accumulated for all VS FORTRAN Library modules other than the mathematical functions and the Error Monitor. This includes lower-level calls to system services.

*UNKNOWN shows the count of sampling interrupts that could not be assigned to any program unit.

SUMMARY
indicates that sampling counts are to be summarized by program unit. This keyword is allowed only in format 2.

DOLOOP
indicates that sampling counts are to be summarized by DO loop. This keyword is allowed only in format 3.

DIRECT | CALLED | ALL
DIRECT indicates that interruptions occurring in the code are to be included in the sampling counts for that code. DIRECT is the default.

CALLED indicates that interruptions occurring in lower-level routines are also to be included in the sampling counts of the code being sampled. This option is valid only when sampling is initiated with the CALLED option.

ALL indicates that sampling counts are to be the sum of the DIRECT and CALLED counts.

TOP(*n*)
indicates that only *n* number of statements (format 1), programming units (format 2), or DO loops (format 3) having the highest counts are to be listed. The output is sorted in descending order by count. The default for n is "1"; the maximum value is "9999."

If TOP is not specified, the output is in the order shown in the specification list, and within that by statement table order.

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. The LISTSAMP command is valid only when sampling has been performed.
2. Qualifiers and unit names are restricted to VS FORTRAN program units compiled with SDUMP.
3. Non-FORTRAN program units (and units compiled with NOSDUMP) are identified by the entry ID located using the value of GPR 15 saved in the save area. Programs that do not follow MVS standards for entry identifiers will not be correctly identified. Note that the entire ID string is shown, up to 31 characters. This often includes blanks and additional information such as date and time of compilation.
4. Sampling counts for non-debuggable program units cannot be requested by name; however, the '*' operand with the SUMMARY option will show sampling counts for both debuggable and non-debuggable program units, including VS FORTRAN library counts.
5. For format 3:
   - The program information for a program unit must have been successfully obtained at initialization of the debugging session in order to display sampling counts by DO loop for the program unit.
   - A LISTSAMP command with a specification list or a program-unit-name list will run if one or more of the DO statement identifiers or program unit names in the list are valid.
   - Since there are likely to be statements in the program that are not contained in DO loops, DIRECT sampling counts for all DO loops in the program may not add up to the total number of samples. Also, since DO loops can be nested, DIRECT sampling counts for all DO loops in the program may add up to more than the total number of samples.

- If a DO loop has been distributed into several loops due to vectorization, then sampling counts are summed for all of the distributed loops, and displayed as if the DO loop was not distributed.

Examples

1. Display a summary of the sampling counts for all program units.

```
listsamp * summary
```

Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 20 MS; TOTAL NUMBER OF SAMPLES
       WAS 9745.
DIRECT SAMPLES:
 PROGRAM UNIT     SAMPLES %TOTAL
 MAIN                 613   6.35 *
 SUB1                  15   0.15
 SUB2                5763  59.71 ************
 SUB3                1882  19.31 ***
 S#IN                1251  12.83 ***
 *LIBRARY             128   1.31
 *UNKNOWN              93   1.01
```

2. Display a summary of the sampling counts for SUB1 and SUB2.

```
listsamp (sub1,sub2) summary
```

Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 20 MS; TOTAL NUMBER OF SAMPLES
       WAS 9745.
DIRECT SAMPLES:
 PROGRAM                SAMPLES %TOTAL
 SUB1                        15   0.15
 SUB2                      5763  59.71 ************
```

3. Display the sampling counts for a range of statements in SUB2. (Sampling counts will include interruptions which occurred in the code of a statement as well as in any lower-level routines called by the statement.)

```
listsamp sub2.10:20 all
```

Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 20 MS; TOTAL NUMBER OF SAMPLES
       WAS 9745.
SUM OF DIRECT AND CALLED SAMPLES:
 STATEMENT              SAMPLES  %UNIT  %TOTAL
 SUB2.10/920               1142  73.15   12.53 ***************
 SUB2.11/930                231  15.42    3.02 ***
 SUB2.13                     12   1.22    0.15
 SUB2.15              COLLAPSED
 SUB2.16                     22   2.38    0.32
 SUB2.18                     14   1.24    0.16
 SUB2.20                     46   5.17    0.77 *
```

4. Display the four highest counts in SUB2. (Sampling counts will include interruptions which occurred in the code of a statement as well as in any lower-level routines called by the statement.)

```
listsamp sub2.* top(4) all
```

Possible output:

```
PROGRAM SAMPLING INTERVAL WAS 20 MS; TOTAL NUMBER OF SAMPLES
       WAS 9745.
SUM OF DIRECT AND CALLED SAMPLES:
 STATEMENT              SAMPLES  %UNIT  %TOTAL
 SUB2.10/920               1142  73.15   12.53 ***************
 SUB2.11/930                231  15.42    3.02 ***
 SUB2.20                     46   5.17    0.77 *
 SUB2.ENTRY/EXIT             42   4.92    0.72 *
```

5. List DIRECT sampling counts by DO loop for the 10 DO loops in the currently qualified program unit that have the highest counts, sorted in order from largest to smallest number of samples.

```
listsamp * doloop top(10)
```

Possible output:

```
PROGRAH SAHPLING INTERVAL WAS 20 HS; TOTAL NUHBER OF SAHPLES
        WAS 9745.
DIRECT SAHPLES:
DO LOOP                    SAHPLES  %UNIT %TOTAL
HAIN.9                         613 100.00   6.35 ***********
HAIN.49                        165  26.92   1.69 ***
HAIN.41                         46   7.48   0.77 *
HAIN.40                         46   7.48   0.77 *
HAIN.27                         35   5.69   0.51 *
HAIN.26                         35   5.69   0.51 *
HAIN.24/200                     35   5.69   0.51 *
HAIN.34/300                     22   3.58   0.32
HAIN.33                         22   3.58   0.32
HAIN.32                         22   3.58   0.32
```

6. List the CALLED sampling count for the DO loop whose DO statement is at ISN 53 in program unit LOLLI, and place the output in the print file.

```
listsamp lolli.53 doloop called print
```

LISTSUBS

Tasks:

Specifying Output to a Print File (page 42)
Displaying Information about Debuggable Program Units (page 54)

Function: LISTSUBS displays a list of all VS FORTRAN program units compiled with SDUMP, including those not listed in AFFON, with the following information for each:

- Compiler level used to produce the object code (if it can be determined)
- Optimization level
- Vectorization level
- Hook existence
- Timing status
- Load status for units compiled with RENT

Abbreviation: LS

Syntax

```
LISTSUBS
    [PRINT]
```

PRINT
specifies that output be sent to the print file instead of the terminal.

Sample output:

```
PROGRAM UNIT    COMPILER    OPT  HOOKED   TIMING
MAINLINE        VSF  2.3.0  V2   YES      ON
SUBBUILD        VSF  1.4.0   3   NO       OFF  RENT NOT LOADED.
SUBDOWN         VSF  (TEST)  0   YES      OFF
```

In the sample output above, VSF (TEST) means that the program unit was compiled prior to VS FORTRAN Version 1 Release 4.0, and the TEST option was specified. In this case, it is not possible to determine the VS FORTRAN release level.

For non-vectorized programs, the "OPT" column displays the level of optimization: 0, 1, 2, or 3. For vectorized programs, the "OPT" column displays "V2."

YES means that hooks are installed at entry and exit points and possibly at some or all statement boundaries as well. The hook settings are controlled by the AFFON file. NO in the "HOOKED" column indicates that no hooks are installed in the program unit.

ON in the "TIMING" column indicates that timing has been activated for the program unit. The TIMER command is used to set timing ON or OFF.

The possible load status indications for RENT program units are:

RENT NOT LOADED The program unit has not yet been called, and has not been located (although it may actually be in storage).

RENT IN USER AREA The program unit has been called and is in user-owned storage.

RENT IN PROT AREA The program unit has been called and is in protected storage.

LISTTIME

Tasks:

Specifying Output to a Print File (page 42)
Program Unit Timing (page 66)
Vector Tuning Assistance (page 67)

Function: LISTTIME displays timing information for either program units or analyzable DO loops. The following information is provided for each program unit or DO loop:

- Total time: total execution time.
- % total: percentage of total execution time.
- Invocations: number of invocations.

Timing information for program units also includes average time (calculated by dividing the total time by the number of invocations).

Use the TIMER command (page 194) to activate, deactivate, or reset timing.

Abbreviation: LT

Format 1

Syntax for Displaying Timing Information by Program Unit

```
LISTTIME
    [PRINT]
```

Format 2

Syntax for Displaying Timing Information by DO Loop

```
LISTTIME
    { [qual.]number[:[qual.]number] | [qual.]* | (list) | *.* }
    DOLOOP
    [PRINT]
```

PRINT
specifies that output be sent to the print file instead of the terminal.

[*qual.*]
specifies a program unit name to temporarily override the current qualifier for the prefixed operand only.

[*qual.*]*number*
specifies the statement label, ISN, or sequence number of the DO statement for a DO loop whose timing is to be displayed. Qualification is optional. A statement label must be prefixed with a slash (/).

[*qual.*]*number*:[*qual.*]*number*
specifies a range of statements in the program for which the timing for all DO loops in the range is to be displayed. Qualification is optional. (A DO

loop is in the range if the DO statement of the loop is in the range. All the statements of the loop do not have to be in the range. The starting and ending statement identifiers do not have to be DO statements.) Qualification is optional. If the second qualifier is specified, it must be the same as that specified or defaulted for the first qualifier.

*[qual]**
specifies that the timing for all DO loops in the specified or currently qualified program unit is to be displayed.

list
specifies a list of individual DO loop specifications. Enclose the list in parentheses and separate entries using commas or blanks.

.
specifies that timing for all DO loops in the program is to be displayed.

Usage Notes

1. The timing information provided by Interactive Debug may include overhead caused by the debugging hooks in your program. Thus, the timing information is not an accurate representation of the time it takes to run *without* Interactive Debug.

   To reduce DO loop timing distortion, restrict hooks in the program to only DO loop analysis hooks. This can be done by creating an AFFON file for the debugging session, with one entry of `(ALL) * DOLOOP` inside it.

2. DO loop timing cannot be performed on unanalyzable loops.
3. The program information for a program unit must have been successfully obtained at initialization of the debugging session in order to time DO loops in the program unit.
4. The DO statement for a DO loop must have been included in the AFFON statement restriction list (page 43) for the program unit in order to display timing for the loop.
5. A LISTTIME command with a specification list will run if one or more of the DO statement identifiers in the list are valid.
6. If a DO loop has been distributed into several loops due to vectorization, then all of the distributed loops are timed and the time is presented as if the DO loop was not distributed.
7. Time is measured in microseconds. Average time is rounded to the nearest microsecond. Percentages are rounded to the nearest hundredth of a percent.
8. The execution times displayed are CPU times. The time may not include time spent in paging and other operating system activities.
9. The accuracy of the times depends on CPU model and operating system. The accuracy may be sensitive to system load.
10. Timing for very small subroutines may be erratic.
11. Timing is measured separately for each entry point in the program unit.
12. If LISTTIME is issued after an ENDDEBUG command, the times and activation counts are those that existed when ENDDEBUG was issued.

Examples

1. Display timing information to the terminal for all program units.

```
listtime
```

Possible output:

```
ENTRY POINT                TOTAL TIME  %TOTAL  INVOCATIONS  AVERAGE TIME
MAIN                            54274    8.69            1         54274
INITIALIZE_OVERVIEW_SYSTEM
                                 3423    0.55            1          3423
FIELDS                          52438    8.40           12          4370
FIELDS_ALPHA                     2432    0.39           24           101
FIELDS_BETA                     23331    3.74            5          4666
XSORT                          487391   78.05          563           866
FHALL                            1142    0.18            2           571
```

2. Display timing information to the terminal for all DO loops.

```
listtime *.* doloop
```

Possible output:

```
DO LOOP                      TOTAL TIME  EXECUTIONS  AVERAGE TIME  STATUS
MAIN.13                           23421           1         23421   ON
INITIALIZE_OVERVIEW_SYSTEM.122/10100
                                   1342           1          1342   ON
FIELDS_ALPHA.23                    2234          24            93   ON
FIELDS_ALPHA.24                    1503         120            63   ON
FIELDS_BETA.48                    19231           5          3846   ON
FIELDS_BETA.49                    10765          25           431   OFF
FIELDS_BETA.65                     2342           5           468   ON
XSORT.12                         473223         563           841   OFF
XSORT.23                         365253         251          1455   OFF
```

In interpreting the values in the EXECUTIONS column, consider the following:

- If a DO loop is divided into several loops due to vectorization, the number of executions of the loop is the number of executions of the first recurrence of the loop.
- If the DO loop (or first recurrence) is nested in a vectorized loop, the number of executions will be less than that obtained if the loop were nested in a scalar loop. This is because the inner loop will be run only once for each vector section processed by the outer loop.

  If Z denotes vector section size, and N is the iteration count of the outer loop, then the inner loop will be run INT ((N - 1) / Z) + 1 times for each execution of the outer loop.

3. Display timing information for DO loops in subroutine SUBSEL.

```
lt subsel.* d
```

LISTVEC

Tasks:

Specifying Output to a Print File (page 42)
Vector Tuning Assistance (page 67)

Function: LISTVEC displays vector length and stride information. The LISTVEC report contains:

- the total number of executions of the DO loop
- the average iteration count (length) of a DO loop
- the compiler estimate for the iteration count
- the average stride for each array indexed by the DO variable of the loop
- the compiler estimates for these strides

Abbreviation: LV

Syntax

LISTVEC
{ *[qual.]number***[:***[qual.]number***] | [***qual.***]* | (***list***) | *.* }**
[TOP [(*n***)]]**
[PRINT]

[qual.]number
is the statement label, ISN, or sequence number for a DO statement of a DO loop for which vector information is to be displayed. A statement label must be prefixed with a slash (/). *qual* specifies a program unit name to temporarily override the current qualifier for the prefixed operand only.

[qual.]number:*[qual.]number*
specifies a range of statements in the program for which the vector information for all DO loops in the range is to be displayed. (A DO loop is in the range if the DO statement of the loop is in the range. All the statements of the loop do not have to be in the range. The starting and ending statement identifiers do not have to be DO statements.) If the second *qual* is specified, it must be the same as that specified or defaulted for the first *qual*.

*[qual.]**
indicates that the vector information for all DO loops in the specified or currently qualified program unit is to be displayed.

list
is a list of individual DO loop specifications. Enclose the list in parentheses and separate entries with commas or blanks.

.
indicates that vector information for all DO loops in the program is to be displayed.

TOP[(*n***)]**
specifies that only array references with the highest *n* average strides are to be displayed. (Since there may be more than one array reference with the same stride, more than *n* array references may be displayed.) The references are listed in order from largest stride to smallest. Array references

with the same stride are listed in order of statement appearance in the source listing.

PRINT
indicates that output be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. Length and stride statistics are only available for DO loops that were analyzable by the compiler. An error message is issued if the LISTVEC command explicitly specifies a DO loop that was not analyzable. If a range of DO loops is specified on the LISTVEC command, and a loop in the range was not analyzable, then the message "NOT ANALYZABLE" appears in place of length and stride information for the loop in the LISTVEC output.
2. The program information for a program unit must have been successfully obtained at initialization of the debugging session in order to display length and stride statistics for DO loops in the program unit.
3. VECSTAT must be issued to turn on length and stride recording before the LISTVEC command can give useful information. If not, then the average length, average stride, and number of invocations for the loop are set to zero.
4. The DO statement for a DO loop must have been included in the AFFON statement restriction list for the program unit in order to record run-time vector information. If not, the message "NO HOOK" appears instead of values for number of executions, average iteration count, and average strides.
5. A LISTVEC command with a specification list runs if one or more of the DO statement identifiers in the list are valid.
6. Length and stride statistics are updated during the execution of the DO statement for the loop. Thus, if the user breaks at a statement in the middle of the loop and performs a LISTVEC, the number of executions, average iteration count, and average strides are already updated for that execution of the loop.
7. Some strides for non-vectorized DO loops may not be obtainable at run-time. For such strides, the word "UNKNOWN" appears instead of a value for the average stride in the LISTVEC output.
8. For the TOP(*n*) option, average strides marked "NO HOOK" or "UNKNOWN" are not included.
9. A question mark (?) following a compiler estimate indicates that the value could not be accurately determined at compile time. The estimate shown is an arbitrary value that was chosen by the compiler for vector cost analysis.
10. If a DO loop has been distributed into several loops due to vectorization, then length and stride information is collected for all of the distributed loops, and presented as if the DO loop was not distributed.
11. DO loops are listed in the LISTVEC output in order of appearance in the source listing. Array references for strides of a loop are listed in order of statement appearance in the source listing. Multiple references in the statement are not listed in any particular order.
12. The PURGE command can be used to terminate output of a LISTVEC command after the output has been suspended by an attention interrupt.

Examples

1. Display length and stride information for all DO loops in the program.

   ```
   LISTVEC *.*
   ```

2. Display length and stride information for all DO loops in SUBROUTINE1 from statement label 10 to statement label 40.

   ```
   LISTVEC SUBROUTINE1./10:/40
   ```

Sample Output

Below is an example of LISTVEC output for all DO loops of a program. The DO statement identifier is listed first, followed by the number of executions, average and estimated iteration counts, and a list of strides for each array reference indexed by the DO variable of the loop.

In interpreting the values for the TOTAL NUMBER OF EXECUTIONS line, consider the following:

- If a DO loop is divided into several loops due to vectorization, the number of executions of the loop is the number of executions of the first recurrence of the loop.
- If the DO loop (or first recurrence) is nested in a vectorized loop, the number of executions is less than that obtained if the loop is nested in a scalar loop. This is because the inner loop only runs once for each vector section processed by the outer loop.

  If Z denotes vector section size, and N is the iteration count of the outer loop, then the inner loop runs INT ((N - 1) / Z + 1 times for each execution of the outer loop.

```
IVA04.7:
  NOT ANALYZABLE
IVA04.10:
  STATUS = ON
  TOTAL NUMBER OF EXECUTIONS =         75
  AVERAGE ITERATION COUNT    =         13
  ESTIMATED ITERATION COUNT  =         20?
    STATEMENT             ARRAY              AVG STRIDE  EST STRIDE
  V IVA04.12              A1                          3           3
  V IVA04.12              A1                    UNKNOWN           3?
  V IVA04.12              C1                         30           4?
  S IVA04.13              B1                          3           3
  S IVA04.13              B1                          3           3
  S IVA04.13              C1                         21          21
IVA04.11:
  STATUS = ON
  TOTAL NUMBER OF EXECUTIONS =        945
  AVERAGE ITERATION COUNT    =          8
  ESTIMATED ITERATION COUNT  =        400?
    STATEMENT             ARRAY              AVG STRIDE  EST STRIDE
  S IVA04.12              A1                        500         500
  S IVA04.12              A1                        500         500
  S IVA04.12              C1                       2500         400?
  V IVA04.13              B1                        500         500
  V IVA04.13              C1                       5500        5500
IVA04.20:
  STATUS = OFF
  TOTAL NUMBER OF EXECUTIONS =    NO HOOK
  AVERAGE ITERATION COUNT    =    NO HOOK
  ESTIMATED ITERATION COUNT  =         20?
  STATEMENT               ARRAY              AVG STRIDE  EST STRIDE
  IVA04.21                A1                    NO HOOK           1?
  IVA04.21                B1                    NO HOOK           1?
TALLY_UP_IVA_STATISTICS.42:
  STATUS = OFF
  TOTAL NUMBER OF EXECUTIONS =         30
  AVERAGE ITERATION COUNT    =        100
  ESTIMATED ITERATION COUNT  =         20?
    STATEMENT             ARRAY              AVG STRIDE  EST STRIDE
  V TALLY_UP_IVA_STATISTICS.43
                          A1                          1           1
  V TALLY_UP_IVA_STATISTICS.43
                          CALCULATED_IVA_GSTATS
                                                      4           1?
```

Figure 41. Sample LISTVEC Output

MOVECURS (full screen mode only)

Function: MOVECURS toggles the cursor between the command line and its most recent position in the main debugging panel.

Abbreviation: MC

Syntax

MOVECURS

Usage Notes

1. MOVECURS cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. You may find it convenient to set MOVECURS to a PF key. To do this, specify KEYS on the command line. You will then see a list of all current PF key assignments. You can change the CURSOR key (usually PF 12) to MOVECURS by typing

   ```
   MOVECURS;
   ```

   next to the appropriate PF key number and pressing ENTER. Note a semicolon must follow MOVECURS. For more information on PF keys, see page 19.

NEXT

Tasks:

Controlling Program Execution (page 58)
Processing External Files (page 76)

Function: NEXT suspends program execution at the next statement, entry, or exit with a debugging hook. Because some statements may not have been included in the AFFON list or may have been collapsed, execution need not necessarily be suspended at the next statement to be run.

Abbreviation: N

Syntax

```
NEXT
```

Usage Notes

1. You are notified of the point where execution is suspended because of NEXT.
2. NEXT suspension is not a breakpoint. It is not listed by LISTBRKS.
3. Certain commands are not allowed while I/O is active. When execution is suspended for output, the NEXT command can be issued to cause execution to be suspended again after the completion of the I/O operation. At the next statement boundary with a debugging hook, execution will be suspended, and you may issue other commands.
4. If the statement at which execution is to be suspended has a breakpoint set with an AT command, including a command list that causes execution to resume, the NEXT command will not cause execution to be suspended at that statement.
5. NEXT is not permitted after the VS FORTRAN program has terminated.
6. The STEP command (page 189) can be issued to replace NEXT/GO combinations.

Examples

1. Suspend execution after executing one statement.

```
AT: MAIN.10
next
go
NEXT: MAIN.11
next
go 40
NEXT: MAIN.40
```

2. You are in an error exit, and want to suspend execution after having performed corrective action.

```
ERROR EXIT: ERROR 209 AT MAIN.11
next
fixup
STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.
NEXT: MAIN.12
```

OFF

Tasks: Using Cursor-Sensitive Commands (page 20)

Function: OFF removes breakpoints in the currently qualified program unit.

Abbreviation: None

Syntax

OFF
{ [*qual.*]{*number*[:[*qual.*]*number*] | **ENTRY** | **EXIT**}
| * | (*number*/**ENTRY**/**EXIT** *list*) }

qual.
specifies a program unit name prefix to temporarily override the current qualifier. The program unit name is used for the prefixed operand only.

number
specifies the statement label, ISN, or sequence number of a single breakpoint you want to remove. Precede a statement label with a slash to distinguish it from an ISN or sequence number.

number:[*qual.*]*number*
specifies a range of statement labels and/or statement numbers (ISNs or sequence numbers). Breakpoints set at any statement within the range are removed. Statement labels and statement numbers (ISNs or sequence numbers) can be combined in the range. Precede each statement label with a slash.

Statement identifiers can be qualified with a program unit name. The default program unit for the first identifier is the current qualifier. The default program unit for the second identifier is the program unit specified or defaulted for the first identifier. Both identifiers must have the same program unit in effect.

ENTRY
specifies that the entry breakpoint is to be removed.

EXIT
specifies that the exit breakpoint is to be removed.

*
specifies that all breakpoints will be removed from the qualified program unit.

(*number*/**ENTRY**/**EXIT** *list*)
specifies a list of statement labels, ISNs or sequence numbers, entry points, exit points, and ranges of numbers. Breakpoints set at each specified statement and within each range are removed. Enclose the list in parentheses, and separate entries with commas or blanks. Precede each statement label with a slash.

If the number of a statement that does not have a debugging hook is entered in the number list, an error message is issued but breakpoints are still removed from the remaining statements.

Usage Notes

1. In full screen mode, OFF can be used as a cursor-sensitive command. If the OFF command (with no parameters) is already assigned to a PF key, place the cursor in the prefix area of the source window, and press the PF key for OFF.

   Instead of assigning a PF key to the OFF command, you can type the OFF command in the prefix area, or you can type OFF on the command line and move the cursor to the target statement number before pressing ENTER.

2. OFF is not permitted after the VS FORTRAN program has terminated.

Examples

1. Remove all breakpoints in the currently qualified program unit.

   ```
   off *
   ```

2. Remove breakpoints at statement numbers 120 and 560 in program SUB1.

   ```
   off (sub1.120 sub1.560)
   ```

3. Remove specific breakpoints in the currently qualified program unit.

   ```
   off (20:80 /100 ENTRY)
   ```

OFFWN

Tasks: Controlling Program Execution (page 58)

Function: OFFWN turns off the monitoring of WHEN conditions.

Abbreviation: None

Syntax

OFFWN
{*condition-name* | * | (*condition-name-list*)}

condition-name
specifies the 1- through 4-character name of a WHEN condition that is currently being monitored and that you want to stop monitoring.

*
turns off all WHEN condition monitoring.

(*condition-name-list*)
specifies a list of such WHEN condition names. Monitoring is stopped for all of them. Enclose the list in parentheses, with individual names separated by commas or blanks.

Usage Notes

1. Use of OFFWN to turn off monitoring of a condition does not remove the definition of the condition. Any condition can be reactivated by using the WHEN command.
2. To see the currently defined conditions, use the LISTBRKS command.
3. OFFWN is not permitted after the VS FORTRAN program has terminated.

Examples

1. Stop monitoring all WHEN conditions.

```
offvm *
```

2. Stop monitoring a certain WHEN condition called ABS.

```
offvm abs
```

POSITION (full screen mode only)

Tasks:

Using Cursor-Sensitive Commands (page 20)
Changing the Way Your Windows Look (page 18)

Function: POSITION allows you to search for a line and place it at the top of a specified window.

Abbreviation: POS

Syntax

POSITION
number

number
specifies either a statement number (an ISN or sequence number), a log line number, or a monitor line number to be used as the target.

Usage Notes

1. POSITION cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. Only the last 1000 lines of the log window and the first 1000 lines of the monitor window are available for display during the debugging session. A target located below the last 1000 lines will cause an error message to be displayed.
3. Sequence numbers (in columns 73 through 80) can be used only for program units that were compiled with VS FORTRAN Version 2 with the SDUMP(SEQ) option. In all other cases, ISNs must be used.
4. If you have MOVECURS; assigned to a PF key, you can type the POSITION command on the command line. Then press the MOVECURS PF key to move the cursor to its previous position in the main debugging panel, and press ENTER to position the specified line at the top of the window.

Example

Search for line number 100 in either the source, monitor, or log window, depending on where the cursor is currently positioned, and place it at the top of the window.

```
position 100
```

PREVDISP (full screen mode only)

Function: PREVDISP re-displays the previous panel displayed by the application program (if ISPF was used). The panels are saved automatically by ISPF, and are re-displayed with an ISPF message "Saved Panel Display."

Warning: The variables in the program are actually reset to the values of the variables in the previously displayed panel, (unless the application program has used VDELETE for those variables). The information in a saved panel is re-displayed exactly as it appeared when the panel was originally displayed, and may no longer be correct. You will not receive a message to remind you of this, nor will the change in values be logged.

Abbreviation: PREV

Syntax

```
PREVDISP
```

Usage Notes

1. PREVDISP cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. If no previous panel exists, the first PREVDISP Help panel is displayed.

   If the HELP panel is displayed, you cannot access the main menu or subsequent panels.
3. The display of a previous panel is not an active display. For example, if you re-display an EDIT session, you cannot edit the panel as if you were in an editing session.

   You can, however, change the values of application variables on a saved panel display, but this is not recommended. The application program may not be prepared to have the variables changed at that point.
4. GDDM must still be active to re-display a panel with a graphic area. If GDDM is not active (for example, if GRTERM has been invoked), the graphic area will be empty.

PROFILE (full screen mode only)

Tasks: Changing the Way Your Debugging Session Runs (page 21)

Function: PROFILE displays a panel containing the settings for various parameters that affect the way your debugging session runs. The current settings are shown for each parameter.

Abbreviation: None

Syntax

PROFILE

The profile panel will contain the following initial settings:

```
            VS FORTRAN INTERACTIVE DEBUG PROFILE
COMMAND ===>

                          CURRENT SETTING
                          ---------------
 STEP DELAY                    50            (100 UNITS = 1 SECOND)
 FREQUENCY COUNT DISPLAY       YES           (YES OR NO)
 MONITOR LINE NUMBERS          YES           (YES OR NO)
 LOG LINE NUMBERS              YES           (YES OR NO)
 OUTPUT REFRESH VALUE          50            (1 TO 9999)




 ENTER:
    SAVE      TO SAVE YOUR PROFILE SETTINGS.
    RESET     TO RETURN TO THE PREVIOUSLY-SAVED SETTINGS.
    END       TO RETURN TO THE MAIN PANEL WITH SPECIFIED PROFILE
               SETTINGS IN EFFECT.
```

Figure 42. Interactive Debug Profile Panel

The parameters displayed in the PROFILE panel are:

Step delay: Controls the pace of animation, measured in hundredths of a second.

Frequency count display: Indicates whether the statement execution counts will be shown in the source window.

Monitor line numbers: Indicates whether line numbers in the monitor window will be displayed.

Log line numbers: Indicates whether line numbers in the log window will be displayed.

Output refresh value: Indicates the number of output lines after which Interactive Debug will refresh the main debugging panel.

The following commands may be entered on the command line:

SAVE: saves the current settings in the profile.

RESET: restores the current settings to last saved settings, or the default settings if no previous save was done.

END or **RETURN:** changes the current settings without changing the profile settings. That is, unless a SAVE has been issued, these settings will remain in effect for only the duration of the current debugging session.

Usage Notes

1. PROFILE cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. Initially, the current setting for any of the parameters displayed will be the same as your profile setting.

PURGE

Function: PURGE terminates the output of a single Interactive Debug command after the output has been suspended by an attention interrupt. Subsequent commands in a command list are not affected. Following the PURGE command, resume execution by entering a null line.

Abbreviation: None

Syntax

```
PURGE
```

Usage Notes

1. PURGE cannot be used to stop output being produced as a result of a VS FORTRAN WRITE statement or a command in a static DEBUG packet.
2. PURGE cannot be used to stop output from the HELP command.
3. The terminal displays the results of commands more slowly than the processor produces these results. This may cause the processor to begin processing the command(s) following the one displayed on the terminal at the same time the null line (ATTN) was issued. In this case, PURGE may not have the desired results.
4. PURGE has no effect outside an attention exit.

Example

The following AT command list is being run:

```
at 100 (list a% set i=0%list b% where% go)
```

A contains 1000 elements. When A starts to be displayed, stop the display before its completion.

```
Press ENTER (or ATTN)
purge
```

This sequence suppresses the output of A. When a null line is entered, execution will resume with the next command in the command list (set i=0).

QUALIFY

Tasks: Referring to Statements or Variables in Other Program Units (page 54)

Function: QUALIFY changes or displays the current qualification. This command allows you to change the default qualification that determines which program unit any unqualified statement or variable references apply to.

Abbreviation: Q

Syntax

QUALIFY
    [*program-unit-name*]

program-unit-name
specifies the name of the main program or subroutine, or the name of a function subprogram.

Usage Notes

1. The QUALIFY command remains in effect until the next QUALIFY command, or until execution is resumed. When execution is resumed, the current qualification is reset to the executing program unit.
2. If QUALIFY is not entered, it is assumed that any Interactive Debug commands apply to the currently executing program unit (except for individually qualified operands).
3. QUALIFY command specified without the program unit parameter will display the name of the currently qualified program unit.
4. QUALIFY is not permitted as the command specified in an IF command.
5. To qualify an individual VS FORTRAN variable, place the variable name after the name of the program unit.

   ```
   list sub1.x
   ```

6. To qualify an individual statement identifier, place the number or statement label after the name of the program unit. The statement label must be preceded by a slash.

   ```
   at sub1./50
   ```

Examples

1. Display the value of all the variables in subroutine SUB1 while execution is suspended at a breakpoint in the main program.

   ```
   qualify sub1
   list *
   ```

2. Display the name of the currently qualified program unit.

   ```
   qualify
   ```

QUIT

Function: QUIT allows you to exit Interactive Debug.

Abbreviation: None

Syntax

```
QUIT
```

Usage Notes

1. If QUIT is issued following an attention interrupt, your program will be terminated. You may then issue certain Interactive Debug commands (for example, LIST) before returning to ISPF, CMS, or TSO. You must issue another QUIT to terminate the session.
2. Commands following QUIT in a command list are ignored.
3. The ISPF END command cannot be used to terminate a debugging session. The QUIT command must be used. This is designed to avoid accidentally terminating a debugging session with PF key 3.

RECONNECT

Tasks: Processing External Files (page 76)

Function: RECONNECT resets a file to its original (preconnected) condition. This may be necessary if you have used the CLOSE command or your program has run a CLOSE, and you wish to make it possible for the program to do additional I/O to the preconnected file.

Abbreviation: RECONN, RECONNEC

Syntax

RECONNECT
{*number* | [*qual.*]*integer-variable* | [*qual.*]*integer-array-element*}

number
is the number of the I/O unit associated with the sequential file on which the reconnect is to be performed.

[*qual.*]*integer-variable*
is the name of an integer variable in the VS FORTRAN program. This variable specifies the number of the I/O unit associated with the sequential file on which the reconnect is to be performed.

[*qual.*]*integer-array-element*
is the name of an element of an integer array in the VS FORTRAN program. This element specifies the number of the I/O unit associated with the sequential file on which the reconnect is to be performed.

Usage Notes

1. *number*, *integer-variable*, or *integer-array-element* must be specified; there is no default number.
2. This command may not be issued when I/O is currently active.
3. RECONNECT is only necessary if the OCSTATUS run-time option is in effect and you wish to allow your program to perform additional I/O on a file that has been closed, without running another OPEN.
4. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example, the following are valid:

```
ARY(J), ARY(3), ARY(J+3) or ARY(J-3)
```

Example

Reconnect the sequentially accessed external file associated with I/O unit 8.

```
reconnect 8
```

REFRESH (full screen mode only)

Function: REFRESH controls whether or not the Interactive Debug panel is completely refreshed when Interactive Debug panels are displayed.

If you have applications that do full screen I/O without using ISPF, there may be changes to the screen contents that ISPF is not aware of, and portions of the application display may remain on the screen. REFRESH is helpful in these situations.

Abbreviation: None

Syntax

```
REFRESH
     [ON|OFF]
```

ON
indicates that every display of an Interactive Debug panel should rewrite the entire screen.

OFF
indicates that ISPF does not need to rewrite portions of the screen that already seem to have the proper contents. This is the initial setting for REFRESH.

Usage Notes

1. REFRESH cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. When refresh is on, animation is less smooth because the entire screen is refreshed for each step. The screen may appear to flash each time.
3. Response time may increase for remote terminals when refresh is on.
4. Entering REFRESH without a parameter queries the status of REFRESH.

Example

Query the current status of REFRESH:

```
refresh
```

RESTART (full screen mode only)

Function: RESTART allows you to restart a debugging session in full screen mode without clearing the log file. The variable values are all cleared unless they are in dynamic commons. The VS FORTRAN program restarts at the first executable statement. Any new log information is appended to the existing log file.

Abbreviation: None

Syntax

```
RESTART
```

Usage Notes

1. RESTART cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. When RESTART is issued, breakpoints and the settings for WHEN conditions, HALT status, and AUTOLIST window, are reset. The TIMER status is turned off and all times are cleared.
3. In TSO, you can recompile a program unit using a split screen, then issue RESTART to restart the debugging session using the new object deck. (This is not possible in CMS.)
4. The AFFON and AFFIN files are re-read when you issue RESTART. If you have modified these files (for example, in a split-screen edit session), the new files are used.

Example

Restart a debugging session in full screen mode, but retain the current log file.

```
restart
```

RESTORE (full screen mode only)

Function: RESTORE is used to restore the source window to the last point of execution.

Abbreviation: RES

Syntax

```
RESTORE
```

Usage Notes

1. RESTORE cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. If the source window is open and the listing is available, the source window is restored to the point of execution with the current statement highlighted.
3. If the currently qualified program unit is different than the program unit where execution was suspended, the currently qualified program unit is reset.

RETRIEVE (full screen mode only)

Function: RETRIEVE is used to re-display up to the last twelve Interactive Debug commands entered on the command line.

RETRIEVE also displays commands that are partly specified on the command line and partly specified by a PF key. For example, if you type PAGE on the command line and press PF8 (initially set to the DOWN command), and then enter RETRIEVE, the following appears on the command line:

```
down page
```

Abbreviation: None

Syntax

RETRIEVE

Usage Notes

1. RETRIEVE cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. Commands issued before a RESTART command cannot be retrieved.
3. If you are using ISPF Version 2 Release 3 or later, RETRIEVE will behave differently:
   - RETRIEVE will re-display Interactive Debug as well as ISPF commands.
   - Commands entered by PF key are not re-displayed. For example, if you type PAGE on the command line and press PF8 (initially set to the DOWN command), and then enter RETRIEVE, the following appears on the command line:

     ```
     page
     ```

   - The number of commands that RETRIEVE can re-display depends on the size of the stack allocated by your system programmer.

REWIND

Tasks: Processing External Files (page 76)

Function: REWIND positions a sequentially accessed external file at the beginning of the first record. Its usage is similar to that of the REWIND statement in the VS FORTRAN Version 2 language, allowing you to move to the beginning of the file.

Abbreviation: REW

Syntax

REWIND
{number | [qual.]integer-variable | [qual.]integer-array-element}

number
is the number of the I/O unit associated with the sequential file that is to be rewound.

[qual.]integer-variable
is the name of an integer variable in the VS FORTRAN program. This variable specifies the number of the I/O unit associated with the sequential file that is to be rewound.

[qual.]integer-array-element
is the name of an element of an integer array in the VS FORTRAN program. This element specifies the number of the I/O unit associated with the sequential file that is to be rewound.

Usage Notes

1. *number*, *integer-variable*, or *integer-array-element* must be specified; there is no default number.
2. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example,

   ```
   ARY(I), ARY(3), ARY(I+3) or ARY(I-3)
   ```

   are valid forms.
3. This command may not be issued when I/O is currently active.
4. VS FORTRAN Version 1 and VS FORTRAN Version 2 support multiple files under the same I/O unit. The REWIND command sets the VS FORTRAN file name to the first in the sequence of files for the specified I/O unit. For example, if you were currently processing file FT08F003 on I/O unit 8, and entered:

   ```
   rewind 8
   ```

   I/O unit 8 would be connected to file FT08F001, which would be positioned at the beginning of the first record.

Example

Rewind the sequentially accessed external file associated with logical unit 4 so that it may be rewritten.

```
rewind 4
```

RIGHT (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Using Cursor-Sensitive Commands (page 20)

Function: RIGHT scrolls the contents of a window so that columns to the right of those currently displayed in the window can be seen.

Abbreviation: None

Syntax

RIGHT
[*number* **| PAGE | HALF | CSR | DATA | MAX]**

number
is the number of columns to scroll right, from 1 to 9999.

PAGE (or P)
scrolls right by the number of columns in the window.

HALF (or H)
scrolls right by half the number of columns in the window.

CSR (or C)
scrolls right by PAGE, unless the cursor is in the window, in which case the window is scrolled right by the appropriate number of columns to place the cursor at the edge of the window.

DATA (or D)
scrolls right by PAGE-1 number of columns. If only one column is visible, the scrolling is equivalent to that of PAGE.

MAX (or M)
scrolls right so that the rightmost column of the window will contain the rightmost column of the contents for the window.

Usage Notes

1. RIGHT cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. RIGHT is cursor-sensitive. The window that is scrolled is determined from the cursor position and the windows currently open.
3. If an operand is not specified with RIGHT, the scrolling amount is taken from the SCROLL ===> field on the main debugging panel.

Example

Scroll right by the number of columns visible in the window.

```
right p
```

SEARCH (full screen mode only)

Tasks: Using Cursor-Sensitive Commands (page 20)

Function: SEARCH searches a window for a given character string.

Abbreviation: None

Syntax

```
SEARCH
     /string[/]
```

string
specifies a character string to be searched for. The search is not case sensitive, so your string can be found in any combination of upper or lower case.

You can use any non-blank character as a delimiter. A slash (/) is shown in the syntax above.

The initial and final delimiter must be the same character. The final delimiter is required if the search string contains trailing blanks. In all other cases, it is optional.

Usage Notes

1. SEARCH cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. SEARCH is a cursor-sensitive command. After typing SEARCH followed by the string on the command line, position the cursor in the desired window and press ENTER.
3. The search is performed starting at the top line displayed on the screen. However, if a search is done immediately after a search, the second search begins at the location of the last-found search argument.
4. When searching the log, only the last 1000 lines of the log are available.
5. If the search reaches the bottom of the window and wraps around to the beginning of the window, the message "SEARCH CONTINUED FROM THE TOP OF AREA" is displayed in the upper right corner of the screen.
6. If you search without a parameter, the most recent character string used as a target is searched for again.
7. If you have

   ```
   MOVECURS;
   ```

   assigned to a PF key, you can type the SEARCH command on the command line. Then press the MOVECURS PF key to move the cursor to its previous position in the main debugging panel, and press ENTER to search the window.
8. If the search string contains double-byte characters and the DBCS option is in effect, highlighting of the search string is suppressed, but the cursor is still positioned in the window at the beginning of the string.

Example

Search for the character string VM/CMS, using a question mark as the delimiter.

```
search ?VM/CMS?
```

SET

Function: SET changes the value of a variable, array, array element, or group of array elements.

Abbreviation: S

Syntax

```
SET
     [qual.]name=value[,value...]
```

qual.
specifies a program unit name to temporarily override the current qualifier for the prefixed name only.

name
specifies the name of a variable, array, or array element.

value
is the value to be assigned to a single variable or single array element. A group of values (separated by commas or blanks) can be assigned to an entire array or part of an array. A value can also be another qualified variable name or array element, and can be prefixed by a numeric replication factor.

Usage Notes

1. Valid SET command assignments for the different types of names are shown in Figure 43. All names can be qualified.
2. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example,

   ```
   ARY(I), ARY(3), ARY(I+3) or ARY(I-3)
   ```

   are valid forms.
3. No arithmetic operations (except negation) are allowed in the value assignments; for example, SET A=B+4 is not allowed.
4. Upper- and lowercase character constants may be entered at the terminal. They must be enclosed in single quotation marks when entered, but the enclosing single quotation marks are removed before the assignment is performed. If a single quotation mark is to be assigned as part of the character constant, two single quotation marks must be entered; for example:

   ```
   set c=''''
   ```

   sets C to a single quotation mark. Character strings are truncated or extended with blanks to match the length of the receiving character variable or array element.

Name	Set	Type of Value	Example
Real, Integer, or Complex scalar	=	Another scalar variable	ALPHA=BETA ALPHA=-BETA
		An array element	ALPHA=A(3) ALPHA=-A(3)
		A constant	NUM=7
Logical	=	Another logical variable	LOG1=LOG2
		An array element	LOG1=LOG(2)
		A logical value	LOG=.TRUE.
Character	=	Another character variable	CHAR1=CHAR2
		An array element	CHAR1=CHAR(2)
		A character constant	MSG='HELLO'
		A substring	A(1:3)='ABC'
Array element	=	Another array element	A(4)=B(1) AR(2,2)=-AR(5,5)
		A scalar variable	C(7)=RATE C(8)=-TIME
		A constant	D(I,J)=0.0
Contiguous array elements	=	Value,value,... (Values can be variables, array elements, or constants; multiple assignments of a value can be entered as n*value.)	A=3*1.0,4*0.0, 7.2,5.,ACCL, 8.5E9 B(J,K)='C', 3*'Q',2*,'X'

Figure 43. Valid SET Command Assignments

5. In assigning values to contiguous array elements, values may be repeated using the notation n*value. For example,

   ```
   set ary=10*4
   ```

   sets the first 10 elements of the array ARY to 4.

6. In assigning values to contiguous array elements, elements can be omitted by using the asterisk notation with no value following the asterisk. For example, a single omission is entered as 1*, and a multiple omission might be entered as 3* (this would leave three successive elements of the array unchanged).

7. Substring notation is permitted with string variables.

8. On the right side of the "=" sign, an array reference can have subscripts that exceed the bounds of the dimensions. A warning message will be issued except for the special case where only the last dimension is exceeded, and that dimension is "*" or "1."

9. An assumed size array is not set unless a specific element or range of elements is specified. An assumed size array is an array with the last upperbound declarator specified as an asterisk (*). Results are unpredictable if you SET array elements beyond the end of the original array.

10. Dummy arguments can only be used or set when the program unit in which they are defined is active. (Note that a program unit is not yet active when suspended at entry.) If this rule is violated, you will get an error message.

11. Variables in a dynamic common can only be used or set after the program unit used to qualify the variable has been activated at least once. If this rule is violated, you will get an error message.

12. When setting a variable, results are unpredictable if:

- More values are specified than will fit in an assumed-size array or an array whose last dimension is "1."
- The right hand side contains an array reference whose subscripts are not all within the array dimensions. You will get a warning message in this case.
- The right-hand side of the statement contains an inaccessible variable. You will get an error message in this case.

Examples

1. Change the values of several variables.

```
set jt=8.9e+7
set m=-int
set a(3)=b(5)
set c(i,2)=4.1
set d(10)=xray
set a(i-1,3)=b(4,j+6)
set main.x=subl.x
set quote='he said: "bye, bye."'
```

2. Set the ten elements of array DH to 0, 0, 0, .666, .21E-08, 1.0, 0, 0, 0, 0.

```
set dh=3*0.0,.666,.21e-8,1.0,4*0.0
```

3. Set the second element of array DATA to 0, leave the third element unchanged, set the fourth through eighth elements to 1.0, and leave all other elements unchanged.

```
set data(2)=0,1*,5*1.0
```

4. Set the third through the fifth characters of character string E423 to 'ABC'.

```
set e423(3:5)='ABC'
```

SIZE (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Changing the Window Configuration (page 21)

Function: SIZE is used to re-size the windows on the main debugging panel. Cursor position is used to determine the size of the window specified by SIZE:

- If the cursor is in the specified window, the window is reduced to the cursor position.
- If the cursor is not in the specified window, the window is enlarged to the cursor location.

The size of the other windows are adjusted accordingly.

Abbreviation: None

Syntax

```
SIZE
     [SOURCE | MONITOR | LOG]
```

SOURCE | MONITOR | LOG
specifies that either the source, monitor, or log window is to be sized.

Usage Notes

1. SIZE cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. The minimum window height is one row and the minimum window width is seven columns.
3. To save re-sized windows, specify WINDOW SAVE (page 206) on the command line after using SIZE.
4. If SIZE is specified with no parameters, the window border that is closest to the current cursor position will move to that position.

STEP

Tasks: Animating the Execution of Your Program (page 73)

Function: STEP runs one or more FORTRAN statements and then gives control back to you. STEP is similar to a series of NEXT and GO command pairs.

In full screen mode, STEP execution is automatically *animated* if the source listing is available to Interactive Debug. Animated execution means that the source window is refreshed at each statement boundary where a hook exists, and the current statement is highlighted. Highlighting is determined by the COLOR settings. The monitor window will also be updated, or *refreshed* as necessary at each debugging hook.

Abbreviation: ST

Syntax

STEP
    [*number***]**

number
specifies the maximum number of hooked statements to be run before execution stops. The number must be a positive integer. The default is one.

Usage Notes

1. To change the pace of animation, use the PROFILE command (page 170). When the profile panel is displayed, change the STEP DELAY field.
2. The STEP command itself causes your program to resume execution. You do not need to use a GO command.
3. If any of the following conditions occur, STEP execution will terminate before the STEP count runs out:
   - A breakpoint is encountered.
   - A WHEN condition is satisfied.
   - The HALT status is satisfied.
   - An error condition is detected.
   - Terminal input occurs with TERMIO IAD in effect.
   - An attention is issued.
   - The program terminates.
4. If STEP processing ends after finding one of the above conditions or after reaching the end of the step count, you cannot resume STEP. You must issue a new STEP command.
5. If a STEP command is terminated because of a terminal READ, a NEXT will be forced at the next hooked statement.
6. If STEP is issued within a command list, the remainder of the list is ignored.
7. STEP counts only those statements that have debugging hooks. Thus, STEP may run many statements before stopping. Statements that have been collapsed and statements not included in the AFFON statement list are not counted and execution does not stop.

8. STEP is not permitted after the VS FORTRAN program has terminated, or while a READ is pending. If issued in an error exit, standard corrective action is taken.
9. The service routine CPUTIME cannot be used while animation is in progress. For more information on CPUTIME, see *VS FORTRAN Version 2 Language and Library Reference*.

Example

Run the next 12 hooked statements before suspending execution:

```
step 12
```

SYSCMD

Tasks: Using System Commands (page 78)

Function: SYSCMD runs system commands during an Interactive Debug session. SYSCMD can also be included in a command list and on the IF command.

Abbreviation: SYS, CMS, TSO

Syntax

SYSCMD
    [*system-command*]

system-command
is a CMS or TSO system command to be run.

Usage Notes

1. Caution should be observed when issuing commands that would cause the currently executing program to be overlaid. For example, a CMS LOAD command with the CLEAR option could cause Interactive Debug and the VS FORTRAN application program to be erased from storage.
2. (CMS) If the system command is omitted, the standard CMS SUBSET will be entered. In this mode, CMS commands may be issued and Interactive Debug will not regain control until the RETURN command is issued. CMS commands issued in this mode (or specified with the SYSCMD command) are limited to those commands allowed in CMS SUBSET mode.
3. (TSO) If the system command is omitted, a special command entry mode will be entered. Interactive Debug will produce the message,

   ```
   ENTER A TSO COMMAND OR A NULL LINE
   ```

   In this mode, TSO commands may be issued and Interactive Debug will pass them along to TSO until a null line is entered.
4. In order to use SYSCMD in batch mode on MVS, it is necessary to run a TSO Terminal Monitor Program (TMP).
5. In batch mode, the system command must be specified, and must not be a command that requires interaction. Interactive Debug cannot guard against system commands that require interaction during a batch session; this is your responsibility.

Examples

1. (CMS) List the files that have been allocated:

   ```
   syscmd q filedef
   ```

2. (TSO) List the data sets that have been allocated:

   ```
   syscmd listalc status
   ```

TERMIO

Tasks: Entering Terminal Input (page 78)

Function: TERMIO allows you to select the I/O routines that you want to use for terminal I/O for your VS FORTRAN program. You can select either the Interactive Debug I/O routines, or the VS FORTRAN library routines.

You can also use TERMIO to send a copy of batch mode output to a user as message text. To query the current settings, enter TERMIO with no operands.

Abbreviation: None

Syntax

```
TERMIO
     [IAD | LIBRARY]
     [MSG [(userid)] | NOMSG]
```

IAD | LIBRARY

IAD indicates that terminal I/O is to be performed using the Interactive Debug I/O routines. The Interactive Debug I/O routines combine input and output from the VS FORTRAN program with Interactive Debug input and output. You must precede terminal input with a percent sign (%) to distinguish it from Interactive Debug commands.

This is the initial setting.

In full screen mode or batch mode, terminal input and output are included in the log file. This includes library error messages.

LIBRARY indicates that terminal I/O is to be performed using the VS FORTRAN library routines. The library I/O routines cause output from the VS FORTRAN program to be displayed as it would be if the program were not being debugged.

Terminal input and output are not included in the log file. In full screen mode, a request for terminal input causes the screen to be cleared and the keyboard to be unlocked. Terminal output is written on a blank screen.

MSG [(*userid*)] | NOMSG

In batch mode only, MSG indicates that a copy of each line of Interactive Debug input and output is to be sent to the specified or defaulted user ID as message text.

userid specifies the user ID to which message text is to be sent. If this operand is omitted, it defaults to the previously established user ID (if one exists), or to the submitter's user ID if available. The operand is required if the submitter's user ID is not available and no user ID has been previously established.

The default user ID is obtained from the JOB card information in MVS, if available. In CMS, no default user ID is available.

NOMSG specifies that Interactive Debug input and output are not to be copied as message text.

Usage Notes

1. When no operands are specified, the current TERMIO setting is displayed.
2. This command does not affect I/O operations other than those requested by the VS FORTRAN application program. This implies that, if Interactive Debug is running in ISPF, a specification of LIBRARY will cause any application program terminal I/O operation to occur in line mode. At the completion of the I/O operation, full screen operation will resume.
3. When running in batch mode on MVS, a VS FORTRAN program has no real terminal inputs or outputs. You can simulate these terminal inputs and outputs by specifying one or more units on the DEBUNIT run-time option for your VS FORTRAN program. If you do not specify the DEBUNIT option, the IAD/LIBRARY operand has no effect on program I/O.
4. Whenever character data is entered, Interactive Debug I/O routines change it to uppercase. If mixed-case input is required (and supported by the host system), the library I/O routines must be used.
5. If output cannot fit on one line in whatever mode Interactive Debug is operating, it is split across multiple lines. The lengths of these lines are 60 characters if Interactive Debug I/O routines are used. When the library routines are used, I/O operations produce the same results as would be produced if the program were run without Interactive Debug.
6. When TERMIO IAD is in effect, any continuation line that begins with leading blanks must be prefixed with a quotation mark ("). The quotation mark will not be passed to the program.

Examples

1. Specify that Interactive Debug routines are to be used for subsequent I/O requests of the VS FORTRAN program.

   ```
   termio iad
   ```

2. Display the current setting of the terminal I/O mode.

   ```
   termio
   ```

3. Specify that Interactive Debug input and output are to be echoed to user ID "SMITH" (in batch mode).

   ```
   termio msg(smith)
   ```

TIMER

Tasks:

Program Unit Timing (page 66)
Vector Tuning Assistance (page 67)

Function: TIMER activates, deactivates, and resets the timing of program units or DO loops. Timing must be activated in order for a program unit or DO loop to be timed when invoked. The LISTTIME command displays the timing information.

Abbreviation: None

Format 1

Syntax for Timing a Program Unit

```
TIMER
    { * | program-unit-name | (program-unit-name-list) }
    [ON | OFF | RESET ]
```

Format 2

Syntax for Timing a DO Loop

```
TIMER
    { [qual.]number: [qual.]number | [qual.]* | (list) | *:* }
    DOLOOP
    [ON | OFF | RESET ]
```

*

specifies that the command applies to all debuggable program units.

program-unit-name
specifies an individual program unit.

program-unit-name-list
specifies a list of program unit names. Enclose the list in parentheses and separate entries with commas or blanks.

[*qual.*]
specifies a program unit name to temporarily override the current qualifier for the prefixed operand only.

[*qual.*]*number*
specifies the statement label, ISN, or sequence number of the DO statement for a DO loop for which timing is to be activated, deactivated, or reset. Qualification is optional. A statement label must be prefixed with a slash (/).

[*qual.*]*number*:[*qual.*]*number*
specifies a range of statements in the program for which the timing for all DO loops in the range is activated, deactivated, or reset. (A DO loop is in the range if the DO statement of the loop is in the range. All the statements

of the loop do not have to be in the range. The starting and ending statement identifiers.do not have to be DO statements.) Qualification is optional. If the second qualifier is specified, it must be the same as that specified or defaulted for the first qualifier.

*[qual.]**
specifies that the timing of all DO loops in the specified or currently qualified program unit is activated, deactivated, or reset.

list
specifies a list of individual DO loop specifications. Enclose the list in parentheses and separate entries using commas or blanks.

.
specifies that timing for all DO loops in the program is activated, deactivated, or reset.

DOLOOP
indicates that timing for DO loops, not program units, is to be measured.

ON | OFF | RESET
specifies that timing is to be activated, deactivated, or reset to zero for the specified program units (format 1) or DO loops (format 2). OFF is the initial setting for TIMER.

Usage Notes

1. Timing is cumulative. Timing values are only reset by specifying RESET.
2. The time for a routine is measured beginning at the entry point and ending at the exit. If a call is made to another routine for which TIMER is on, the time spent in the second routine (and lower-level routines) is not included in the measurement for the first routine. However, time spent in called (and lower-level) non-timed routines *is* included in the measurement for the calling routine. For example, if program A calls program B and you do not want the time in B to be included in the timing of A, you must specify: `TIMER (A,B) ON`. Program B must be debuggable.
3. In MVS, timing measurements will be incorrect if your program uses the STIMER macro, or if it uses a system service that calls STIMER. This includes the BTAM OPEN and LINE OPEN operations, and Dynamic Allocation.
4. Timing for very small routines may be erratic.
5. The timing information provided by Interactive Debug includes overhead caused by the debugging hooks in your program. Thus, the timing information is not a completely accurate representation of the time it takes to run *without* Interactive Debug.

   However, the overhead is consistent. Therefore, you can use program unit and DO loop timing to measure changes to run times due to vector tuning efforts.

   To get the most accurate timing information for program units, place hooks only at entry and exit of the program unit. This can be done by specifying this in the AFFON file:

   ```
   (all) entry
   ```

Likewise, to get the most accurate timing information for DO loops, specify this in the AFFON file:

```
(all) * doloop
```

6. The ENDDEBUG command turns timing off for all program units and DO loops.
7. TIMER is not permitted after the VS FORTRAN program has terminated.
8. Timing is measured separately for each entry point into the program unit.
9. The service routine CPUTIME cannot be used while timing is in progress. For more information on CPUTIME, see *VS FORTRAN Version 2 Language and Library Reference*.
10. Format 2 only:
    - DO loop timing cannot be performed on unanalyzable loops.
    - The program information for a program unit must have been successfully obtained at initialization of the debugging session in order to time DO loops in the program unit.
    - The DO statement for a DO loop must have been included in the AFFON statement restriction list for the program unit in order to time the loop.
    - If a DO loop has been distributed into several loops due to vectorization, then all of the distributed loops are timed and the time is presented as if the DO loop was not distributed.

Examples

1. Turn timing on for program units MAIN and SUB2:

```
timer (main,sub2)
```

2. Reset timing to zero for program unit MAIN:

```
timer main reset
```

3. Turn timing off for all debuggable program units:

```
timer * off
```

4. Activate timing for all DO loops in the program:

```
timer *.* doloop on
```

5. Reset timing for DO loops in the ISN range 12 to 44 in program unit PLACES.

```
timer places.12:44 doloop reset
```

TRACE

Tasks:

Specifying Output to a Print File (page 42)
Tracing Program Execution (page 71)

Function: TRACE starts or stops tracing of the flow of the program as it runs. You can trace each transfer of control in the program, trace just entries to and exits from debuggable subroutines, or determine the current trace status.

Abbreviation: T

Syntax

```
TRACE
     [GOTO | ENTRY | OFF]
     [PRINT]
```

GOTO | ENTRY | OFF
GOTO specifies that a record of each apparent branch taken within the program is to be created. GOTO produces a listing showing a statement label or statement identifier for the origin and destination of each transfer made, including entries to and exits from debuggable subroutines.

ENTRY specifies that only a record of entries to and exits from debuggable subroutines is to be produced.

OFF turns off tracing that you previously initiated.

PRINT
specifies that output be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. Tracing continues until turned off, or altered by another TRACE command, or an ENDDEBUG command is issued.
2. TRACE operations apply to all debuggable program units.
3. TRACE GOTO issues a trace message if two consecutively executed debugging hooks are not on consecutively stored statements. This also occurs if statements have been collapsed or vectorized, or have no debugging hooks.
4. TRACE is not permitted after the VS FORTRAN program has terminated.
5. If no operand is specified, TRACE issues a message describing the current trace status.

Examples

1. Trace each program transfer, and have the trace output sent to the print data set instead of to the terminal.

   ```
   trace goto print
   ```

2. Discontinue tracing entirely.

   ```
   trace off
   ```

3. Trace only the calls to and returns from subroutines and functions.

   ```
   trace entry
   ```

UP (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Using Cursor-Sensitive Commands (page 20)

Function: UP scrolls the contents of a window so that lines above those currently displayed in the window can be seen.

Abbreviation: None

Syntax

UP
[*number* **| PAGE | HALF | CSR | DATA | MAX]**

number
is the number of lines to scroll up, from 1 to 9999.

PAGE (or P)
scrolls up by the number of lines in the window.

HALF (or H)
scrolls up by half the number of lines in the window.

CSR (or C)
scrolls up by PAGE, unless the cursor is in the window, in which case the window is scrolled up by the appropriate number of lines to place the cursor at the edge of the window.

DATA (or D)
scrolls up by PAGE-1 number of lines. If only one line is visible, the scrolling is equivalent to that of PAGE.

MAX (or M)
scrolls up so that the uppermost line of the window will contain the top-of-data marker (see page 15).

Usage Notes

1. UP cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. UP is cursor-sensitive. The window that is scrolled is determined from the cursor position and the windows currently open.
3. If an operand is not specified with UP, the scrolling amount is taken from the SCROLL ===> field on the main debugging panel.

Example

Scroll up by half the number of lines visible in the window.

```
up half
```

VECSTAT

Tasks: Vector Tuning Assistance (page 67)

Function: VECSTAT is used to activate, deactivate, or reset DO loop length and stride recording.

Abbreviation: VEC

Syntax

```
VECSTAT
     { [qual.]number[:[qual.]number] | [qual.]* | (list) | *.* }
     [ON | OFF | RESET]
```

[qual.]number
is the statement label, ISN, or sequence number of a DO statement for a DO loop in the specified or currently qualified program unit. *qual.* specifies a program unit to temporarily override the current qualifier for the prefixed operand only.

[qual.]number:[qual.]number
specifies a range of statements in the program for which vector information for all DO loops in the range is to be recorded. (A DO loop is in the range if the DO statement of the loop is in the range. All the statements of the loop do not have to be in the range. The starting and ending statement identifiers do not have to be DO statements.) Qualification is optional. If the second *qual* is specified, it must be the same as that specified or defaulted for the first *qual.*

*[qual.]**
indicates that vector information is to be recorded for all DO loops in the specified or currently qualified program unit.

list
is a list of individual DO loop specifications. Enclose the list in parentheses and separate entries with commas or blanks.

.
indicates that vector information for all DO loops in the program is to be recorded.

ON | OFF | RESET
ON activates length and stride recording.

OFF deactivates length and stride recording.

RESET clears length and stride information. Length and stride averages and number of invocations are set to zero. Also, length and stride recording is activated for the DO loops.

Usage Notes

1. Length and stride statistics can only be gathered for DO loops that were analyzable by the compiler.
2. The program information for a program unit must have been successfully obtained at initialization of the debugging session in order to record length and stride statistics for DO loops in the program unit.
3. The DO statement for a DO loop must have been included in the AFFON statement restriction list for the program unit in order to record vector information.
4. A VECSTAT command with a specification list will run if one or more of the DO statement identifiers in the list are valid.
5. If a DO loop has been distributed into several loops due to vectorization, then length and stride information is collected for all of the distributed loops, and presented as if the DO loop was not distributed.

Examples

1. Activate length and stride data recording for all DO loops in the program.

```
VECSTAT *.* ON
```

2. Reset the length and stride data for DO loops in function FUN1 at ISN 23, and in subroutine SUB1 at ISN 44.

```
VECSTAT (FUN1.23, SUB1.44) RESET
```

WHEN

Tasks: Controlling Program Execution (page 58)

Function: WHEN allows you to suspend execution every time a particular condition is met. You can define a condition and supply its name, or restart monitoring of a previously defined condition. The condition is tested at all statements with debugging hooks.

Abbreviation: WN

Syntax

WHEN
condition-name
[(*condition*) **|** *variable***]**

condition-name
identifies the condition. *condition-name* must be 1 through 4 alphameric characters, with the first character alphabetic. *condition-name* is only a name; it is not to be confused with the definition of the condition.

(*condition*)
defines a condition to be monitored. The condition itself appears only in the initial WHEN command. The condition must be enclosed in parentheses.

Only scalars and single array elements are allowed in the condition expressions. They can be explicitly qualified; for example, `sub1.x=4.0`

It can be either a relational or a logical condition:

- Relational condition: a signed or unsigned variable or array element or constant, followed by a *relational operator*, followed by another signed or unsigned variable or array element or constant.

  There are six relational operators that can be used to define test conditions between scalar variables, array elements or constants.

  = or .EQ.
  ¬ = or .NE.
  > or .GT.
  < or .LT.
  > = or .GE.
  < = or .LE.

- Logical condition: a logical variable or a logical array element, optionally preceded by the negation operator (¬ or .NOT.). No other operators are permitted. If the value being tested is a logical variable or a logical array element, the negation operator can be applied in the condition definition. For example:

```
WHEN HITON (¬ LOGVAR)
```

variable
specifies the name of a variable or an array element to be monitored for any change in value. Only the name is specified; no parentheses to enclose the name are used. The name can be explicitly qualified.

Usage Notes

1. The condition itself is only defined once. At that time, a condition name must be supplied. Subsequent WHEN commands referring to that condition should contain only the condition name.
2. Monitoring remains in effect after a condition is satisfied. To turn the condition off, issue an OFFWN command, naming the condition.
3. After being turned off by an OFFWN, condition monitoring can be reactivated by reissuing WHEN and specifying only the condition name.
4. WHEN conditions are evaluated at each debugging hook in each debuggable program unit. If you use a variable subscript, it is reevaluated each time the condition is tested. The array element actually tested, therefore, depends on the subscript value at that moment.
5. You can redefine an existing WHEN condition by entering a new WHEN command with the same condition name and a new condition definition.
6. Variables can be prefixed by a program unit qualifier to override the current qualification (at the time the WHEN command is issued). For example,

   ```
   WHEN COND (MAIN.A .LT. SUB1.B)
   ```

7. When referring to array elements, the subscripts must use simple arithmetic expressions no more complex than the form "variable plus (or minus) a constant." For example,

   ```
   ARY(I), ARY(3), ARY(I+3) or ARY(I-3)
   ```

   are valid.

8. In the relational condition form of the WHEN command, for example, WHEN TEST (A .GT. B):
   - When either variable or constant is a logical, character, or complex type, both must be of that same type, and a sign preceding the variable or constant is not permitted.
   - When either variable or constant is a logical or complex type, only the relational operators .EQ. and .NE. (or = and ¬ =) may be used.
   - When character variables or constants of unequal length are compared, the shorter is considered to be extended with blanks during the comparison.
9. When an unparenthesized variable name is specified as the condition (for example, WHEN TEST NAME) and the variable is a character variable:
   - The length of the character variable is determined at the time the WHEN command is issued.
   - If the character variable subsequently changes length (perhaps because it is a parameter to the subroutine being monitored), the old and new values are compared by effectively extending the shorter one with blanks.
   - If a change in value is found, the new value is preserved, either extended or truncated, using the size of the variable at the time the WHEN command was issued.

- The current length and value of the variable can be captured at any time by issuing the WHEN command with just the condition name (for example, WHEN TEST). It is not necessary to issue an OFFWN command first or to state the variable name again. (The WHEN command can be embedded in an AT command list to automate this process.)

10. When a specified condition is met, messages will be displayed. These indicate the condition name that was satisfied, and where execution is currently suspended.
11. If you refer to undefined variables (such as dummy arguments in an inactive subprogram), you will receive an error message at each statement where the condition is tested. To avoid these messages, you can use OFFWN in an AT EXIT command list, and WHEN to reactivate it in an AT ENTRY command list.
12. WHEN is not permitted after the VS FORTRAN program has terminated.

Examples

1. Start monitoring variable MIKE to see if it changes. Call the condition (the change to variable MIKE) OLD.

```
when old mike
```

2. Define a condition named OVFL as an array element A(1,5) greater than 3.5E10. Start monitoring it.

```
when ovfl (a(1,5) .gt. 3.5e+10)
```

3. If the monitoring from Example 2 has been turned off with an OFFWN command, restart monitoring condition OVFL.

```
when ovfl
```

WHERE

Tasks:

Specifying Output to a Print File (page 42)
Tracing Program Execution (page 71)

Function: WHERE identifies the statement at which execution is suspended.

Abbreviation: W

Syntax

```
WHERE
    [TRBACK]
    [FLOW]
    [PRINT]
```

TRBACK
specifies a traceback showing the names of all program units that are currently active.

FLOW
specifies a trace of the last 10 program transfers that were run.

PRINT
specifies that output is be sent to a print file (AFFPRINT) instead of the terminal.

Usage Notes

1. If WHERE is used following an attention interrupt, it can only be used to identify the current statement. Any parameters are ignored.
2. If there are no active program units, WHERE TRBACK indicates that no subroutines have been called.
3. The current statement identified by the WHERE command has not yet been run.
4. An automatic WHERE command is forced at the first debugging hook found. This is usually the first executable statement in the main program.

Examples

1. Find out where you are after execution was suspended by an attention interrupt.

   ```
   where
   ```

2. Find out the sequence of control transfers that led to the current breakpoint.

   ```
   where trback flow
   ```

3. Conditionally indicate that a breakpoint has been reached, even though the AT command list does not cause execution to be suspended. The WHERE command will only be run if A is less than or equal to 4.8.

   ```
   at 120 (if (a .gt. 4.8) go%where%list a%go) nonotify
   ```

WINDOW (full screen mode only)

Tasks:

Changing the Way Your Windows Look (page 18)
Using Cursor-Sensitive Commands (page 20)
Changing the Window Configuration (page 21)

Function: WINDOW allows you to:

- Change and save the window configuration of the debugging session
- Open or close a specified window

Abbreviation: None

Syntax for Changing the Window Configuration

```
WINDOW
```

Syntax for Saving the Window Configuration

```
WINDOW
     SAVE
```

Syntax for Opening or Closing a Window

```
WINDOW
     {OPEN | CLOSE}
     {SOURCE | MONITOR | LOG}
```

WINDOW
allows you to change the window configuration. After you type WINDOW with no operands on the command line and press ENTER, the Window Configuration Selection Panel will appear. The panel initially looks like Figure 44 on page 207.

WINDOW SAVE
saves the current window configuration.

WINDOW CLOSE {SOURCE | MONITOR | LOG}
closes the specified window.

WINDOW OPEN {SOURCE | MONITOR | LOG}
opens the specified window.

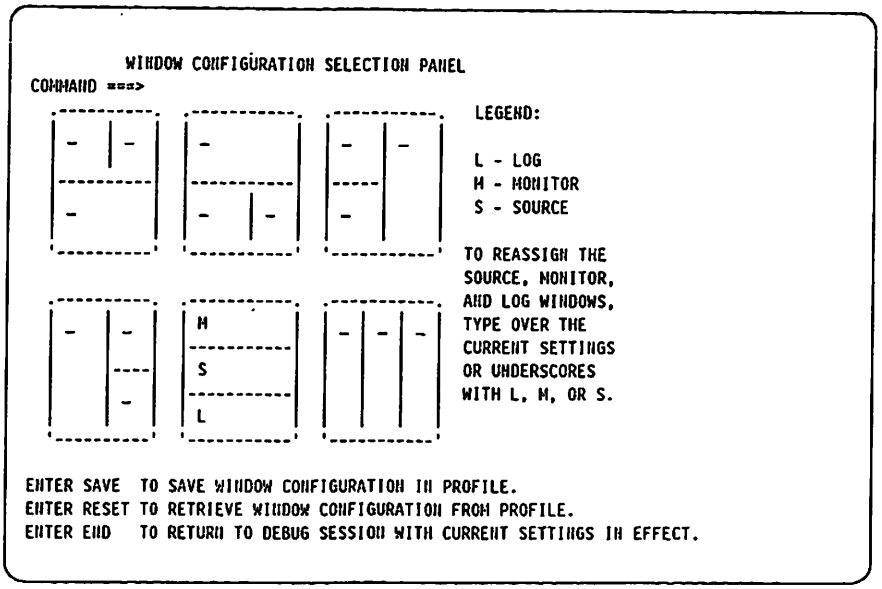

Figure 44. Interactive Debug Window Configuration Selection Panel

From this panel, the following commands may be entered on the command line:

SAVE: saves the current window configuration in the profile.

RESET: restores the current window configuration to the last saved configuration, or to the initial configuration if no previous save was done.

END or **RETURN:** changes the current configuration without changing the last saved configuration. That is, unless a SAVE has been issued, this configuration will remain in effect for only the duration of the current debugging session.

Usage Notes

1. WINDOW cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. Besides specifying WINDOW CLOSE with either SOURCE, MONITOR, or LOG, you can also close a window by typing WINDOW CLOSE on the command line, positioning the cursor in the desired window, and pressing ENTER.
3. If you close a window, and then specify WINDOW OPEN with no operands, you will open the window you just closed.
4. When changing the window configuration, only one configuration may be selected at a time, and each object type can be assigned to only one window area.
5. You can use WINDOW SAVE to save windows re-sized with the SIZE command (page 188).

ZOOM (full screen mode only)

Tasks: Using Cursor-Sensitive Commands (page 20)

Function: ZOOM allows you to toggle between displaying one window (either the source, monitor, or log window) on the entire screen, and displaying a configuration of windows as defined by the WINDOW command.

Abbreviation: None

Syntax

```
ZOOM
     [SOURCE | MONITOR | LOG]
```

SOURCE | MONITOR | LOG
specifies either the source, monitor, or log window to be allocated to the entire screen.

Usage Notes

1. ZOOM cannot be issued in: a command list, an IF command, an attention exit, or a restart file.
2. ZOOM is a cursor-sensitive command. That is, if ZOOM is specified with no operands, the hierarchy of windows for cursor-sensitive commands will determine which window will be affected.
3. To return to the configuration of windows specified by the WINDOW command, enter ZOOM with no operands.
4. When one window is displayed in the entire screen, default cursor-sensitive commands apply to that window.
5. Any subsequent WINDOW OPEN or WINDOW CLOSE command will cancel the ZOOM mode.

Appendix A. Interactive Debug Messages

Each Interactive Debug message can be identified by its own message number. It is composed of:

1. The prefix AFF
2. A 3-digit number
3. A suffix, which indicates the level of the message:
   - I - informational
   - W - warning
   - E - error
   - A - action required

You can control whether or not the message numbers are displayed. In CMS, use the SET EMSG command. In TSO, use the PROFILE MSGID command.

In this appendix, an Explanation section is provided for each Interactive Debug message. User Response and System Action sections are also provided if they differ from the following:

- User Response - reenter the command, correcting the problem that was identified in the error message.
- System Action - issue the Interactive Debug prompt (FORTIAD) in line mode, or re-display the screen in full screen mode, and await entry of another command.

Debugging messages

AFF000E THIS MESSAGE IS RESERVED FOR FUTURE USE; INFORM IBM

Explanation: This message should never occur. If it does, it is the result of a programming error within Interactive Debug.

User Response: Contact your IBM representative.

AFF001A FORTIAD

Explanation: This is the standard prompt displayed when running Interactive Debug in line mode.

User Response: Enter any Interactive Debug command.

AFF002A IAD/E

Explanation: This is the prompt displayed when execution is suspended during an error exit for an error detected by the VS FORTRAN Version 1 or VS FORTRAN Version 2 Library.

User Response: When appropriate, use the FIXUP command to supply corrected values for invalid arguments. Use the GO command to cause standard corrective action to be taken.

AFF003A IAD/A

Explanation: This is the prompt displayed when execution is suspended because of an attention interrupt. When this prompt is displayed, processing is within the Interactive Debug attention exit.

User Response: Enter an Interactive Debug command or a null line.

AFF004A PENDING:

Explanation: This is the prompt displayed when an input line has been continued by using the continuation character "-".

AFF010I VS FORTRAN VERSION 2 RELEASE 3 INTERACTIVE DEBUG

Explanation: This message is issued during initialization of Interactive Debug to identify the product and release level.

AFF011I 5668-806 (C) COPYRIGHT IBM CORP 1985, 1988

Explanation: This message is issued during initialization of Interactive Debug.

AFF013I LICENSED MATERIALS - PROPERTY OF IBM

Explanation: This message is issued during initialization of Interactive Debug.

AFF020E INTERNAL IAD ERROR *number*. FORTRAN PROGRAM MAY HAVE MODIFIED IAD STORAGE

Explanation: This message is issued when an internal error occurs.

User Response: Make sure the application program has not modified Interactive Debug storage. If it has not, contact your IBM representative.

AFF100E "PRINT" OPTION NOT PERMITTED

Explanation: The AUTOLIST command was entered with the PRINT option.

AFF102E "*name*" MUST SPECIFY A COMMAND IN BATCH MODE

Explanation: A SYSCMD, CMS, or TSO command was entered with no operand while operating in batch mode.

User Response: Correct the problem and resubmit the job. If you want to issue a sequence of system commands, you must enter separate SYSCMDs.

AFF103E "*name*" IS SUPPORTED ONLY IN FULLSCREEN MODE; COMMAND IGNORED

Explanation: AUTOLIST, REFRESH, or RESTART was entered in line mode or batch mode.

User Response: Be sure you are running in ISPF before entering this command.

AFF111I PROGRAM TERMINATED EARLY BECAUSE MAXIMUM COUNT WAS REACHED

Explanation: The maximum count value specified in the sublist for the MAXSAMP keyword of the ENDDEBUG command was reached. STOP was also specified which caused the program to be terminated.

AFF112E INTERVAL TIMER WAS RESET BY USER PROGRAM, THUS CANCELLING SAMPLING

Explanation: A non-VS FORTRAN routine called by a VS FORTRAN program performed an STIMER macro, resetting the STIMER set by IAD program sampling.

System Action: Program sampling was discontinued.

AFF121E THE AFFON STATEMENT RESTRICTION LIST WILL BE IGNORED FOR "*name*" BECAUSE IT WAS COMPILED WITH THE "TEST" OPTION

Explanation: You cannot restrict statement hooks if the program unit is compiled with "TEST" because the compiler inserts the hooks.

User Response: Remove the statement restriction list, or recompile the program unit.

System Action: The restriction list is ignored.

AFF122E THE AFFON STATEMENT RESTRICTION LIST FOR "*name*" CONTAINS AN INVALID RANGE, WHICH WILL BE TREATED AS A SINGLE ISN

Explanation: If an invalid range syntax is specified in the AFFON file, only the first ISN will be considered.

User Response: Correct the restriction list, and rerun the job.

System Action: The second ISN is ignored.

AFF123E THE AFFON STATEMENT RESTRICTION LIST FOR "*program-unit-name*" CONTAINS INVALID SYNTAX AND WILL BE IGNORED

Explanation: A hook restriction list on an AFFON entry contains invalid syntax. This can be caused by any of the following:

- An invalid character was found (not alphanumeric or a colon).
- An unrecognized keyword was found (probably misspelled).
- ENTRY or NONE keywords were followed by other operands.
- Missing an ISN number before or after a colon.

System Action: The restriction list is ignored. For a program unit name entry, hooks will be set according to the current defaults. For an ALL entry, the default hook restriction list will remain unchanged.

User Response: Determine the cause of the invalid syntax in the entry, correct the restriction list, and restart the debugging session.

AFF124E THE AFFON STATEMENT RESTRICTION LIST FOR "*name*" CONTAINS AN ENTRY THAT EXCEEDS THE MAXIMUM POSSIBLE ISN; THE MAXIMUM IS ASSUMED

Explanation: An ISN greater than 16777215 was specified in the AFFON restriction list.

User Response: Correct the restriction list, and rerun the job.

System Action: The entry is treated as 16777215.

AFF190E ATTEMPT TO REFERENCE INACCESSIBLE STORAGE

Explanation: Interactive Debug tried to examine storage in an area where it was not allowed to look. This was probably caused by either an invalid address entered as part of a LIST command, or an invalid address within a VS FORTRAN module.

User Response: If you specified an invalid address on a LIST command, reissue the command with valid addresses. If not, try to determine the cause of the invalid address in the program.

AFF192I CURRENT HALT STATUS: *status*

Explanation: This message tells you whether HALT has been issued to indicate when execution is to be suspended. Possible status is STMT, GOTO, or ENTRY.

User Response: None required. This is an informational message.

AFF194E I/O IS ALREADY ACTIVE; COMMAND IGNORED

Explanation: An attempt to issue BACKSPACE, CLOSE, ENDFILE, RECONNECT or REWIND, has been detected while I/O was already active.

User Response: If you need to issue one of these commands, issue a NEXT command so that execution will be suspended after the I/O event is completed.

AFF195E NO DEBUGGABLE FORTRAN PROGRAMS WERE FOUND

Explanation: Interactive Debug has not found any programs within the module to be run that are debuggable program units. In general, for a program unit to be considered debuggable, it must have been compiled with the SDUMP option.

User Response: The problem may be because of an incorrectly specified AFFON file. If so, you should have received other messages detailing other errors.

AFF196E THERE IS NO PRINT DATA SET DEFINED

Explanation: An error has occurred while attempting to open the AFFPRINT data set.

User Response: Correct the cause of the error.

System Action: Processing continues. You will not be able to use the PRINT keyword on any Interactive Debug command.

AFF200E STORAGE EXHAUSTED; SIMPLIFY THE COMMAND OR REMOVE SOME BREAKPOINTS

Explanation: While attempting to build internal control blocks to represent a command, Interactive Debug used up all available storage.

User Response: If possible, issue a less complicated command, or re-invoke Interactive Debug with more virtual storage (VM) or a larger user region (MVS).

AFF210E STORAGE EXHAUSTED DURING SAMPLING, ENTRY POINT SAMPLING INFORMATION WILL BE INCOMPLETE

Explanation: IAD was not able to obtain storage for recording the sampling counts for some entry points of nondebuggable VS FORTRAN routines, VS FORTRAN math. library routines, or non-VS FORTRAN routines. Counts that would have normally been categorized by entry point are grouped together in the *UNKNOWN count.

User Response: Reinvoke Interactive Debug with more virtual storage (VM) or a larger user region (MVS).

AFF220I *synad message*

Explanation: This is the error message returned by the operating system when an attempt was made to write to the AFFPRINT data set.

User Response: Correct the error that caused the message. If printed output is required, after correcting the problem, re-invoke Interactive Debug.

AFF224I "*count*" LINES OF OUTPUT WRITTEN TO AFFPRINT

Explanation: Confirms that information was written to the print file, following a command that was specified with the print option.

AFF225E ERROR WRITING THE PRINT DATA SET; SUBSEQUENT OUTPUT WILL BE WRITTEN TO THE TERMINAL

Explanation: An error was detected when attempting to send "printed" output to the AFFPRINT data set. From this point on, output that would be destined for the print data set is redirected to the terminal.

User Response: If you need to have the print output in a print data set, terminate your debugging session, correct the error, and re-invoke the program.

System Action: Further print output is sent to the terminal.

AFF226E ERROR WRITING THE PRINT DATA SET; SUBSEQUENT OUTPUT WILL BE DISCARDED.

Explanation: An error was detected when attempting to send "printed" output to the AFFPRINT data set in TSO batch mode. From this point on, output that would be destined for the print data set is discarded.

User Response: If you must have the print output in a print data set, terminate your debugging session, correct the error, and re-invoke the program.

System Action: Further print output is discarded.

AFF229E INVALID COUNT VALUE SPECIFIED IN "*number*"

Explanation: A count value larger than 65535 was specified on the AT command.

User Response: Specify a smaller count value, and reissue the AT command.

AFF230E NO BREAKPOINTS CAN BE SET AT STATEMENT "*number*" BECAUSE IT IS COLLAPSED

Explanation: The indicated statement occupies no storage so a breakpoint cannot be set.

User Response: Set your breakpoint at a statement before or after the indicated statement. You can use LISTFREQ to see which statements have hooks.

AFF231E NO BREAKPOINT CAN BE SET AT STATEMENT "*number*" BECAUSE THERE IS NO HOOK THERE

Explanation: The specified statement was not included in the AFFON file restriction list, or is an ENTRY or EXIT of a main program unit.

User Response: None required. You can use LISTFREQ to see which statements have hooks.

AFF240W A SUBSCRIPT IS OUT OF RANGE IN "*array element*"

Explanation: A subscript has been specified for the indicated array element that exceeds the dimension specified when the array element was defined.

AFF241W WARNING: A SUBSTRING BOUNDARY IS OUT OF RANGE IN "*string*"

Explanation: A substring value has been specified which is outside the defined variable length.

System Action: The command is run as normal.

AFF242E "*name*" CANNOT BE ACCESSED; IT COULD BE IN AN UNINITIALIZED DYNAMIC COMMON

Explanation: Variables in a dynamic common cannot be accessed until the common has been initialized and the address has been obtained for the qualifying VS FORTRAN program unit. This occurs the first time the program unit is entered.

User Response: Set a breakpoint at some point that will be reached after the program unit is entered, and access the variables when you get there. If the dynamic common has been initialized, you may be able to access it using a different program unit that has already been entered at least once.

AFF245E "WHERE" INFORMATION IS NOT AVAILABLE AFTER "ENDDEBUG" IS ISSUED

Explanation: ENDDEBUG has been issued and WHERE information cannot be determined.

AFF292E LISTING FILE "*dsname*" CANNOT BE READ

Explanation: Issued when a read error occurs while attempting to annotate a listing. Can occur due to an actual read error or unexpected file format.

User Response: Insure that the file or data set is sequential or is a PDS member and that the LRECL is not greater than 151.

AFF293E AN ARRAY WAS USED WHERE A SCALAR IS REQUIRED IN "*variable*"

Explanation: While scanning the syntax of the previous command, an array variable was found when a scalar variable was required.

AFF294E A SIGN WAS SPECIFIED IN "*text*," BUT THE VARIABLE IS NOT A NUMERIC SCALAR

Explanation: While scanning the previous command, a sign (+ or -) was specified for a non-numeric variable. Only numeric variables may have signs.

AFF295E THERE IS NO ROOM TO INSERT A HOOK IN STATEMENT "*name.number*"; STATEMENT TREATED AS COLLAPSED

Explanation: This is an internal error and should not occur. It indicates that the VS FORTRAN Version 1 or VS FORTRAN Version 2 compiler only allocated two bytes for a VS FORTRAN statement. The compiler should allocate at least four bytes so that an Interactive Debug hook can be inserted.

User Response: Contact your IBM representative. Debugging may be continued, however.

System Action: The statement is treated as a collapsed statement, and you will not be allowed to set a breakpoint at the statement.

AFF296E THE AFFON FILE CANNOT BE READ; FILE IGNORED

Explanation: An I/O error occurred trying to access the AFFON file.

User Response: Correct the cause of the I/O error if you want the AFFON file to be read.

System Action: The AFFON file is ignored and processing continues.

AFF299E ERROR WRITING AFFOUT FILE; FILE IGNORED

Explanation: An I/O error has occurred while attempting to write to the AFFOUT data set. No further attempts will be made to access the file.

User Response: Correct the problem that caused the message. If the log is required, re-invoke Interactive Debug after correcting the problem.

AFF300I AT: *name.number*

Explanation: Execution has been suspended at the identified statement in the identified program unit. It is suspended because a prior AT command requested a breakpoint at this statement.

User Response: Enter debugging commands, or GO to resume execution.

AFF301I NEXT: *name.number*

Explanation: Execution has been suspended at the identified statement in the identified program unit. It is suspended because a NEXT or STEP command was issued.

User Response: Enter debugging commands, or GO to resume execution.

AFF303I TRACE STATUS: status

Explanation: Provides the current trace status in response to the TRACE command with no operands.

AFF304I TRACE: FROM *name.number* **TO** *name.number*

Explanation: Execution has passed from the first identified statement to the second identified statement. The second statement did not immediately follow the first statement in the VS FORTRAN source. This message is received because of an earlier TRACE command that was issued.

User Response: None required.

AFF305W "ERROR" COMMAND TERMINATED AFTER PROCESSING ERROR NUMBER *number*

Explanation: The PURGE command was used to terminate excessive output from an ERROR command. The last error number processed was "number."

User Response: None required.

AFF306I PROGRAM HAS TERMINATED; RC=(*code***)**

Explanation: The application program being run has completed. If a return code was coded on the STOP statement, it is provided.

User Response: None required.

System Action: You will be allowed to continue entering commands until a QUIT command is entered, at which time the debugging session will be terminated.

AFF307W COMMAND OUTPUT REFLECTS THE STATE OF EXECUTION PRIOR TO ENTERING "ENDDEBUG"

Explanation: The information presented as output for the command which was just issued, is not necessarily current information. It was correct when you issued an ENDDEBUG command earlier in the debugging session, and has not been updated since ENDDEBUG was issued.

User Response: None required.

AFF310E MISSING ISN OR "*" BEFORE "DOLOOP", "DONEST", OR "DOVECT" IN AFFON ENTRY FOR *"program-unit-name"***.**

Explanation: A DOLOOP, DONEST, or DOVECT keyword was found on an AFFON entry without an ISN, an ISN range, or an asterisk preceding it.

System Action: The restriction list is ignored. For a program unit name entry, hooks will be set according to the current defaults. For an ALL entry, the default hook restriction list will remain unchanged.

User Response: Determine the cause of the invalid syntax in the entry, correct the restriction list, and restart the debugging session.

AFF311E PROGRAM INFORMATION FILE *"file-name"* SPECIFIED FOR *"program-unit-name"* CANNOT BE FOUND.

Explanation: The data set or file specified to contain program information for the program unit could not be opened.

System Action: The program unit is treated as having an undefined program information file.

User Response: The name is probably misspelled. The name must be fully qualified and enclosed in quotes. If it is spelled correctly, then the file either does not exist or cannot be accessed. You may need to recompile the program to generate the file, or obtain read access to the file. Then, restart the debugging session.

AFF312E FILE *"file-name"* SPECIFIED AS PROGRAM INFORMATION FILE FOR *"program-unit-name"* IS NOT A PROGRAM INFORMATION FILE.

Explanation: The data set or file specified to contain program information for the program unit is not a program information file.

System Action: The program unit is treated as having an undefined program information file.

User Response: Check to make sure that the right data set or file was specified. If so, then the file may have been modified in some way since compilation. Try recompiling the program to re-create the file.

AFF313E PROGRAM INFORMATION FILE *"file-name"* WAS SPECIFIED FOR *"program-unit-name"*, BUT PROGRAM UNIT IS NOT COMPILED WITH THE IVA OPTION.

Explanation: A program information file was specified for the program unit, but the program unit was not compiled with the IVA sub-option of the VECTOR option.

System Action: The program is treated as having an undefined program information file.

User Response: If you wish to perform vector analysis functions with this program unit, you will need to recompile it with the IVA sub-option of the VECTOR option.

AFF315E ERROR READING PROGRAM INFORMATION FILE FOR *"program-unit-name"*.

Explanation: A read error occurred while reading the program information for the program unit.

System Action: The program unit is treated as having an undefined program information file.

User Response: Check to make sure you are properly accessed to the file. If on CMS, try re-accessing the mini-disk containing the file.

AFF316E ERROR IN THE PROGRAM INFORMATION FILE DATA FOR *"program-unit-name"*.

Explanation: An error was found in the program information for the program unit. This is probably because the file has been changed in some way since compilation.

System Action: The program unit is treated as having an undefined program information file.

User Response: Check to make sure you are properly accessed to the file. If on CMS, try re-accessing the mini-disk containing the file. If the file has been modified in some way since compilation, you will need to recompile the program unit to re-create the file.

AFF317E PROGRAM INFORMATION FILE *"file-name"* DOES NOT CONTAIN DATA FOR *"program-unit-name"*.

Explanation: The program information file specified for the program unit does not contain program information for the program unit. The wrong program information file was probably specified.

System Action: The program unit is treated as having an undefined program information file.

User Response: Check to make sure that the file specified is correct for the program unit.

AFF318E PROGRAM INFORMATION for *"program-unit-name"* DOES NOT HAVE THE SAME COMPILATION DATE AS THE CODE FOR THE PROGRAM UNIT.

Explanation: Time stamps in the program information file and in the code for the program unit do not match.

System Action: The program unit is treated as having an undefined program information file.

User Response: Re-compile or re-link edit the program to make sure that the module being debugged contains the latest compilations of the program units and that the program information file is from the latest compilation.

AFF319E AN "(ALL)" ENTRY IN THE AFFON FILE HAS A STATEMENT RESTRICTION LIST THAT CONTAINS ISN'S. THE RESTRICTION LIST IS IGNORED.

Explanation: The "ALL" entry in the AFFON file specified an ISN or an ISN range. These are not allowed on the ALL entry, because the ALL entry is supposed to contain defaults for all program units. ISN's are only meaningful to a specific program unit.

System Action: The default hook restriction list will remain unchanged.

User Response: Remove the ISN's on the ALL entry in the AFFON file. You may specify "*", "* DOLOOP", "* DONEST", "* DOVECT", "ENTRY", or "NONE" as the statement restriction list for an ALL entry.

AFF320E AFFON ENTRY FOR *"program-unit-name"* CONTAINS STATEMENT RESTRICTION *"isn* DOLOOP", BUT ISN IS NOT A DO STATEMENT.

Explanation: The statement restriction list for a program unit in the AFFON file specifies a DO loop ISN that is not the ISN of a DO statement.

System Action: The statement restriction is ignored. If this is the only statement restriction in the list, then hooks are only set at entry and exit of the program unit.

User Response: Check to make sure the correct ISN was specified. It must be the ISN of a DO statement.

AFF321E AFFON ENTRY FOR *"program-unit-name"* CONTAINS STATEMENT RESTRICTION *"isn* DONEST", BUT ISN IS NOT THE DO STATEMENT OF AN OUTERMOST NESTED DO LOOP.

Explanation: The statement restriction list for a program unit in the AFFON file specifies an ISN that either is not the ISN of a DO statement, or is the ISN of a DO statement but the DO statement is not for the outermost loop in a nest of DO loops.

System Action: The statement restriction is ignored. If this is the only statement restriction in the list, then hooks are only set at entry and exit of the program unit.

User Response: Check to make sure the correct ISN was specified. It must be the ISN of the DO statement of the outermost DO loop in a nest of DO loops.

AFF322E AFFON ENTRY FOR *"program-unit-name"* CONTAINS STATEMENT RESTRICTION *"isn* DOVECT", BUT ISN IS NOT THE DO STATEMENT OF A VECTORIZED DO LOOP.

Explanation: The statement restriction list for a program unit in the AFFON file specifies an ISN that either is not the ISN of a DO statement, or is the ISN of a DO statement but the DO loop is not vectorized.

System Action: The statement restriction is ignored. If this is the only statement restriction in the list, then hooks are only set at entry and exit of the program unit.

User Response: Check to make sure the correct ISN was specified. It must be the ISN of the DO statement of a vectorized DO loop.

AFF323I *statement-id*:

Explanation: This is the first line of the LISTVEC output for each DO loop listed. It is the identifier of the DO statement for the loop.

AFF324I STATUS = *status*

Explanation: This message is part of the LISTVEC output for a DO loop. It shows the current status of vector statistics recording for the loop (set by the VECSTAT command). It has two settings, "ON" and "OFF". If the setting is OFF, then the statistics are not updated when the loop is run.

AFF325I TOTAL NUMBER OF EXECUTIONS = *number*

Explanation: This message is part of the LISTVEC output for a DO loop. It shows the total number of executions of the DO loop. The number of executions of a DO loop is defined as the number of times the DO statement for the loop is entered. If the total number of executions could not be determined because the DO loop does not contain DO loop analysis hooks, then "NO HOOK" appears instead of the value.

AFF326I AVERAGE ITERATION COUNT = *number*

Explanation: This message is part of the LISTVEC output for a DO loop. It shows the average iteration count of the loop. The iteration count is equivalent to the length for all vectors in the loop. The iteration count is calculated by dividing the sum of the iterations for each execution of the loop by the number of executions of the loop. If the average iteration count could not be determined because the DO loop does not contain DO loop analysis hooks, then "NO HOOK" appears instead of the value.

AFF327I ESTIMATED ITERATION COUNT = *number ?*

Explanation: This message is part of the LISTVEC output for a DO loop. It shows the compiler estimate for the iteration count (length) of the DO loop. A question mark after the compiler estimate indicates that the compiler could not accurately estimate the value. The value displayed is an arbitrary value that the compiler used for vector cost analysis.

AFF328I STATEMENT ARRAY AVG STRIDE EST STRIDE

Explanation: This message is part of the LISTVEC output for a DO loop. It is the header message before the listing of the strides of a DO loop.

AFF329I *v statement-id array-name number number ?*

Explanation: This is a line of the LISTVEC output for a stride of a DO loop. It shows:

- a "V" or "S" indicating that the statement is part of a vector or scalar loop, respectively. "S" will also be placed on statements belonging to the scalar portion of a partially vectorized loop.
- the statement identifier of the array reference,
- an array reference indexed by the DO variable of the loop,
- the average stride of reference, and
- the compiler estimated stride.

If the stride could not be obtained for an array reference, because the DO loop does not contain DO loop analysis hooks, then "NO HOOK" appears instead of the value. If the stride could not be obtained because information was not supplied by the compiler to locate and calculate the value, then, "UNKNOWN" appears instead of the value. A question mark (?) after the compiler estimate indicates that the compiler could not accurately estimate the value. The value displayed is an arbitrary value that the compiler used for vector cost analysis.

AFF330I NOT ANALYZABLE

Explanation: This message is displayed as part of the LISTVEC output, in place of the vector statistics for a DO loop. The listed DO loop could not be analyzed for vectorization by the VS FORTRAN compiler. Run-time statistics could not be gathered for the loop, and compiler estimates are not available.

AFF331E PROGRAM INFORMATION FILE DATA IS NOT AVAILABLE FOR *"program-unit-name"*.

Explanation: A program information data set or file was not defined for the program unit. This file is needed by Interactive Debug to perform DO loop analysis functions (VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, and LISTSAMP DOLOOP).

User Response: Create or change the AFFON file entry for the program unit to specify a correct program information file for the program unit. You may also need to recompile the program unit with the VECTOR(IVA) option. Restart the debugging session.

AFF332E DO LOOP *"statement-id"* IS NOT ANALYZABLE.

Explanation: A VECSTAT, LISTVEC, TIMER DOLOOP, or LISTTIME DOLOOP command was entered stating a DO loop (not in a range) that was not analyzable by the compiler.

User Response: If you wish to gather length and stride information for this loop, or time this loop, you must modify your program to make the loop analyzable.

AFF333E DO LOOP *"statement-id"* DOES NOT CONTAIN DO LOOP ANALYSIS HOOKS.

Explanation: The DO loop explicitly specified on the command does not contain the DO loop analysis hooks. These hooks are needed to gather length and stride information, and to time DO loops.

User Response: Change the AFFON file entry for the program unit to specify that DO loop analysis hooks be placed in the loop. Restart the debugging session.

AFF334E STATEMENT *"statement-id"* IS NOT A DO STATEMENT.

Explanation: A statement explicitly specified on a VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, or LISTSAMP DOLOOP command is not a DO statement.

AFF335I DO LOOP SAMPLES %UNIT %TOTAL

Explanation: This is the header message for LISTSAMP DOLOOP output.

AFF3361 *statement-id* *number* *prcnt* *prcnt* *histogram*

Explanation: This is a line in the LISTSAMP DOLOOP output. It shows:

- the statement identifier of the DO statement for the loop,
- the total number of samples in the loop,
- the percentage of samples to the total for the program unit,
- the percentage of samples to the total for the program, and
- a histogram of the percentage of samples to the total for the program (where one asterisk is 5 percent, rounded).

AFF3371 NO STRIDE INFORMATION AVAILABLE

Explanation: This message is issued in place of a list of strides for a DO loop on a LISTVEC report if the DO loop does not contain any vectors.

AFF3391 DO LOOP STATUS TOTAL TIME EXECUTIONS AVERAGE TIME

Explanation: This is the header message for LISTTIME output for DO loops.

AFF3401 *statement-id* *number* *number* *number* *status*

Explanation: This is a line of LISTTIME output for DO loops. It shows:

- the statement identifier of the DO statement for the loop,
- the execution time of the DO loop (in microseconds),
- the number of executions of the DO loop, and
- the average time per execution of the loop (in microseconds).
- the timing status for the DO loop (set by the TIMER command). The settings are "ON" and "OFF".

AFF3471 PROGRAM UNIT PAGE FREQUENCY DISTRIBUTION

Explanation: This is the header message for an annotate summary for statement frequency counts.

AFF3481 *program-unit-name* *number* *frequency histogram*

Explanation: This is a line of an annotate summary for statement frequency counts. It shows:

- the program unit name
- the sum of the frequencies of all statements in the program unit
- a histogram showing the sum in item as a percentage of the sum of frequencies of all statements in the entire program.

AFF349E INVALID SYNTAX *"text"* AT START OF AFFON ENTRY. PROGRAM UNIT NAME, "(ALL)", OR "*" EXPECTED.

Explanation: An entry in the AFFON file begins with a character string that is not valid syntax for a program unit name, does not begin with the string "(ALL)" used to signify ALL entries, and does not begin with "*" used to indicate comment entries.

System Action: The entry is ignored.

User Response: Correct cause of invalid syntax, and restart debugging session.

AFF350E ERROR OBTAINING STORAGE FOR PROGRAM INFORMATION FOR *"program-unit-name"*.

Explanation: There is not enough storage to contain program information file data for the program unit.

System Action: The program unit is treated as having an undefined program information file.

User Response: Re-invoke Interactive Debug with more virtual storage (VM) or a larger user region (MVS).

AFF351E ERROR OBTAINING STORAGE FOR AFFON FILE INFORMATION.

Explanation: There is not enough storage to contain AFFON file information.

System Action: The AFFON file is ignored.

User Response: Re-invoke Interactive Debug with more virtual storage (VM) or a larger user region (MVS).

AFF352E NO DO LOOP INFORMATION IS AVAILABLE FOR PROGRAM UNIT *"program-unit-name"*.

Explanation: A VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, or LISTSAMP DOLOOP, command specified a program unit that does not contain any DO loops.

AFF353E THERE ARE NO DO LOOPS IN ANY DEBUGGABLE PROGRAM UNIT.

Explanation: A VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, or LISTSAMP DOLOOP command was issued with the *.* operand, yet there are no DO loops in any debuggable program unit.

AFF354E THERE ARE NO PROGRAM INFORMATION FILES DEFINED FOR ANY DEBUGGABLE PROGRAM UNIT.

Explanation: A VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, or LISTSAMP DOLOOP command was issued with the *.* operand, yet there is no program information for any debuggable program unit.

AFF355E THERE ARE NO DO LOOPS IN THE RANGE *"statement-range-id"*.

Explanation: A statement range was specified on a VECSTAT, LISTVEC, TIMER DOLOOP, LISTTIME DOLOOP, or LISTSAMP DOLOOP command, but there are no DO loops in the range.

AFF356I *array-name number number ?*

Explanation: This is a continuation message for message AFF329I.

AFF357I *number number ?*

Explanation: This is a continuation message for message AFF356I.

AFF358I *number prcnt prcnt histogram*

Explanation: This is a continuation message for message AFF336I.

AFF360E ERROR READING AFFLST FILE; *"program-unit-name"* CANNOT BE ANNOTATED.

Explanation: There was a problem reading a listing from the AFFLST file during the execution of an ANNOTATE command. The AFFLST file is probably not defined correctly.

User Response: Check the AFFLST DD statement in the JCL to make sure the listings are properly defined. Correct the JCL and rerun the job.

AFF361E *"program-unit-name"* CANNOT BE ANNOTATED BECAUSE ITS LISTING WAS NOT FOUND IN THE AFFLST FILE.

Explanation: The listing for the specified program unit was not in the AFFLST file.

User Response: Check the AFFLST DD statement in the JCL to make sure that the listing for the program unit is specified. Correct the JCL and rerun the job.

AFF362E ERROR READING AFFPIF FILE; FILE IGNORED.

Explanation: There was an error reading program information from the AFFPIF file. The file is probably not defined correctly.

System Action: The AFFPIF file is ignored. Program units whose program information was to be taken from the AFFPIF file are treated as having undefined program information.

User Response: Check the AFFPIF DD statement in the JCL to make sure that it is properly defined. Correct the JCL and rerun the job.

AFF363E THE AFFPIF FILE DOES NOT CONTAIN PROGRAM INFORMATION.

Explanation: The AFFPIF file contains data that could not be interpreted as program information.

System Action: The AFFPIF file is ignored. Program units whose program information was to be taken from the AFFPIF file are treated as having undefined program information.

User Response: Check the AFFPIF DD statement in the JCL to make sure that program information is being properly defined. Correct the JCL and rerun the job.

AFF364E THE AFFPIF FILE DOES NOT CONTAIN PROGRAM INFORMATION FOR *"program-unit-name"*.

Explanation: Program information for the specified program unit was not in the AFFPIF file.

System Action: The program unit is treated as having undefined program information.

User Response: Check the AFFPIF DD statement in the JCL to make sure that program information for the program unit is specified. Correct the JCL and rerun the job.

AFF370E A SHIFT-IN CONTROL CHARACTER WAS NOT FOUND IN THE DBCS LITERAL.

Explanation: A shift-in control character is required to indicate the end of a DBCS character string.

AFF370W LINE TRUNCATED

Explanation: Line continuation was used to enter a line to Interactive Debug that exceeded 255 characters.

System Action: The line is truncated to 255 characters.

User Response: If the input line is a command, simplify the command (perhaps by using abbreviations) to fit into 255 characters.

AFF371I INPUT ABORTED BY "END"

Explanation: This message is issued when an continued input line (using the line continuation character "-") was aborted by typing "END" on a continuation line.

AFF372E THE SHIFT-OUT CHARACTER CAN NOT BE USED AS A DELIMITER.

Explanation: The DBCS shift-out control character can not be used as a delimiter of the SEARCH command.

AFF372W *"command-name"* **IS SUPPORTED ONLY IN FULLSCREEN MODE; COMMAND IGNORED.**

Explanation: A full screen only command was issued in line mode or batch mode.

User Response: Make sure you are running in ISPF before entering this command.

AFF373W *"command-name"* **COMMAND CANNOT BE ISSUED IN A RESTART FILE.**

Explanation: The specified command (a full screen command or the RESTART command) is not allowed in a restart (AFFIN) file.

System Action: The command is ignored.

AFF400I THE COMMAND LIST FOR THE BREAKPOINT AT "*name.number***" HAS BEEN TERMINATED BY AN ATTENTION**

Explanation: Because of entering an attention exit, the indicated command list cannot be completed.

User Response: If necessary, enter the commands that were not completed.

AFF405E THE NONIMMEDIATE COMMAND "*command***" WAS IGNORED DUE TO A PENDING ERROR EXIT**

Explanation: The indicated command could not be run now because it is not a command that can be immediately processed (that is, processed easily by issuing a message, like WHERE, or by setting a flag, like NEXT), and cannot be deferred to later processing because execution is currently within an error exit.

User Response: Issue a NEXT, and then issue the command after execution has left the error exit.

AFF410E UNKNOWN COMMAND

Explanation: The syntax of the previous "command" name was not a valid Interactive Debug command.

User Response: Check your spelling of the command name for accuracy or insure that you have included the SYSCMD command for a system command.

AFF450E *"name"* **IS AN ASSUMED SIZE ARRAY; SUBSCRIPTS MUST BE SPECIFIED FOR "LIST"**

Explanation: A LIST or SET command was issued for the specified array. However, the final dimension of the array is unknown because it was not defined at compile time. For example, B(*) may have been specified. Interactive Debug will not LIST or SET values in an assumed size array unless a specified element (for example, B(3)) is specified.

AFF454E "GOTO" OR "ENTRY" MUST BE SPECIFIED WITH "PRINT"

Explanation: GOTO or ENTRY was not specified with the PRINT option on the TRACE command.

User Response: Reissue the TRACE command with either the GOTO or ENTRY options.

AFF455E INVALID OPERAND SYNTAX SPECIFIED IN "*text*"

Explanation: Invalid syntax has been specified for the indicated command operand.

AFF456E "*word*" IS NOT A VALID KEYWORD FOR THE "*cmnd*" COMMAND

Explanation: Issued when an invalid keyword option is detected in an IAD command.

AFF457E "*abbr*" IS AN AMBIGUOUS KEYWORD ABBREVIATION FOR THE "*cmnd*" COMMAND

Explanation: Issued when an ambiguous abbreviation is detected in an IAD command.

User Response: Use a longer abbreviation for the keyword.

AFF458E KEYWORD "*word*" WAS SPECIFIED MORE THAN ONCE FOR THE "*cmnd*" COMMAND

Explanation: Issued when an option keyword is specified more than once in an IAD command.

User Response: Specify the option keyword only once.

AFF459E "*word1*" AND "*word2*" CANNOT BOTH BE SPECIFIED ON THE "*cmnd*" COMMAND

Explanation: Conflicting keywords are specified.

User Response: Reissue the command with only one of the keywords.

AFF460E UNBALANCED DELIMITERS SPECIFIED IN "*text*"

Explanation: While scanning the syntax of the previous command, an invalid use of delimiters was detected. Usually, a right or left parenthesis is missing.

AFF461E INVALID SUBLIST SYNTAX IN "*text*"

Explanation: Issued when incorrect syntax is detected in a parenthesized sublist following a keyword.

AFF462E KEYWORD "*word*" OF THE "*cmnd*" COMMAND REQUIRES A SUBLIST

Explanation: Issued when a parenthesized sublist was not specified for a keyword that requires one.

User Response: Reissue the command with the required sublist.

AFF463E KEYWORD "*word*" OF THE "*cmnd*" COMMAND DOES NOT ALLOW A SUBLIST

Explanation: Issued when a parenthesized sublist was specified for a keyword that does not permit one.

AFF464E SUBLIST VALUE "*number*" IS TOO LARGE FOR KEYWORD "*word*"

Explanation: The value specified in the sublist is larger than the maximum allowed.

AFF470E UNKNOWN COMMAND "*text*"

Explanation: The indicated string was found in a command list, but is not recognized as a valid Interactive Debug command.

AFF481E THE DESTINATION CANNOT BE BRANCHED TO IN "*text*"

Explanation: A GO command was entered that references a VS FORTRAN statement that has no hook. This includes a GO EXIT for a VS FORTRAN Version 1 MAIN program.

User Response: You can use LISTFREQ to see which statements have hooks.

AFF483E "GO" WITH A STATEMENT IDENTIFIER CANNOT BE ISSUED FROM AN ENTRY

Explanation: A GO command with a statement identifier cannot be issued from the entry point of any VS FORTRAN program unit. At entry, the unit is not yet active.

User Response: Issue STEP 1 to get to the first statement with a hook. You should be able to issue the GO command from there.

AFF484E CONTINUATION IS NOT PERMITTED WITH THE "*command*" COMMAND

Explanation: A command valid only in full screen mode was entered in full screen mode, but with continuation. Continuation is not supported for these commands.

AFF485E "*cmnd*" COMMAND CANNOT BE ISSUED FROM AN ATTENTION EXIT

Explanation: A full screen display command is not allowed from an attention exit.

User Response: Exit the attention mode and reissue the command.

AFF486E "*cmnd*" COMMAND CANNOT BE ISSUED FROM A RESTART FILE

Explanation: A full screen display command is not allowed in a RESTART file.

System Action: The command is ignored.

AFF500E STATEMENT *number* IS NOT EXECUTABLE

Explanation: The statement reference does not identify an executable statement. Possibly the current qualification identifies a program unit that does not contain a statement with the specified number.

User Response: Use the QUALIFY command to set the proper qualification, or select a different statement.

AFF510E INVALID RANGE; THE RIGHT SIDE (*number*) IS LESS THAN THE LEFT SIDE (*number*)

Explanation: If a command has been entered with a range of statements specified, the first statement must appear prior to the second in the program.

AFF511E INVALID SUBSTRING RANGE; THE RIGHT SIDE IS LESS THAN THE LEFT SIDE IN "*string*"

Explanation: The value specified on the right side of the substring notation is larger than the value specified on the left side.

AFF520E *program* IS NOT A DEBUGGABLE FORTRAN PROGRAM UNIT

Explanation: The indicated program unit cannot be debugged.

1. If it is a VS FORTRAN Version 1 or VS FORTRAN Version 2 program, it probably was compiled with the NOSDUMP option. Either SDUMP or TEST must be specified (or defaulted) for the program unit to be eligible for debugging.
2. A valid AFFON file was found but did not contain this program unit.

User Response: If you want to debug the indicated program unit, terminate the current debugging session, using the QUIT command, correct the AFFON file or recompile the program unit with the appropriate compiler options, and re-execute the program.

AFF530E "*text*" IS INVALID "FIXUP" SYNTAX

Explanation: Invalid keyword syntax has been detected for a FIXUP command. The only valid keywords are ARG1 and ARG2 and both must contain a value within parentheses following the keyword.

AFF535E NONNUMERIC VALUE "*text*" IS NOT ALLOWED IN FIXUP

Explanation: A logical or character value has been specified in a FIXUP command as value for either ARG1 or ARG2. Only numeric values are valid.

AFF540E FORTRAN TERM "*text*" IS NOT ALLOWED IN FIXUP

Explanation: Usually, valid syntax has been detected where it is not allowed in a FIXUP command (for example, a duplication factor).

AFF545E A NULL FORTRAN TERM IS INVALID

Explanation: One of the sides of a range specification is missing (for example, "LIST A(1):").

AFF549E PROGRAM SAMPLING REQUIRES VS FORTRAN VERSION 2 RELEASE 2.0 LIBRARY OR LATER

Explanation: The module being debugged was link-edited with an older release of the VS FORTRAN library. Interactive Debug needs Version 2 Release 2.0 in order to perform program sampling.

User Response: Relink-edit the program with the latest release of the library.

AFF550I PROGRAM SAMPLING INTERVAL WAS *m* MS; TOTAL NUMBER OF SAMPLES WAS *n*

Explanation: This is the header message for a LISTSAMP display, where "m" is the sampling time interval used and "n" is the total number of sampling interruptions that occurred.

AFF551I "*type*" SAMPLES:

Explanation: The sampling counts that follow are for sampling interruptions of the type indicated.

AFF552I SUM OF DIRECT AND CALLED SAMPLES:

Explanation: The counts that follow are the sum of both DIRECT and CALLED counts.

AFF553E "*unitname*" CANNOT BE ANNOTATED BECAUSE ITS LISTING FILE IS NOT KNOWN

Explanation: Issued when the listing file has not been identified for a program unit that has had annotation requested for it.

User Response: Specify the file or data set name in the AFFON file, or use the LISTINGS panel in full screen mode.

AFF554E "*unitname*" CANNOT BE ANNOTATED BECAUSE IT WAS NOT FOUND IN "*dsname*"

Explanation: Issued when the listing for a specified program unit cannot be found in the specified listing data set.

User Response: Be sure the correct file or data set name is specified in the AFFON file.

AFF555I STATEMENT SAMPLES %UNIT %TOTAL

Explanation: This is the title line for a LISTSAMP output when statement information is listed.

AFF556I PROGRAM UNIT SAMPLES %TOTAL

Explanation: This is the title line for LISTSAMP output when summary information is listed.

AFF557I statement samples %unit %total histogram

Explanation: This is the program sampling information for a statement, showing statement number, number of samples, percentage of total samples for the program unit, percentage of total samples for the entire program, and a histogram (bar chart) that graphically displays the size of this value relative to the other sampling values listed.

AFF558I program samples %total histogram

Explanation: This is the program sampling information for a program unit, showing program unit name, number of samples, percent of total samples, and a histogram (bar chart).

AFF559E "*unitname*" CANNOT BE ANNOTATED BECAUSE IT WAS NOT COMPILED WITH VS FORTRAN V2

Explanation: Issued when annotation is requested for a program unit that was not compiled with VS FORTRAN VERSION 2. Annotation is not supported for listings produced by previous versions of the compiler.

AFF560E PROGRAM SAMPLING HAS NOT BEEN DONE; ISSUE "ENDDEBUG" WITH THE "SAMPLE" OPTION TO INITIATE PROGRAM SAMPLING

Explanation: Issued when LISTSAMP or ANNOTATE with the SAMPLING option is entered but program sampling has not been initiated.

AFF561I ANNOTATING LISTING FOR PROGRAM UNIT "*unitname*"

Explanation: An informational message indicating the progress of the ANNOTATE command.

AFF562I samples %unit %total histogram

Explanation: This is program sampling information for a statement, showing number of samples, percentage of total samples for the program unit, percentage of total samples for the entire program, and a histogram (bar chart) that graphically displays the size of this value relative to the other sampling values listed. This information will appear on the line below the statement number, in those cases where these fields and the statement number are too long to display together on a single line.

AFF563I VS FORTRAN INTERACTIVE DEBUG V2 R3 ANNOTATED LISTINGS:

Explanation: This is the heading line for annotated listings.

AFF564I PROGRAM UNIT PAGE %TOTAL DISTRIBUTION

Explanation: This is the heading line for the summary page of annotated listings.

AFF565I unitname page percent histogram

Explanation: This is the format of the output of the summary lines shown on the summary page of annotated listings.

AFF566I ANNOTATE: *status*

Explanation: Indicates the current status of the ANNOTATE controls that are used for displaying bar charts on the source window. The status may be any one of the following:

ON SAMPLING DIRECT
ON SAMPLING CALLED
ON SAMPLING ALL
OFF SAMPLING DIRECT
OFF SAMPLING CALLED
OFF SAMPLING ALL
ON FREQUENCY
OFF FREQUENCY
ON MESSAGE
OFF MESSAGE

AFF567E "CALLED" IS NOT VALID UNLESS SAMPLING WAS INITIATED WITH THE "CALLED" OPTION

Explanation: The CALLED keyword must be specified in the ENDDEBUG command in order for CALLED to be valid in the ANNOTATE and LISTSAMP commands.

User Response: Reissue the command without the CALLED option, or restart the debugging and specify the CALLED option, in ENDDEBUG, when initiating sampling.

AFF568I FREQUENCIES

Explanation: Heading for annotated listing by program unit when frequency values are displayed.

AFF569E "*word***" KEYWORD IS NOT PERMITTED WITHOUT THE "SAMPLE" KEYWORD**

Explanation: The MAXSAMP or CALLED keywords are only valid if SAMPLE has been specified for the ENDDEBUG command.

AFF570E TIMING INTERVAL MUST BE AN INTEGER LARGER THAN ZERO

Explanation: The sample time specified in the SAMPLE sublist of the ENDDEBUG command must be an integer value larger than zero.

AFF571E INVALID STOP PARAMETER SPECIFIED IN "MAXSAMP" SUBLIST

Explanation: STOP, or an abbreviated form of STOP, was incorrectly specified in the MAXSAMP sublist of the ENDDEBUG command.

AFF572E NO SUBLIST SPECIFIED FOR THE "*word*" KEYWORD

Explanation: A sublist is required with the indicated keyword.

AFF573E "*number*" IS AN INVALID "*word*" VALUE

Explanation: The numeric value specified in the keyword sublist is too large.

AFF574E MAXIMUM NUMBER OF SAMPLING INTERRUPTS MUST BE AN INTEGER GREATER THAN ZERO

Explanation: The maximum number of sampling interrupts specified in the MAXSAMP sublist of the ENDDEBUG command must be an integer value greater than zero.

AFF575E "*text*" IS NOT A VALID SUBLIST FOR "*word*"

Explanation: A sublist after the keyword on a command contains invalid syntax.

AFF576E "DUMP" IS NOT PERMITTED WITH A CONSTANT OPERAND

Explanation: LIST or AUTOLIST has been specified with both a constant operand and the dump option. This combination is not permitted.

AFF577E THE MINIMUM SAMPLING INTERVAL ON CMS IS 4 MILLISECONDS.

Explanation: When using CP timer assist, the minimum accuracy of the interval timer on CMS is about 3.3 milliseconds. To prevent sampling interruptions from occurring in the operating system code servicing of the interruption, the interval time is restricted to 4 milliseconds or greater.

AFF605E ONLY REALS ALLOWED IN COMPLEX CONSTANT "*text*"

Explanation: Integers have been used as part of a complex constant. Only real numbers are allowed.

AFF610E REAL AND IMAGINARY PARTS OF "*variable*" DIFFER IN LENGTH

Explanation: One part of a complex constant has been entered as a REAL*4 number and the other part has been entered as a REAL*8 number. Both parts must be of equal length.

AFF615E "*text*" IS INVALID FORTRAN TERM SYNTAX

Explanation: While scanning the previous command, an operand that must be a VS FORTRAN term had invalid syntax.

AFF620E SUBSCRIPTS ARE NOT PERMITTED ON "*variable*"; THE VARIABLE IS NOT AN ARRAY

Explanation: Subscripts have been specified for a variable that is not an array.

AFF625E THE NUMBER OF SUBSCRIPTS ON "*array*" DOES NOT MATCH THE DECLARED NUMBER OF DIMENSIONS

Explanation: There are too many or too few subscripts specified for the indicated array.

AFF630E "*variable*" IS AN INVALID FORTRAN VARIABLE NAME

Explanation: The name of a VS FORTRAN variable is invalid. For example, the name contains more than six characters, or does not begin with an alphabetic character.

AFF631E PROGRAM UNIT "*name*" IS NOT ACTIVE; "*varname*" IS A DUMMY ARGUMENT AND CANNOT BE ACCESSED

Explanation: An AUTOLIST, LIST, SET, IF, or WHEN command attempted to access variables that have no storage because the program unit is not active.

User Response: Set a breakpoint within the program unit and access the variables when you get there. If the dynamic common has been initialized, you may be able to access it using a different program unit.

AFF632E "*varname*" IS A DUMMY ARGUMENT THAT IS NOT DEFINED AT THE ENTRY POINT BY WHICH "*program*" WAS ENTERED, AND CANNOT BE ACCESSED

Explanation: An attempt was made to reference a dummy argument that is defined only in an alternate entry point.

User Response: Wait until the program unit is entered by an entry point that defines the dummy variable you want to access.

AFF633E "QUIT" HAS BEEN ISSUED. ENTER "QUIT" AGAIN TO FORCE AN ABNORMAL TERMINATION

Explanation: This message is issued by the attention interrupt handler.

User Response: Enter QUIT again if you want to terminate debugging; otherwise, enter any command that is appropriate in an attention exit.

AFF635E VARIABLE "*variable*" IS NOT IN PROGRAM UNIT *program*

Explanation: The specified variable was not found in the specified program unit. The variable may be misspelled, the program qualification may be missing or incorrect, or the variable may have been removed by the optimizer.

User Response: Make sure that the variable is defined in the currently qualified program unit. If not, then either use the QUALIFY command, or explicitly qualify the variable name in the Interactive Debug command.

AFF636E "COMMAND NOT FOUND"

Explanation: The TSO command requested using the SYSCMD command was not found by TSO.

AFF640E "*text*" IS INVALID CONSTANT SYNTAX

Explanation: There is a syntax error in the indicated constant.

AFF645E CONSTANT "*number*" EXCEEDS THE MACHINE CAPACITY

Explanation: The indicated constant is too large. This may occur if leading zeros are specified with the constant.

AFF650E INVALID SUBSCRIPT IN "*text*"; A SUBSCRIPT CANNOT BE AN ARRAY OR ARRAY ELEMENT

Explanation: The subscripts of the indicated array must be scalar constants or variables.

AFF655E ONE OR MORE PARTS OF COMPLEX CONSTANT "*text*" ARE MISSING

Explanation: Either the real or the imaginary portion of the indicated complex constant is missing. Both portions are required.

AFF660E "*text*" HAS INVALID SUBSCRIPT SYNTAX

Explanation: While scanning what appears to be a subscript, invalid syntax was discovered. Possibly the right parenthesis was missing.

AFF665E INVALID SUBSCRIPT IN "*array*"; A NON-INTEGER VARIABLE WAS SPECIFIED

Explanation: A logical, real, character, or complex variable has been used as a subscript for the indicated array. Only integer numbers may be used as subscripts.

AFF670E "*text*" IS INVALID ON THE LEFT SIDE OF A SET COMMAND

Explanation: A duplication factor, a minus sign, or a constant appears on the left side of a SET command. None of these are valid on the left side.

AFF675E "*text*" IS INVALID "SET" SYNTAX

Explanation: Either the equal sign (=) or the right side of the SET assignment has been omitted.

AFF680E "*text*" IS INVALID IN THE "LIST" RANGE

Explanation: An item is specified in a range that is not allowed in a LIST command.

AFF690E VARIABLE "*variable*" IS INVALID IN "*command*" COMMAND

Explanation: A duplication factor, a minus sign (-), or a constant appears with a variable in the indicated command.

AFF691E LITERAL OR NUMERIC CONSTANTS ARE NOT PERMITTED IN THE "DESCRIBE" COMMAND

Explanation: Only variables and array names can be specified on the DESCRIBE command.

AFF700E NULL VARIABLE LIST/RANGE SPECIFIED IN "*text*"

Explanation: While scanning the previous command, the end of the command was found before a list or range specification was completed.

AFF715E "*variable*" IS NOT A LOGICAL VARIABLE

Explanation: While scanning the previous command, a logical variable was expected, but the indicated variable, which is not a logical variable, was found instead.

AFF720E "*condition*" HAS AN INVALID CONDITION

Explanation: The indicated condition is not syntactically correct. For example, ".EE." may have been used instead of ".EQ.."

AFF725E INVALID COMBINATION OF DATA TYPES IN CONDITION "*condition*"

Explanation: While scanning the syntax of the previous command, an invalid combination of data type was found within the condition specification. The data types of the variables must be the same or compatible.

AFF730E "*condition*" IS INVALID CONDITION SYNTAX

Explanation: In the specification of an arithmetic condition, either the right side of the condition has been omitted, or some extraneous data follows what appears to be a complete condition.

AFF735E CONDITION "*condition*" HAS AN INVALID FORTRAN TERM

Explanation: A duplication factor is specified within a condition specification. This is not valid.

AFF740E "*text*" IS INVALID "IF" SYNTAX

Explanation: Invalid syntax has been detected within an IF command. For example, parentheses may be missing around the condition definition.

AFF741E A USERID MUST BE SPECIFIED WITH THE MSG OPERAND; NO DEFAULT IS AVAILABLE

Explanation: TERMIO MSG was entered and the default user ID cannot be determined.

User Response: If the MSG operand is desired, specify the desired user ID.

AFF742W "MSG" OPERAND IGNORED; "MSG" IS NOT VALID OUTSIDE BATCH MODE

Explanation: A TERMIO command containing a MSG operand was entered while Interactive Debug was being used interactively. The MSG operand is only for batch mode.

AFF743E "*command*" COMMAND IS NOT PERMITTED IN BATCH MODE

Explanation: The indicated command was entered while in batch mode, but is not permitted there.

AFF745E "*text*" IS INVALID "WHEN" SYNTAX

Explanation: In the specification of an arithmetic condition, either the right side of the condition has been omitted, or some extraneous data follows what appears to be a complete condition.

AFF750E A COMMAND MUST BE SPECIFIED AFTER "*text*"

Explanation: No command was specified after the condition on an IF statement.

AFF755E "QUALIFY" IS NOT PERMITTED IN AN "IF" COMMAND

Explanation: QUALIFY cannot be the command specified as the action to be taken if the condition specified in an IF command is true.

AFF760E "*text*" REQUIRES AN OPERAND; THE COMMAND IS IGNORED

Explanation: An operand must be specified on the indicated command.

AFF765E "*condition*" HAS AN INVALID "WHEN" CONDITION NAME

Explanation: The name of the indicated condition has a syntactically invalid name. Valid names must begin with an alphabetic character and contain no more than four alphameric characters.

AFF768W "*condition*" CONDITION IS NOT ON

Explanation: An attempt to turn off the indicated condition has been detected, but the condition is already off.

AFF775E *"command"* COMMAND IGNORED IN AN IF OR COMMAND LIST

Explanation: The indicated command was found in an IF command or a command list specified with an AT command. The command is not valid in this context and is ignored.

AFF780E *"command"* IS NOT PERMITTED TO HAVE OPERANDS; THE COMMAND IS IGNORED

Explanation: A keyword has been specified for a command that has no keywords. The extra keyword is ignored.

User Response: Determine why the extra keyword was entered. Possibly the wrong command was issued. If so, issue the right command.

AFF795E INVALID COMBINATION OF DATA TYPES IN *"text"*

Explanation: In a SET command, the data type of the variables and constants on the right side of the equal sign is different from the data type of the variable on the left side of the equal sign.

- A character variable may be assigned only character data items.
- A logical variable may be assigned only logical data items.
- An arithmetic variable may be assigned only arithmetic data items.

AFF800E INVALID "GO"; THE STATEMENT IDENTIFIER IS NOT IN THE CURRENT PROGRAM UNIT

Explanation: An attempt has been detected to go to a statement outside the program unit that was being run when processing was suspended.

User Response: Reenter the GO command either without an ISN or sequence number, or with an ISN or sequence number that is within the correct program unit.

AFF801I QUALIFICATION IS *program*

Explanation: This message is issued in response to a QUALIFY command, and identifies the currently qualified program unit. Unless you have issued a QUALIFY command to set a different program unit, the currently qualified program unit will be the program unit which is currently being run.

AFF802E COMMAND IGNORED; THE PROGRAM HAS FINISHED EXECUTION

Explanation: The VS FORTRAN program has finished execution. Interactive Debug will allow you to enter most commands before you enter the QUIT command, but the last command entered is not one of those that may be issued at this time.

User Response: Enter a valid command. Valid commands are:

```
ANNOTATE      POSITION
AUTOLIST      PROFILE
BACKSPACE     PREVDISP
CLOSE         PURGE
COLOR         QUALIFY
comment       QUIT
DESCRIBE      RECONNECT
ENDFILE       REFRESH
HELP          RESTART
LIST          REWIND
LISTBRKS      SEARCH
LISTFREQ      SET
LISTINGS      SYSCMD
LISTSAHP      TERMIO
LISTSUBS      WHERE
LISTTIME      WINDOW
MOVECURS
```

For a list of commands that are invalid, see "Issuing Commands after a Program Runs" on page 84.

AFF805W *program* **IS OPTIMIZED; "GO" WITH STATEMENT ID MAY CAUSE UNPREDICTABLE RESULTS**

Explanation: The indicated program unit was compiled with OPT(n) with n>0. Because the program is optimized, it may depend on values being kept in registers between some statements. A GO command to a specific statement may bypass code that is needed to set registers, and may cause unpredictable results.

System Action: Message AFF806 is issued to confirm whether the command should be run.

AFF806A DO YOU WISH TO EXECUTE THIS COMMAND? (YES OR NO)

Explanation: This message was preceded by message AFF805, which warned of the possible consequences of running the GO command. You must now confirm your desire to issue the command.

User Response: Reply YES or NO.

System Action: If YES is specified, the command is issued. If NO is specified, the command is not issued. Following either action, processing continues.

AFF820E OPEN ERROR ON AFFOUT FILE

Explanation: This message is issued to your SYSMSG file or spooled console when the OPEN of the AFFOUT file fails while running batch mode.

User Response: Correct the cause of the I/O error.

AFF821E ERROR PROCESSING AFFOUT FILE

Explanation: This message is issued to your SYSMSG file or spooled console when a PUT to the AFFOUT file fails while running batch mode.

User Response: Correct the cause of the I/O error.

AFF822E OPEN ERROR ON AFFIN FILE

Explanation: This message is issued when the OPEN of the AFFIN file fails while running batch mode.

User Response: Correct the cause of the I/O error.

System Action: Debugging is terminated.

AFF823E ERROR PROCESSING AFFIN FILE

Explanation: This message is issued if a GET to the AFFIN file fails while running batch mode.

User Response: Correct the cause of the I/O error.

System Action: Debugging is terminated.

AFF824E END-OF-FILE ON AFFIN FILE

Explanation: This message is issued when end-of-file is reached on the AFFIN file for batch mode.

User Response: Correct the input file and rerun the job if more commands are desired.

System Action: A QUIT command is forced.

AFF825E UNABLE TO WRITE DIAGNOSTIC MESSAGE CONCERNING AFFIN FILE

Explanation: This message is issued to your SYSMSG file or spooled console when an AFFIN diagnostic cannot be written to the AFFOUT file in batch mode.

User Response: Correct the cause of the I/O error.

AFF831E "ENTRY" AND "EXIT" ARE NOT PERMITTED IN A RANGE

Explanation: An ENTRY or EXIT keyword was issued in a statement ID range.

User Response: Use ISNs or sequence numbers for ranges. ENTRY and EXIT are usable only as individual elements.

AFF832E THE QUALIFIER ON THE RIGHT SIDE OF A RANGE MUST MATCH THE QUALIFIER ON THE LEFT SIDE

Explanation: A statement ID range was entered where a qualifier was specified for the right statement ID that does not match the one on the left side.

User Response: Use a matching qualifier or allow it to default.

AFF840E LIST RANGE IGNORED; THE SECOND VARIABLE PRECEDES THE FIRST IN STORAGE

Explanation: A LIST command has been entered with a range of variables specified. In a range, the first variable must appear in storage prior to the second variable so that the area between the two variables may be displayed.

AFF841I RANK = *number*; DUMMY ARRAY ARGUMENT OF INACTIVE SUB-PROGRAM OR ALTERNATE ENTRY POINT

Explanation: This message is produced by the DESCRIBE command for a dummy array argument of an inactive subprogram or alternate entry point.

AFF850I *variable value*

Explanation: This is the output of a LIST command that did not include the DUMP keyword. The name of the requested variable is shown along with its current value.

AFF851I PROGRAM UNIT COMPILER OPT HOOKED TIMING

Explanation: This is the title line for LISTSUBS output.

AFF852I ENTRY POINT TOTAL TIME %TOTAL INVOCATIONS AVERAGE TIME

Explanation: This is the header message for LISTTIME output for program units.

AFF853I *entry-point-name number prcnt number number*

Explanation: This is a line of LISTTIME output for program units. It shows:

- the name of the entry point into the program
- the execution time of the program (in microseconds)
- the percent of total execution time for the program
- the number of invocations of the program unit
- the average time per invocation of the program unit (in microseconds)

AFF854I NO TIMING INFORMATION IS AVAILABLE

Explanation: This message is issued if no program units are timing or have accumulated time.

AFF855I *name datatype*

Explanation: This is the DESCRIBE output for a scalar variable.

AFF856I RANK = *number*, SIZE = *number* ELEMENTS

Explanation: This message provides DESCRIBE information about the size of an array variable.

AFF857I DIM *number*, LBOUND = (*number*), UBOUND = (*number*)

Explanation: This message provides DESCRIBE information about the dimensions of an array variable.

AFF859I DIMENSION INFORMATION NOT AVAILABLE

Explanation: Dimension information is not available for dummy arguments of an inactive subprogram or for an alternate entry point.

AFF860E FIXUP IGNORED; SUBSCRIPT ERROR

Explanation: A subscript specified within a FIXUP command was out of range for the variable it was subscripting.

AFF861E FIXUP IGNORED; NOT IN ERROR EXIT

Explanation: You can issue a FIXUP command only if a run-time error has occurred in the VS FORTRAN program.

AFF862E FIXUP IGNORED; NO ARGUMENTS MAY BE MODIFIED

Explanation: A FIXUP command with either ARG1 or ARG2 (or both) specified was entered for an error that has no modifiable arguments.

User Response: Enter a corrected FIXUP command or a GO command.

AFF863E FIXUP IGNORED; ARG2 MAY NOT BE MODIFIED

Explanation: A FIXUP command has been entered with a value specified for ARG2, but the VS FORTRAN library subroutine that detected the error only has one modifiable argument.

AFF865E "GO" WITH A STATEMENT ID IS NOT ALLOWED IN AN ERROR EXIT

Explanation: This type of GO command is not allowed within an error exit.

User Response: If you want to continue processing at some other statement, you may issue a NEXT command followed by a GO command. When execution is suspended because of the NEXT command, you may issue the GO command with a specific statement identification.

AFF866E LAST COMMAND IGNORED DUE TO AN ERROR EXIT

Explanation: The last command entered within an attention exit was not run because an error exit occurred.

AFF867E ERROR EXIT: ERROR *number* **AT** *name.number*

Explanation: This is the notification message that occurs when an error exit is taken. It is also the response to a WHERE command in an error exit. The indicated statement and program unit name identify the last statement that was running prior to the indicated error.

User Response: None required. Use the FIXUP or GO command to terminate the error exit processing.

AFF868E *"text"* **IS INVALID I/O COMMAND SYNTAX**

Explanation: The syntax of the previous BACKSPACE, CLOSE, ENDFILE, or REWIND command is invalid. An integer variable or constant must be specified.

AFF871I WHEN: *condition* **SATISFIED**

Explanation: The indicated WHEN condition has been satisfied.

System Action: This message will be followed by message AFF872I.

AFF872I CURRENTLY AT *name.number*

Explanation: This message is issued after message AFF871I to identify the current location within the program unit.

AFF873I PROGRAM TERMINATED BY USER REQUEST

Explanation: This message is issued when you enter the "QUIT" command in an attention exit to terminate execution of a VS FORTRAN program.

System Action: Control is returned to Interactive Debug.

AFF874E ERROR ON AFFIN FILE; FILE IGNORED

Explanation: An error has occurred while attempting to open the AFFIN file. No further attempts to access this file are made.

User Response: Correct the problem that caused the message. If the file is required, correct the problem and re-invoke Interactive Debug.

AFF875E WHEN IGNORED; CONDITION "*condition***" NOT DEFINED**

Explanation: A WHEN command has been detected that contains only a condition name. This is interpreted as a request to "turn the condition on." However, no condition with the specified name has been defined.

AFF876E WHEN IGNORED; SUBSCRIPT ERROR

Explanation: A subscript error was encountered while processing the WHEN command.

AFF880E OFFWN IGNORED; NO WHEN CONDITIONS ARE DEFINED

Explanation: The previous OFFWN command was ignored because there are no WHEN conditions defined that could be turned off.

AFF881E OFFWN IGNORED FOR UNDEFINED CONDITION "*condition***"**

Explanation: The previous OFFWN command specified a WHEN condition that does not exist. The command is ignored.

AFF900E SYSTEM COMMAND RETURN CODE: *code*

Explanation: The indicated code was received from processing the previous system command using Interactive Debug's SYSCMD command.

AFF910I CURRENT TERMIO STATUS: *status*

Explanation: This message indicates the status of settings that are controlled by the TERMIO command.

AFF920E SET COMMAND NOT COMPLETED NORMALLY

Explanation: Because of an error described in the preceding message, the SET command was unable to be completed; not all values were assigned.

AFF915I CURRENT DBCS STATUS: *status*

Explanation: Provides the current DBCS status in response to the DBCS command with no operands.

AFF925E TOO MANY VALUES; EXCESS IGNORED; SET COMPLETED

Explanation: Too many values for an array have been specified with a SET command. The extraneous values are ignored.

AFF929E MULTIPLE FORTRAN MAIN PROGRAMS HAVE BEEN FOUND; PROGRAM "program-unit-name" IS ALSO A MAIN PROGRAM

Explanation: Multiple main FORTRAN program units were found in the program being debugged. Only programs with a single main program unit can be debugged.

System Action: Processing continues. Interactive Debug assumes that the last main program unit in the load module is the main program that was invoked. If this is not the case, message AFF937E will be issued.

User Response: Relink-edit the program to contain only one main program unit.

AFF930E NAME "name" GREATER THAN 31 CHARACTERS; TRUNCATED

Explanation: A program unit name in the AFFON file is greater than 31 characters. The name is truncated to 31 characters.

User Response: If the name is misspelled, correct the spelling in the AFFON file and re-invoke the debugger.

AFF931E INVALID FILE NAME IN AFFON ENTRY FOR *"program-unit-name".*

Explanation: A data set name or file identifier specified in the AFFON file entry for the named program unit (or ALL) is invalid.

System Action: The data set name or file identifier is ignored. For a program unit name entry, the default name will be used. For an ALL entry, the default name will remain unchanged.

User Response: The data set name or file identifier is misspelled. Check to make sure the name is fully qualified and enclosed in quotes.

AFF932E DUPLICATE NAME "*name*"

Explanation: The indicated program unit name has been found more than once in the AFFON file.

System Action: The duplicate entry is ignored, and processing continues.

AFF933I THE AFFON FILE WAS PROCESSED WITH *number* ERRORS

Explanation: This is an informational message informing you that the AFFON file was processed. *number* is the number of errors found from parsing the file. Additional errors may be printed after this message indicating that certain data in the AFFON file is not valid for certain program units.

AFF934E PROGRAM NAME "*program*" NOT FOUND

Explanation: The indicated program unit name was specified in the AFFON file, but could not be found in the program to be debugged.

System Action: The program unit name is ignored, and processing continues.

AFF935E INTERNAL ERROR IN GENERATED CODE FOR "*program*"

Explanation: This is an internal error and should never occur. It indicates that, within the first four bytes generated by the VS FORTRAN Version 1 or VS FORTRAN Version 2 compiler for a VS FORTRAN statement, an instruction of the form "BALR x,y" was detected. Such an instruction is only valid if y=0.

User Response: Contact your IBM representative.

System Action: Debugging is discontinued.

AFF937E HOOK FOUND AT UNKNOWN LOCATION

Explanation: This is probably an internal error. It indicates that what appears to be an Interactive Debug hook was found at a location that Interactive Debug could not recognize.

User Response: Contact your IBM representative.

System Action: Interactive Debug returns control to the VS FORTRAN Version 2 Library for its normal program check handling.

AFF938E ADDRESSING MODE CHANGED IN *program unit name*

Explanation: An external routine called from a VS FORTRAN program unit has changed the MVS/XA addressing mode and not restored it. Unless this is an unusual or intentional situation, an abend will probably occur. Be aware that there are some situations in which an abend will occur before Interactive Debug can get control and issue this message. For example, a program running above the 16-megabyte line that switches to 24-bit addressing mode will abend immediately.

There are situations, however, where this message will be issued as informational only. For example, the Release 4 VS FORTRAN Version 1 library always enters user error exits in 31-bit addressing mode. If an MVS/XA program is linked to run 24/24, this message will be issued the first time such a routine is

entered. In this case, it should be treated as informational only, and not regarded as an error condition.

User Response: Unless the change of addressing mode was intentional, or unless a user exit was taken by a 24/24 program, you should check and correct the addressing mode logic in any external routines called by the specified program unit.

AFF940I ZERO-FREQUENCY STATEMENTS

Explanation: This is the heading of a listing of statements within the indicated program unit that have not been run, in response to LISTFREQ.

System Action: The ISNs or sequence numbers are listed.

AFF942I *name.statement*

Explanation: This is the output of the LISTFREQ command when the ZEROFREQ keyword has been specified.

AFF943I NONE

Explanation: One of the following is true:

- A LISTFREQ command with the ZEROFREQ keyword was entered, and all statements in the currently qualified program unit have been run at least once.
- A LISTBRKS command was entered, and no breakpoints have been established using the AT command.
- A LISTBRKS command was entered, and no WHEN conditions have been established.

AFF945I STATEMENT FREQUENCY

Explanation: This is the heading for a listing of execution counts, in response to LISTFREQ.

System Action: The ISNs or sequence numbers are listed.

AFF947I *name.number frequency*

Explanation: This is the output of the LISTFREQ command.

AFF950I CURRENT BREAKPOINTS:

Explanation: This is the heading for the output of the LISTBRKS command for breakpoints set by the AT command.

System Action: All active breakpoints are listed.

AFF952I *name.statement*

Explanation: This is the output from the LISTBRKS command for AT commands with a COUNT of 1.

AFF953I *name.statement* **COUNT(***count***)**

Explanation: This is the output from the LISTBRKS command for AT commands with a COUNT greater than 1.

AFF955I CURRENT WHEN CONDITIONS:

Explanation: This is the heading for the output of the LISTBRKS command for WHEN conditions.

AFF957I *condition name setting condition*

Explanation: This is the output from the LISTBRKS command for WHEN conditions. Each defined condition is shown, along with an indication of whether the condition is currently being monitored (ON) or not (OFF).

AFF960W COMMAND WILL BE ATTEMPTED, BUT PROGRAM HAS FINISHED EXECUTION

Explanation: The previous command has been accepted and will be run normally, but execution of the program has completed and may not be restarted.

AFF970W *program* **IS OPTIMIZED; SETS MAY BE INEFFECTIVE AND REFERENCES MAY GET INVALID VALUES**

Explanation: The indicated program unit has been compiled with the OPT(n) or VECTOR(n) compiler option with $n > 0$. Because the program unit is optimized, Interactive Debug commands that reference storage locations may produce unexpected results. For example, LIST A will report the value in the storage location the compiler established for variable A, but the compiled code may not be using the storage location to keep the value of the variable; it may be kept in a register instead. Interactive Debug cannot detect this situation, but it is likely to occur in optimized program units.

User Response: Use affected commands with care. See "Debugging Optimized and Vectorized Code" on page 86, which discusses the effect of optimized code on Interactive Debug.

System Action: Processing continues. This message should only appear the first time you issue one of the affected commands in a program unit.

AFF971E STATEMENT *number* **IS COLLAPSED AND MAY NOT BE USED FOR DEBUGGING**

Explanation: The indicated statement cannot have a breakpoint associated with it because it occupies no storage (because of optimization or vectorization).

User Response: If necessary, set a breakpoint at a statement immediately before or after the collapsed statement. You can use LISTFREQ to show which statements are collapsed.

AFF972I STATEMENT *number* DOES NOT HAVE AN ESTABLISHED BREAKPOINT

Explanation: An OFF command specifies that a breakpoint at the indicated statement is to be removed. There is no breakpoint at this statement.

AFF973I "OFF" IGNORED; NO BREAKPOINTS ARE DEFINED

Explanation: An OFF command has been issued, but there are no breakpoints defined in the currently qualified program unit.

AFF975E *command* COMMAND IGNORED; NO CURRENT QUALIFICATION

Explanation: There is no program unit serving as the current qualification. Perhaps execution was suspended before any debuggable program was executed.

User Response: Set a qualification using the QUALIFY command, and reenter the command causing the message.

AFF976E SYNTAX ERROR IN *command* COMMAND

Explanation: A syntax error was detected in scanning the indicated command.

User Response: Determine the cause of the error, and reenter the command with the correct syntax.

AFF977E I/O UNIT NUMBER IN "*command*" MUST BE AN INTEGER

Explanation: While scanning the previous command, where an integer value or integer variable was expected, a duplication factor or some other type of data was found.

User Response: Reenter the command with an integer value or the name of an INTEGER variable.

AFF978E COMMAND IGNORED; "ENDDEBUG" HAS BEEN ISSUED

Explanation: The last command entered is being ignored because of a previously issued ENDDEBUG command. After ENDDEBUG is issued, you no longer have the ability to debug.

User Response: You may issue QUIT if you no longer want to run the program, or may enter a null line to continue execution.

AFF979E "*command*" IGNORED; TERMINAL INPUT PENDING

Explanation: A GO, ENDDEBUG, or STEP command was entered, but is not permitted when a program is waiting for input.

User Response: If you want to enter a GO, ENDDEBUG, or STEP command, enter a NEXT command now, and, when execution stops for the NEXT, enter the GO or ENDDEBUG command.

AFF980E THE "STEP" OPERAND MUST BE AN INTEGER GREATER THAN ZERO

Explanation: STEP was entered with an invalid operand.

AFF981I ALL BREAKPOINTS IGNORED FOR PROGRAM UNIT "*name*"

Explanation: A dynamically loaded program unit with pending breakpoints was found to be loaded in read-only storage.

AFF982I NO BREAKPOINTS ARE DEFINED WITHIN THE SPECIFIED RANGE

Explanation: The OFF command specified a range where no breakpoints were defined.

AFF990I START OF RUN OR NO DEBUGGABLE PROGRAM EXECUTING

Explanation: A WHERE command has been detected, but either only the first prompt has been received and execution has not really started, or execution was interrupted while a nondebuggable program was executing.

AFF991I *program* **CALLED AT** *name.number*

Explanation: This message is the output of the WHERE command with the TRBACK keyword. The message indicates the logic flow through the program units.

AFF992I NO SUBROUTINES CALLED

Explanation: This message is issued following a WHERE TRBACK command when no subroutines have been called. This implies that execution is currently within the topmost debuggable program unit (usually the "main" program).

AFF995I WHERE: *name.number*

Explanation: This is the output of the WHERE command. It shows that execution is currently at the indicated statement in the indicated program unit. If execution is currently within a program unit that is not debuggable (and execution was suspended because of an attention interrupt), then the specified statement will be the last statement executed in a debuggable program unit.

AFF996I TO: *name.number* **FROM:** *name.number*

Explanation: This message is the output of the WHERE command with the FLOW keyword. The message indicates the logic flow through the program units.

AFF998E INTERNAL ERROR *number* **AT LOCATION** *number*; **IAD STORAGE MAY HAVE BEEN OVERLAID BY PROGRAM**

Explanation: This message is issued when an internal Interactive Debug abend occurs.

User Response: Contact your IBM representative.

System Action: Abend 101 is issued.

Full screen mode messages

These messages appear only in full screen mode. They are displayed below the command line, and will not appear in the log.

AFFA001E "AT" OR "OFF" ARE THE ONLY VALID PREFIX COMMANDS

Explanation: An invalid prefix command was detected. In the prefix command area, only the AT and OFF commands are allowed.

User Response: Press ENTER or type a valid command.

AFFA002E *string* **PLACED ON A NON-EXECUTABLE STATEMENT**

Explanation: The statement corresponding to where the prefix command was entered is not an executable statement.

AFFA004E THE ONLY VALID LISTING PANEL COMMANDS ARE "END" OR "RETURN"

Explanation: While in the listings panel, Interactive Debug commands are not recognized.

User Response: To issue Interactive Debug commands; first type END or RETURN in the listings panel to return to the main panel.

AFFA006I SEARCH HAS BEEN CONTINUED FROM TOP OF AREA

Explanation: The target of the SEARCH command was not found between the starting point of the search to the end of the object, so the search for the string wrapped to the top of the object.

AFFA007I NO VEC(REP) MESSAGE

Explanation: There is no vector message for the line pointed to by the cursor.

AFFA008E TARGET OF *command* **NOT FOUND**

Explanation: The argument of the POSITION or SEARCH command was not found.

AFFA009E NO PREVIOUS SEARCH ARGUMENT EXISTS; SEARCH NOT PERFORMED

Explanation: A SEARCH command was issued without an operand and there was no previous instance of a search. Thus, Interactive Debug has no string for which to search.

AFFA011I THIS IA A DISPLAY OF A SAVED PANEL. ALL COMMANDS WILL BE IGNORED

Explanation: By issuing the PREVDISP command, Interactive Debug will display the last displayed ISPF panel of the application program. The saved panel display is not an active panel; it is merely a picture of the display information. Thus, any commands or modifications made on this panel will have no effect.

AFFA015I CURSOR MUST BE IN THE OBJECT AREA OF THE WINDOW

Explanation: A cursor sensitive command was issued but the cursor placement was not in the correct area of the window for the operation to take place.

User Response: Place the cursor in one of the three windows where the operation is to take place.

AFFA018I CURSOR MUST BE IN LOG, SOURCE OR MONITOR WINDOW

Explanation: A cursor sensitive command was issued but the cursor was not in a window.

User Response: Place the cursor in one of the three windows where the operation is to take place.

AFFA019I CURRENT COMMAND IS INCOMPLETE, PENDING MORE INPUT

Explanation: When Interactive Debug recognizes that a block of commands are being entered, it will display this informational message until the command block is closed by an END statement.

AFFA020I ONLY COMMAND LINE INPUT IS ACCEPTED DURING PENDING MODE

Explanation: When the command continuation mode is on, the only input areas recognized are the command area and the log window where command modification may take place.

AFFA030E VALID DISPLAY FIELD VALUES ARE: YES, NO FOR THIS FIELD.

Explanation: The only valid values are YES and NO.

AFFA037E *txtlib-name* **TXTLIB COULD NOT BE FOUND**

Explanation: In CMS, the specified txtlib name on the invocation panel could not be found on any of the accessed disks.

AFFA045I INTERACTIVE DEBUG HAS ENDED NORMALLY

Explanation: The Interactive Debug session has completed.

AFFA050I THE RETRIEVE QUEUE IS EMPTY

Explanation: No previous command was issued from the command line during the current debug session.

AFFA051I EACH WINDOW MUST HAVE UNIQUE LETTERS OF L, M, AND S

Explanation: Objects cannot simultaneously reside in multiple windows. Each window must contain a unique object.

AFFA052I *window-name* **WINDOW IS ALREADY OPEN**

Explanation: A WINDOW OPEN command was issued for a window which is already open.

AFFA053I *window-name* **WINDOW IS ALREADY CLOSED**

Explanation: A WINDOW CLOSE command was issued for a window which was already closed.

AFFA054I ONE WINDOW MUST BE OPEN AT ALL TIMES

Explanation: A WINDOW CLOSE command was given with only one window open. At least one window must be open at any time.

AFFA055I SPECIFY A WINDOW NAME ON THE *command-name* **COMMAND**

Explanation: A full screen command which requires one of the window object names as an operand was not given a window object name as an argument.

AFFA501S INSUFFICIENT STORAGE FOR ENVIRONMENT SETUP, CANNOT CONTINUE.

Explanation: During initialization, not enough storage was available to set up the execution environment.

AFFA502S FAILURE DURING PROGRAM LOAD. CANNOT CONTINUE.

Explanation: During initialization, several modules are dynamically loaded. If there were any kind of errors while loading system routines, this message will be issued.

AFFA503S TSO COMMAND NOT EXECUTED. ATTENTION HANDLER INVOCATION ERROR.

Explanation: In line mode TSO operation, an attention handler is established when the TSO command is run. This message will appear if the attention handler cannot be set up.

AFFA504S TSO COMMAND NOT EXECUTED. COMMAND NOT RECOGNIZED OR INVALID.

Explanation: In line mode TSO operation, an operand following the TSO command has been passed to TSO for execution. TSO doesn't recognize the operand as a command.

AFFA505S HELP COMMAND NOT EXECUTED. ATTENTION HANDLER INVOCATION ERROR

Explanation: In line mode TSO operation, an attention handler is established when the HELP command is run. This message will appear if the attention handler cannot be set up.

AFFA751E LISTING TAGS MISSING. LISTING MUST BE COMPILED WITH FORTRAN V2R3

Explanation: The listing file was not compiled with the VS FORTRAN Version 2 Release 3 compiler.

AFFA752E LISTING FILE COULD NOT BE OPENED.
AFFA753E LISTING DATASET COULD NOT BE OPENED.
AFFA754E LISTING COULD NOT BE LOADED INTO STORAGE.
AFFA755E INSUFFICIENT STORAGE TO LOAD LISTING.

Explanation: There is not enough storage available to load the listing for the program unit.

AFFA756E LISTING FILE COULD NOT BE FOUND ON ACCESSED DISKS.

Explanation: The default or the specified listing name could not be found on any of the CMS accessed disks.

AFFA757E LISTING DATASET COULD NOT BE FOUND IN ALLOCATED DATASETS.

Explanation: The default or the specified dataset name could not be found on any of the allocated datasets.

AFFA758E SPECIFIED PROGRAM UNIT NAME NOT FOUND IN LISTING.

Explanation: The compile unit name entered on the listing panel could not be found in the listing associated with it.

AFFA851E LOG FILE LRECL GREATER THAN 256; LOG WILL NOT BE CREATED.

Explanation: A user defined log file must have a logical record length greater than or equal to 32 bytes and less than or equal to 256 bytes.

AFFA852E LOG FILE LRECL LESS THAN 32; LOG WILL NOT BE CREATED.

Explanation: A user defined log file must have a logical record length greater than or equal to 32 bytes and less than or equal to 256 bytes.

AFFA853E ERROR WRITING TO LOG FILE.

Explanation: An I/O error occurred while trying to write to the log file.

AFFA854E LOG FILE COULD NOT BE OPENED.

Explanation: An I/O error caused some problem with creating the log file.

AFFS002 ERROR DETECTED BY IAD

Explanation: An error was detected while trying to process an Interactive Debug command.

User Response: Check the log window for an explanation.

AFFS003 THE FORTRAN LIBRARY DETECTED AN ERROR DURING THE PROGRAM EXECUTION

Explanation: An error in the program was detected by the VS FORTRAN library.

User Response: Check the log window for an explanation.

AFFS006 *filetype* **FILE NOT FOUND**

Explanation: Either the LOG or PRINT file was not found by Interactive Debug. Therefore, it cannot be browsed after IAD completion.

AFFS007A THE PARAMETERS USED ARE INVALID AND HAVE NO EFFECT WITH THIS COMMAND

Explanation: This command does not require an operand. The command is executed as if no operand was specified.

AFFS008A NO PREVIOUS SEARCH ARGUMENT EXISTS; SEARCH NOT PERFORMED.

Explanation: A SEARCH command was entered without operands, but there was no previous search argument to use as a default.

User Response: Reissue the command with an operand.

AFFS010 *file-id* **LOADS TOO LOW IN STORAGE TO RUN UNDER ISPF**

Explanation: A MODULE could not be loaded into storage for debugging since its origin is too low.

User Response: GENMOD the module at a larger address. (ORIGIN 22000 should be high enough.)

AFFS016 FUNCTION ERROR, RC = *rc*
LONG FORM: ERROR TRYING TO LOAD PROGRAM, RC = *rc*
CHECK IAD MANUAL FOR INFO.

Explanation: The VS FORTRAN program could not be invoked. The possible return codes from Interactive Debug are:

5 The OPEN failed when trying to use a LOADLIB.

6 The BLDL failed when trying to use a LOADLIB, other than the specified member was not found.

12 The LOADMOD failed when trying to use a MODULE, other than the MODULE origin too low.

99 The LOAD failed when trying to use a TEXT file or LOADLIB.

AFFS017 EXECUTED WITHOUT DEBUG
LONG FORM: NODEBUG SPECIFIED AS EXECUTION TIME OPTION, DEBUG NOT INVOKED

Explanation: Interactive Debug was not invoked because NODEBUG was specified as an execution-time option.

AFFS018 TXTLIB NOT FOUND
LONG FORM: SPECIFIED TXTLIB NOT FOUND ON ANY ACCESSED DISK

Explanation: One of the TXTLIBs specified on the invocation panel or either VSF2FORT or CMSLIB TXTLIB was not found when trying to invoke Interactive Debug.

User Response: Make sure that the specified TXTLIBs are properly accessed.

AFFS018A LOADLIB NOT FOUND
LONG FORM: VSF2LOAD LOADLIB NOT FOUND ON ANY ACCESSED DISK

Explanation: The VS FORTRAN Interactive Debug library was not found on any of your accessed disks.

User Response: Make sure the VS FORTRAN disk is properly accessed.

Appendix B. ISPF Dialog Variable Names Defined by Interactive Debug

Following are names of ISPF dialog variables defined by Interactive Debug, and should not be used in defining ISPF dialog variables in VS FORTRAN Version 2 programs.

AFFRSTRT	AMT_LITERAL	AQACRP_LITERAL
AQAHELP_LITERAL	AQAMENU	AQATRAP
CMD_LINE_FIELD	CMD_MODE_STRING	CNAME
COL1	COL2	COL3
CONFIG	COUNTL	CPOS
DYNAM_AREA_NAME	FREQUENCY_DISPLAY	ISPALRM
ISPCLST	ISPCRLST	ISPDSMS
ISPDSN	ISPFCMD	ISPFID
ISPHALT	ISPLINE	ISPLNUM
ISPLOGN	ISPLOGNO	ISPMO
ISPMONNO	ISPPGM	ISPSRFL
ISPWAIT	ISP1LST	ISP2LST
LAST	LINE	LISTING_TYPE
LOG_OUTPUT_TYPE	LOGDSN	LOGFID
MIXED_MODE_FIELD	MLOG	NO_LITERAL
PARM1	PARM2	PFKEY_NAME
PREVDISP_MARK	QUALIFY_FIELD	REFRESH_FIELD
RESET_LITERAL	RESTFID	RETCBUF
RETCLEN	ROW1	ROW2
ROW3	RST	RSTDSN
SAVE_LITERAL	SCREEN_WIDTH	TABLE_LINE
VGET_BUFFER	WHERE_FIELD	WINDOW1
WINDOW2	WINDOW3	W1OPEN
W2OPEN	W3OPEN	YES_LITERAL
ZAPPLID	ZHILITE	ZKEYS
ZSCREEND	ZSCREENW	ZSCRMIN2
ZSCROLLN_LITERAL	ZSPLIT_LITERAL	ZTDSELS_LITERAL
ZUSER	ZVERB	

Figure 45. ISPF Dialog Variable Nmaes Defined by Interactive Debug

Glossary

This glossary includes definitions developed by the American National Standards Institute (ANSI), and the International Organization for Standardization (ISO).

This material is reproduced from the American National Dictionary for Information Processing, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

An asterisk (*) to the right of an item number indicates an ANSI definition in an entry that also includes other definitions.

The symbol "(ISO)" at the beginning of a definition indicates that it has been discussed and agreed upon at meetings of the International Organization for Standardization Technical Committee 97/Subcommittee 1 (Data Processing Vocabulary), and has also been approved by ANSI and included in the *American National Dictionary for Information Processing*.

A

addressing mode. The length of an address, either 24 bits or 31 bits, used by the processor. Indicated by the high-order bit of the PSW in an MVS/XA environment.

alphabetic character. A character of the set A, B, C, ... Z. See also "letter." In VS FORTRAN Version 1 and VS FORTRAN Version 2, the currency symbol ($) is considered an alphabetic character. In VS FORTRAN Version 2, lowercase letters (a through z) are also valid.

alphameric. Pertaining to a character set that contains letters (A through Z) and digits (0 through 9) only. In VS FORTRAN Version 2, the character set may also contain lowercase letters (a through z).

alphameric character set. A character set that contains both letters and digits.

alternate entry. (1) In VS FORTRAN, an entry provided by means of the ENTRY statement. (2) As used by Interactive Debug, an entry other than the one by which the subprogram was actually entered.

analyzable DO loop. See also "DO loop." A DO loop that is eligible for vectorization by the compiler. The loop may or may not have been vectorized, depending on compiler directives used and decisions made by the compiler based on cost analysis.

animation. In ISPF with Interactive Debug, the ability to highlight the command currently running and control the pace of execution when using the STEP command. This creates an "animated" picture of your program's execution.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function.

arithmetic constant. A constant of type integer, real, double-precision, or complex.

arithmetic expression. One or more arithmetic operators and/or arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs VS FORTRAN to perform an arithmetic operation. The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

array. An ordered set of data items identified by a single name.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The name of an ordered set of data items that make up an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be of type character, logical, or arithmetic. When the assignment statement is run, the expression to the right of the equal

sign replaces the value of the variable or array element to the left.

B

basic real constant. A string of decimal digits containing a decimal point, and expressing a real value.

blank common. An unnamed common block.

breakpoint. (1) (ISO) A place in a computer program, usually specified by an instruction, where its execution may be interrupted by external intervention or by a monitor program. (2) As used by Interactive Debug, a VS FORTRAN statement where the user has specified that execution is to be suspended, or that some action is to be taken.

C

character constant. A string of one or more characters enclosed in apostrophes. The delimiting apostrophes are not part of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character type. A data type that can consist of any characters; in storage, one byte is used for each character.

collapsed statement. A statement for which no machine code has been generated because of the nature of the statement, or as a result of optimization or vectorization.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. An approximation of the value of a complex number, consisting of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected file. A file that has been connected to a unit and defined by a FILEDEF command or by job control statements.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

continuation character. a hyphen (-), used to continue a command that is longer than the Interactive Debug command line. Enter it as the last character before continuation.

control statement. Any of the statements used to alter the normal sequential execution of FORTRAN statements, or to terminate the execution of a VS FORTRAN program. FORTRAN control statements are any of the forms of the GO TO, IF, and DO statements, or the PAUSE, CONTINUE, and STOP statements.

cursor-sensitive command. A command for which its operands can be specified by pointing to them with the cursor, instead of typing them on the command line. Applies to full screen mode debugging only.

D

data. (1) * (ISO) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. (2) In FORTRAN, data includes constants, variables, arrays, and character substrings.

data item. A constant, variable, array element, or character substring.

data set. The major unit of data storage and retrieval consisting of data collected in one of several prescribed arrangements and described by control information to which the system has access.

data set reference number. A constant or variable in an input or output statement that identifies a data set to be processed.

data type. The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character.

debugging hook. See also "DO loop analysis hook" and "hook." A type of hook used to give Interactive Debug control of a user's program for debugging purposes, such as for AT and WHEN breakpoints, tracing, and single stepping. These hooks are different from DO loop analysis hooks.

*** digit.** (ISO) A graphic character that represents an integer. For example, one of the characters 0 through 9.

DO loop. See also "analyzable DO loop." A range of statements run repetitively by a DO statement.

DO loop analysis hook. See also "debugging hook" and "hook." A type of hook used to give Interactive Debug control of the user's program for DO loop analysis purposes, such as for vector length and stride gathering and DO loop timing. These hooks are different from debugging hooks.

double-byte character. A character which is represented by two bytes.

double-byte data. Data consisting of double-byte characters.

double precision. The standard name for real data of storage length 8.

DO variable. A variable, specified in a DO statement, that is initialized or incremented prior to each execution of the statement or statements within a DO range. It is used to control the number of times the statements within the range are run.

dummy argument. A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

dynamic common. A VS FORTRAN named common which the DC compiler option specifies should be allocated at run- time.

E

executable program. A program that can be run as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

executable statement. A statement that causes an action to be taken by the program; for example, to calculate, to test conditions, or is a control statement.

existing file. A file that has been defined by a FILEDEF command or by job control statements. A valid unit number in FORTRAN's internal unit assignment table, as specified at installation time.

The INQUIRE statement considers a file to exist on the basis of FORTRAN I/O statements that have been processed.

existing unit. A valid unit number in FORTRAN's internal unit assignment table, as specified at installation.

expression. A notation that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. An expression can be arithmetic, character, logical, or relational.

external file. A set of related external records treated as a unit; for example, in stock control, an external file would consist of a set of invoices.

external function. A function defined outside the program unit that refers to it.

external procedure. A SUBROUTINE OR FUNCTION subprogram written in FORTRAN.

F

file. A set of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

file definition statement. A statement that describes the characteristics of a file to a user program. For example, the OS/VS DD statement or the FILEDEF command for CMS processing.

file reference. A reference within a program to a file. It is specified by a unit identifier.

formatted record. (1) A record, described in a FORMAT statement, that is transmitted, when necessary with data conversion, between internal storage and internal storage or to an external record. (2) A record transmitted with list-directed READ or WRITE statements and an EXTERNAL statement.

FORTRAN-supplied procedure. See "intrinsic function."

function reference. A source program reference to an intrinsic function, to an external function, or to a statement function.

function subprogram. A subprogram invoked through a function reference, and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

H

hexadecimal constant. A constant that is made up of the character Z followed by two or more hexadecimal digits.

hierarchy of operations. The relative priority order used to evaluate expressions containing arithmetic, logical, or character operations.

hook. See also "debugging hook" and "DO loop analysis hook." A segment of code, imbedded in the user's program, that gives control to Interactive Debug during execution of the program. This code is independent of the algorithm being implemented by the program. Hooks may be compiled in (by using the TEST option) or inserted during initialization.

I

IAD. Interactive Debug.

implied DO. An indexing specification, similar to a DO statement, causing repetition over a range of data elements. (The word DO is omitted, hence the term "implied.")

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type, capable of expressing the value of an integer. It can have a positive, negative, or zero value; it must not include a decimal point.

internal file. A set of related internal records treated as a unit.

internal statement number (ISN). A number assigned to each statement in a VS FORTRAN program by the VS FORTRAN compiler. ISNs are assigned sequentially beginning with 1, and are visible on the listing produced by the compiler.

ISN is sometimes referred to as "internal sequence number." See also "statement identifier" and "sequence number."

interruption localizing. A function that occurs at optimization level 2. It restricts certain optimizations so that no code is moved out of a loop if it would cause an interruption that would not occur without optimization.

intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations.

I/O. Pertaining to either input or output, or both.

I/O list. A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

L

labeled common. See "named common."

length specification. A source language specification of the number of bytes to be occupied by a variable or an array element.

letter. A symbol representing a unit of the English alphabet.

list-directed. An input/output specification that uses a data list instead of a FORMAT specification.

logical constant. A constant that can have one of two values: "true" or "false."

logical expression. A combination of logical primaries and logical operators. A logical operator can have one of two values: true or false.

logical operator. Any of the set of operators .NOT. (negation), .AND. (connection: both), or .OR. (inclusion: either or both), .EQV. (equal), .NEQV. (not equal).

logical primary. A primary that can have the value "true" or "false." See also "primary."

logical type. A data type that can have the value "true" or "false" for VS FORTRAN Version 1 or VS FORTRAN Version 2. See also "data type."

looping. Repetitive execution of the same statement or statements. Usually controlled by a DO statement.

M

main program. A program unit, required for execution, that can call other program units but cannot be called by them.

mixed character data. character data which contains both single-byte and double-byte data.

N

name. A string of from one through thirty-one alphameric characters, the first of which must be alphabetic. The underscore (_) is a valid character. Used to identify a constant, a variable, an array, a function, a subroutine, or a common block.

named common. A separate common block consisting of variables, arrays, and array declarators, and given a name.

nested DO. A DO statement whose range of statements is entirely contained within the range of another DO statement.

nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing information, or of statement functions, but does not cause an action to be taken by the program.

nonexisting file. A file that has not been defined by a FILEDEF command or by job control statements.

*** numeric character.** (ISO) Synonym for digit.

numeric constant. A constant that expresses an integer, real, or complex number.

P

preconnected file. A unit that was defined at installation time. However, a preconnected unit does not exist for a program if the unit is not defined by a FILEDEF command or by job control statements.

predefined specification. The implied type and length specification of a data item, based on the initial character of its name in the absence of any specification to the contrary. The initial characters I-N type data items as integer; the initial characters A-H, O-Z, and $ type data items as real. No other data types are predefined. For VS FORTRAN Version 1 and VS FORTRAN Version 2, the length of both types is 4 bytes.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

procedure subprogram. A function or subroutine subprogram.

program return code. When a program is terminated with a nonzero return code, the code is available for interrogation by means of job control language for the appropriate operating system.

program unit. A sequence of statements constituting a main program or subprogram.

R

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or zero value.

record. A collection of related items of data treated as a unit.

relational expression. An expression that consists of an arithmetic expression followed by a relational operator, followed by another arithmetic expression or a character expression followed by a relational operator, followed by another character expression. The result is a value that is true or false. In Interactive Debug the only arithmetic expression permitted in a relational expression is a variable, an array element, or a constant.

relational operator. Any of the set of operators that can express a comparison between arithmetic expressions, and that can be either true or false:

.GT. greater than
.GE. greater than or equal to
.LT. less than
.LE. less than or equal to
.EQ. equal to
.NE. not equal to

residence mode. Where a program resides in virtual storage in an MVS/XA environment: above or below 16 megabytes.

S

scale factor. A specification in a FORMAT statement that changes the location of the decimal point in a real number (and, if there is no exponent, the magnitude of the number).

sequence number. A number found in positions 73 through 80 of records containing source statements for the VS FORTRAN compiler. Sequence numbers are not necessarily unique, in sequence, or present in every record. See "internal statement number" and "statement identifier."

shift-in (SI) character. The character indicating the end of double-byte data. It has an internal representation of X'0F'.

shift-out (SO) character. The character indicating the beginning of double-byte data. It has an internal representation of X'0E'.

single-byte character. A character which is represented by a single byte. Also known as an EBCDIC character.

single-byte data. Data consisting of single-byte characters.

specification statement. One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies the information required to allocate data storage.

specification subprogram. A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

statement. The basic unit of a VS FORTRAN program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or

information about the program itself. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression. In the remainder of the program the name can be used as a substitute for the expression.

statement function definition. A statement that defines a statement function. Its form is a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement function reference. A reference in an arithmetic, logical, or character expression to the name of a previously defined statement function.

statement identifier. The statement label, internal statement number (ISN), or sequence number used by Interactive Debug to identify a statement in a VS FORTRAN program. The options that existed when the program was compiled determine whether the ISN or the sequence number is valid. In many cases, you can also use the statement label, preceded by a slash, as a valid identifier for a statement.

See also "internal statement number" and "sequence number."

statement label. A number of from one through five decimal digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO statement, or to refer to a FORMAT statement.

subprogram. A program unit that is invoked by another program unit in the same program. In FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subroutine subprogram. A subprogram whose first statement is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

*** subscript.** A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity. A component of a subscript: an integer constant, an integer variable, or an expression evaluated as an integer constant.

T

type specification. The explicit specification of the type of a constant, variable, array, or function by use of an explicit type specification statement.

U

unformatted record. A record that is transmitted unchanged between internal storage and an external record.

unit. A means of referring to a file in order to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit identifier. The number that specifies an external unit.

1. An integer expression whose value must be zero or positive. For VS FORTRAN Version 1 and VS FORTRAN Version 2, this integer value of length 4 must correspond to a DD name, a FILEDEF name, or an ASSGN name.
2. An asterisk (*) that corresponds on input to FT05F001 and on output to FT06F001.
3. The name of a character array, character array element, or character substring for an internal file.

V

variable. A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution.

vectorization. The process of creating machine instructions that will run on the special vector processing facility of the IBM 3090 Vector Facility.

vectorize. To compile a source program so its eligible DO loop statements are transformed into vector object code.

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Numerics

VS FORTRAN Version 2
Interactive Debug
Guide and Reference

SC26-4223-2

Reader's Comment Form

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Chapter/Section _____

_____ Page No. _____

Comments:

Note: Staples can cause problems with automatic mail-sorting equipment. Please use pressure-sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information.

Name _____ Phone No. (_____) _____

Company _____

Address _____

_____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

SC26-4223-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

VS FORTRAN Version 2
Interactive Debug
Guide and Reference

Reader's Comment Form

SC26-4223-2

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Chapter/Section _____

_____ Page No. _____

Comments:

Note: Staples can cause problems with automatic mail-sorting equipment. Please use pressure-sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information.

Name _____ Phone No. (_____) _____

Company _____

Address _____

_____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

SC26-4223-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

VS FORTRAN Version 2
Interactive Debug
Guide and Reference

Reader's Comment Form

SC26-4223-2

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Chapter/Section _____

_____ Page No. _____

Comments:

If you want a reply, please complete the following information.

Name _____ Phone No. (_____) _____

Company _____

Address _____

_____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4223-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

Program Number
5668-805
5668-806

File Number
S370-40

The VS FORTRAN Version 2 Library

Diagnosis Guide	LY27-9516
General Information	GC26-4219
Installation and Customization for MVS	SC26-4340
Installation and Customization for VM	SC26-4339
Interactive Debug Guide and Reference	SC26-4223
Language and Library Reference	SC26-4221
Licensed Program Specifications	GC26-4225
Programming Guide	SC26-4222
Reference Summary	SX26-3751

Printed in U.S.A.