High Level Assembler for z/OS & z/VM & z/VSE

Toolkit Feature Interactive Debug Facility User's Guide

Version 1 Release 6

GC26-8709-08

High Level Assembler for z/OS & z/VM & z/VSE

IBM

Toolkit Feature Interactive Debug Facility User's Guide

Version 1 Release 6

GC26-8709-08

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 291.

This edition applies to IBM High Level Assembler for z/OS & z/VM & z/VSE Toolkit Feature, Release 6, Program Number 5696-234 and to any subsequent releases until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality.

IBM welcomes your comments. For information on how to send comments, see "How to send your comments to IBM" on page xv.

Contents

Figures

About this book

This book provides information about how to use the IBM® High Level Assembler Toolkit Feature Interactive Debug Facility (IDF).

Throughout this book, we use these indicators to identify platform-specific information:

- Prefix the text with platform-specific text (for example, "Under CMS...")
- Add parenthetical qualifications (for example, "(CMS)")
- A definition list, for example:

  z/OS Informs you of information specific to z/OS®.

  z/VM Informs you of information specific to z/VM®.

  z/VSE Informs you of information specific to z/VSE®.

CMS is used in this manual to refer to Conversational Monitor System on z/VM.

Syntax notation

Throughout this book, syntax descriptions use this structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a statement.

  The ───► symbol indicates that the statement syntax is continued on the next line.

  The ►─── symbol indicates that a statement is continued from the previous line.

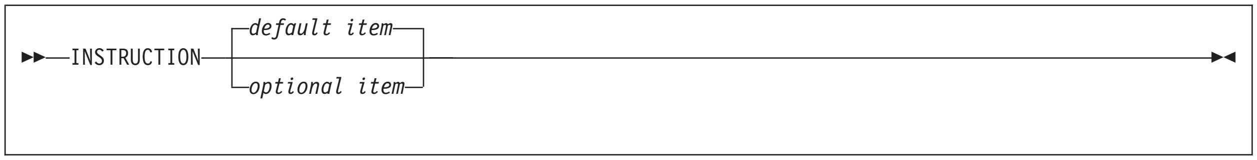
  The ───►◄ indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►─── symbol and end with the ───► symbol.
- **Keywords** appear in uppercase letters (for example, ASPACE) or uppercase and lowercase (for example, PATHFile). They must be spelled exactly as shown. Lowercase letters are optional (for example, you could enter the PATHFile keyword as PATHF, PATHFI, PATHFIL, or PATHFILE).

  Variables appear in all lowercase letters in a special typeface (for example, *integer*). They represent user-supplied names or values.
- If punctuation marks, parentheses, or such symbols are shown, they must be entered as part of the syntax.
- Required items appear on the horizontal line (the main path).

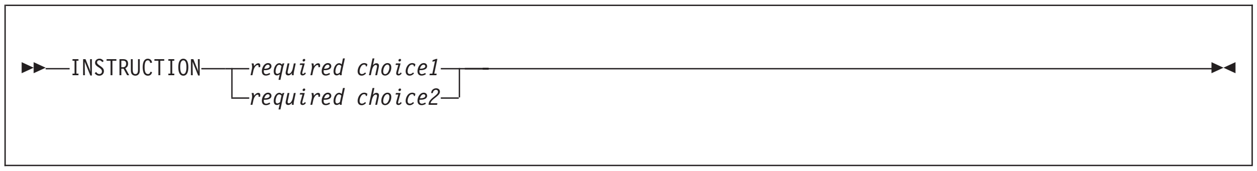
```
►►──INSTRUCTION──required item──────────────────────────────────────────►◄
```

- Optional items appear below the main path. If the item is optional and is the default, the item appears above the main path.

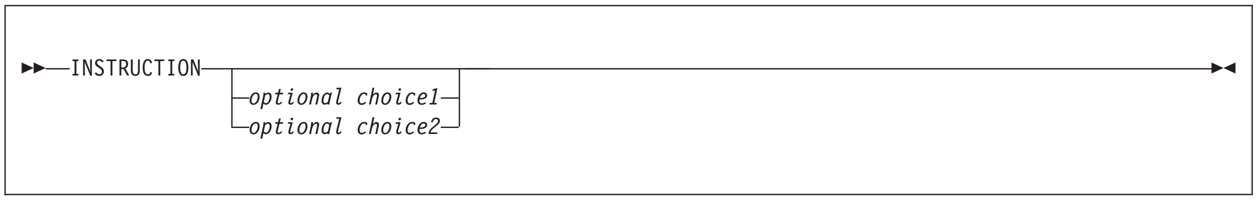


- When you can choose from two or more items, they appear vertically in a stack.

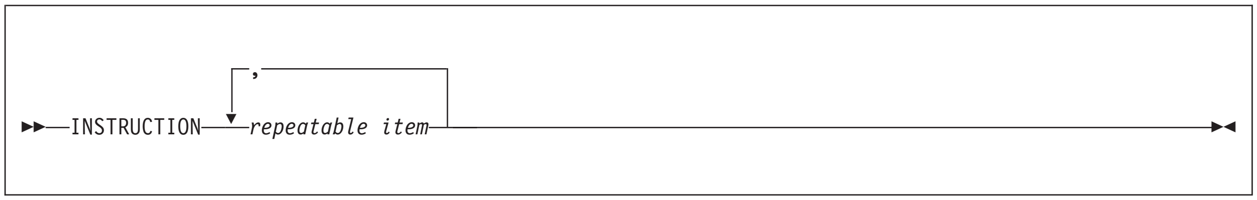
  If you **must** choose one of the items, one item of the stack appears on the main path.



  If choosing one of the items is optional, the whole stack appears below the main path.



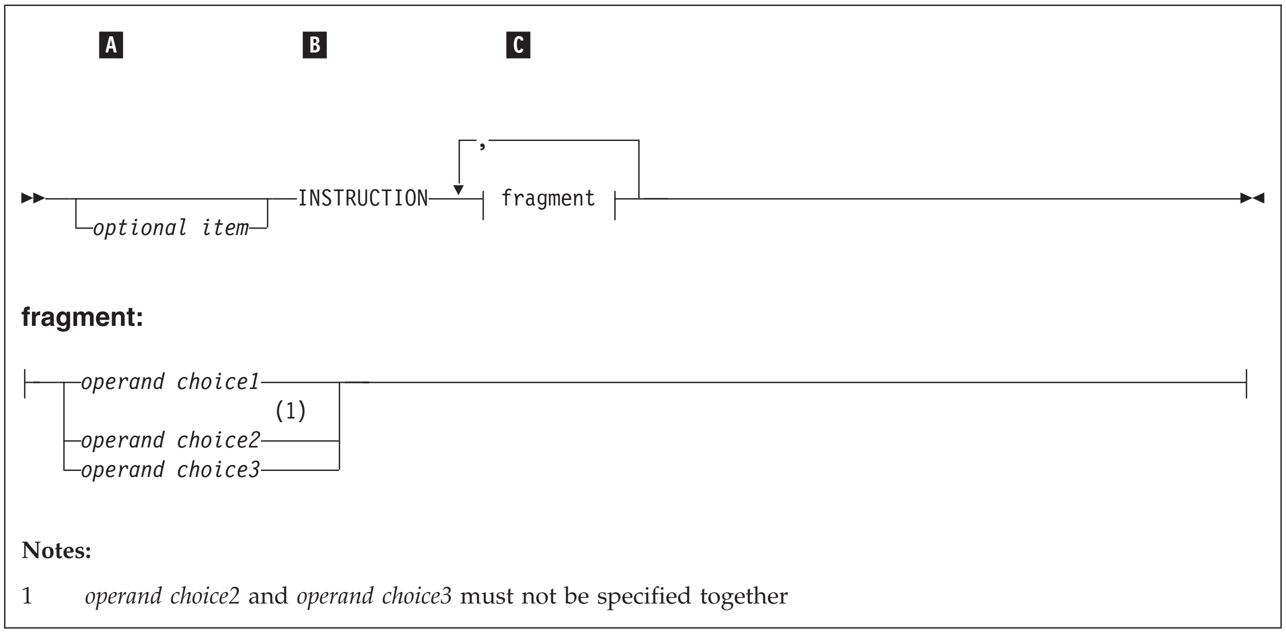
- An arrow returning to the left above the main line indicates an item that can be repeated. When the repeat arrow contains a separator character, such as a comma, you must separate items with the separator character.



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

Format

The following example shows how the syntax is used.



Notes:

1 *operand choice2* and *operand choice3* must not be specified together

A The item is optional, and can be coded or not.

B The INSTRUCTION key word must be specified and coded as shown.

C The item referred to by "fragment" is a required operand. Allowable choices for this operand are given in the fragment of the syntax diagram shown below "fragment" at the bottom of the diagram. The operand can also be repeated. That is, more than one choice can be specified, with each choice separated by a comma.

How to send your comments to IBM

If you especially like or dislike anything about this book, feel free to send us your comments.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information that is in this book and to the way in which the information is presented. Speak to your IBM representative if you have suggestions about the product itself.

When you send us comments, you grant to IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

You can get your comments to us quickly by sending an e-mail to **idrcf@hursley.ibm.com**. Alternatively, you can mail your comments to:

User Technologies,
IBM United Kingdom Laboratories,
Mail Point 095, Hursley Park,
Winchester, Hampshire,
SO21 2JN, United Kingdom

Please ensure that you include the book title, order number, and edition date.

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative
- Call IBM technical support
- Visit the IBM support web page

Summary of changes

There are no significant changes of content between this edition of the User's Guide and the previous edition.

The document has been reformatted to IBM's latest standards.

Part 1. What is IDF and how do I start using it?

Chapter 1. Introduction to the Interactive Debug Facility

The Interactive Debug Facility (IDF) is a symbolic debugging tool for Assembler Language programs. You can also use it to debug programs written in compiled languages, at the object code level.

The *Language Support Module* (LSM) functions provide for debugging Assembler Language programs at the source level. The ASMLANGX utility reads the ADATA file, produced by the High Level Assembler when you assembled your program, and builds a file that IDF can use for source-level debugging.

You do not have to learn a new command language to use IDF. All IDF's basic functions can be performed by using the default PF keys.

IDF is window oriented. The screen can be filled with windows of various types, including:
- Current Registers window
- Disassembly window
- Dump window
- LSM Information window

Windows can be mixed and matched in any order or combination on the screen. You can open multiple:
- Dump windows to view multiple areas of non-contiguous storage
- LSM Information windows
- Disassembly windows to view and modify your program at the object code level

These windows will show the source statements if they are available.

In general, the command that opens the window (for example, REGS for the Current Registers window) is used to close that window on the screen.

Capabilities

IDF can be used for disassembly-level (object code level) or source-level debugging of user programs under z/OS, CMS, and z/VSE.
- On z/OS
  - Programs can be “batch” or TSO Command Processors, with support provided for reentrant modules.
  - IDF provides for debugging programs (for example, ISPF programs) for which some environmental setup is needed.
  - Parameters are passed to your program using standard z/OS `EXEC PGM` or TSO Command Processor linkage conventions.
- On CMS
  - Programs can be a mix of user-area programs, transients, nucleus extensions, and self-relocating nucleus extensions.
  - Unless one of the debugged programs is a transient, CMS SUBSET is available throughout the debugging session.
  - IDF provides for debugging programs (for example, a REXX function package) for which some environmental setup is needed.
  - Parameters are passed to your program using standard CMS linkage conventions, with both tokenized and untokenized lists.
- On z/VSE
  - Parameters are passed to your program using standard z/VSE `EXEC PGM` linkage.

The following sections provide an overview of IDF's capabilities.

Execution control

You can single-step through a program. If a Disassembly window is open as the program progresses, the next instruction to be executed is highlighted.

You can use exit routines, written in REXX or compiled languages, to examine conditions surrounding a breakpoint and determine whether or not to inform you of its occurrence, or to ignore the breakpoint and continue execution.

By using the PATH option you can capture the number of times each instruction in your program is executed. This can be helpful in locating "dead code".

You can use the HISTORY command to review the last 1,023 machine instructions executed by your program. This is often useful in determining the circumstances leading to a "wild branch" or similar error.

An exit routine can use the MSTEP command to perform a dynamic path analysis of your program. (This analysis is slow, since part of a REXX exit routine is executed for each instruction your program executes.)

z/VM If Extended Control (EC) mode is available to IDF, PER is used to provide register or storage alteration stops. Breakpoints and single-step mode are available regardless of EC mode availability.

All SVC instructions issued by your program can be trapped.

If your program invokes a nucleus extension through BALR linkage, an option lets you single-step through that nucleus extension.

IDF can debug programs that "steal" the new PSWs, if your program's instructions which access these PSWs are declared to IDF.

Symbol support

You get symbol support by assembling with the ADATA option and using ASMLANGX to create an extraction file from the SYSADATA file.

This allows you to enter instruction and data addresses in symbolic notation, and show data areas with their associated labels, or in an unformatted display.

IDF provides a disassembly capability, with code section (CSECT) and external symbols intensified. Storage references are disassembled to their symbolic names wherever possible.

Conditional branch instructions are disassembled to their extended mnemonics unless you specify otherwise.

Typeover storage modification

When a Dump window is open or when a Disassembly window contains storage being dumped, you can change locations by positioning the cursor to the desired area and typing over the contents; you can do this in either the hex or character area of the display.

If the Current Registers window is displayed, you can alter the PSW, general purpose registers, and floating point registers in the same way.

When a Disassembly window is open, you can modify the hex values of the instructions by typing over them. These changes are immediately reflected on the screen with different instruction mnemonics, addresses, and so on.

Intelligent cursor sensing

When the Current Registers window is open, you can display the area of storage addressed by a general purpose register by placing the cursor in the register and pressing the correct PF key. Similarly, you can display storage addressed by the instruction address of the PSW. If the cursor is in an access register, commands that can use an *access-list-entry token* (ALET) use the contents of the access register in addition to the contents of the associated general purpose register.

The same things can be done in the Old Registers window as in the Current Registers window.

When a Dump window is open or when the Disassembly window contains storage being dumped, a full word memory location may be used to specify the next area to be displayed by positioning the cursor in it and pressing a PF key. This is useful for following a chain of control blocks.

When a Disassembly window is open, you can set breakpoints by positioning the cursor on the desired instruction and pressing a PF key.

z/VM When a Dump window is open or when the Disassembly window contains storage being dumped, you can set PER address (storage alteration) stops by positioning the cursor on the desired location and issuing an ADSTOP command.

Screen swapping

IDF provides a screen swapping facility for debugging programs that perform full-screen I/O. When you use this facility, your program's screen is captured when a breakpoint is reached, and restored when control is returned to your program for more than a single instruction.

Record and playback

The IDF command record and playback facility makes recreating debugging sessions easier.

Customization - profile and macros

You can use a profile, written in the REXX programming language, to redefine display colors, PF key functions, initial display format, and many other options.

A subcommand environment supports IDF macros written in REXX. The ENTER key or any of the 24 PF keys can be assigned to run a macro or an IDF command.

User-written macros can:

- extract an argument based on cursor position
- set the cursor into a given window
- set data into either of IDF's message areas
- place data on the command line
- sound the terminal's audible alarm
- obtain the disassembled text associated with an instruction

Where can I run IDF?

IDF can be run on most ESA/370, or later, processors, from terminals (or emulators) of the 3270 family (or equivalents).

Environments supported

IDF runs on these operating systems, and unless otherwise stated, on subsequent versions, releases, and modification levels of these systems:

z/OS OS/390® Version 2 or z/OS Version 1

Required:

TSO/E Version 2 or higher
DFSMS/MVS™ 1.3 or higher

For more information, see Chapter 5, "Debugging programs on z/OS," on page 41.

CMS VM/ESA Release 2

For more information, see Chapter 6, "Debugging programs on CMS," on page 51.

z/VSE VSE/ESA Release 2.3

For more information, see Chapter 7, "Debugging programs on z/VSE," on page 59.

Limitations when debugging on TSO

The following limitations apply on TSO:

- IDF can only be used to debug programs that have their own (E)SPIE or (E)STAE exits if SVC 97 is used for breakpoints. The default is to use SVC 97 for breakpoints. You can override this by using the NOSVC97 option, or a SET OPTION ON NOSVC97 command in the PROFILE macro. For more details see "Breakpoint method selection (TSO)" on page 41.
- IDF cannot debug programs that issue ISPF SELECT service calls or those that involve z/OS multitasking *unless* SVC 97 is used for breakpoints. The default and override are as for the previous point.
- The REXX system interface requires that external functions return an EVALBLOK. If you are debugging a REXX function package that has been called by a REXX exec and you quit from IDF before the function has completed, the REXX interpreter issues an error message. You may need to logoff TSO and log back on.

Limitations when debugging on CMS

The following limitations apply on CMS:

- You must take special care when debugging programs which:
  - Employ interrupt-driven exit routines, such as ABNEXIT or HNDEXT exit routines
  - Reside above the address contained in the VMSIZE word of NUCON
  - Reside in DCSS, whether read only or read-write
- The REXX system interface requires that external functions return an EVALBLOK. If you are debugging a REXX function package that has been called by a REXX exec and you quit from IDF before the function has completed, the REXX interpreter issues an error message. You may need to re-IPL CMS.
- An IDF macro should invoke MRUN or MSTEP (or any other IDF command which causes immediate execution of the target program) using the LPSW Fastpath REXX addressing environment, which is the default addressing on entry to IDF macros. Detailed usage notes under "MRUN" on page 149 indicate the limitations of and effects of using Address ASM.
- If you use IDF's PER Y mode to debug your program, IDF uses the hardware Program Event Recording (PER) support for register and storage alteration events.

  PER events might be triggered during the execution of CMS's SVC handling (for example, for a register alteration). If the event occurs in the small window between the start of that CMS SVC handling and the point where it has saved the SVC interrupt status, there is a problem if IDF tries to issue an SVC itself (for example, to obtain working storage or to display a full-screen panel). CMS recognizes an illegal SVC recursions, issues a message, and requires a re-IPL.

  IDF uses special logic when running on CMS level 8 (or higher) to detect when CMS is in that small window of its SVC handler; if you are on CMS 7 (or lower), the IDF detection logic is bypassed. CMS may detect an *Illegal reentry into INTSVC*, and require a re-IPL of CMS.

  Because IDF cannot issue an SVC itself when in the *SVC window*, it cannot display a full-screen panel, or even issue a normal LINEDIT or LINEWRT or WRTERM (which each require an SVC). Instead, it generates a set of `CP MSG *` messages to the terminal to provide some basic information about the event that could not be handled normally. IDF execution continues normally after these messages.

Limitations when debugging under z/VSE

The following limitations apply on z/VSE:

- IDF only supports debugging of phases that are loaded into the same partition as IDF.
- Logical Transients or phases loaded into the SVA cannot be debugged
- The z/VSE Linkage Editor does not include any SYM records in the phase. Therefore only external symbols are available for IDF processing on z/VSE, unless you use the source level facilities provided by ASMLANGX, see "Batch language extraction on z/VSE" on page 18.

Chapter 2. Getting started with IDF

This chapter provides a description of the steps to debug the IDF sample program.

IDF needs some changes to your normal run-time environment to allow it to debug your programs. These changes ensure that IDF receives and retains control when your program runs.

Program preparation

Before invoking IDF, you need to make some information about your program available to IDF.

In any case where IDF's source-level debug capabilities are used, before a debugging session you must generate an IDF language extract file, using the ASMLANGX utility.

Detailed information about program preparation is provided in Chapter 3, “Using ASMLANGX to extract source-level information,” on page 15.

Program debug basics

Having prepared your program, you can use IDF to help you debug it.

A typical sequence of operations is:
- Start the debugging session by invoking ASMIDF (IDF), specifying your program name, IDF options (if appropriate), and parameters to pass to your program.
- Issue the LANGUAGE LOAD command to load the IDF Language extract files for all other program sections which are relevant to the problem being debugged.
- Issue IDF commands to tailor your debugging environment and to determine where your program is in error. You can issue commands as IDF macros, or interactively at the command prompt.

Commands to IDF come from:
- Entering them on the IDF command line.
- Pressing PF keys, which issues the associated IDF command.

  The IDF commands on PF keys are the default set of commands provided by IDF, or user commands, registered using the PFK command.
- IDF macros (written in REXX).
- User programs.

Refer to Chapter 8, “Windows, PF keys, cursor positioning, and other operational details,” on page 65 and Chapter 10, “Commands and operating procedures,” on page 91 for more details.

Sample debug session on TSO

This section provides a simple example of preparing and debugging an assembler language program. It assumes that both the High Level Assembler and the IDF components of the Toolkit were installed on your system and you have access to them.

Sample program preparation

1. Assemble and link the sample program:

```
//jobname JOB
//ASMMSAMP EXEC ASMACL,PARM.C=(OBJ,ADATA)
//C.SYSIN    DD DSN=<hlq>.SASMSAM2(ASMMSAMP),DISP=SHR
//C.SYSADATA DD DSN=<myid>.SYSADATA(ASMMSAMP),DISP=SHR
//L.SYSLMOD  DD DSN=<myid>.LOAD(ASMMSAMP),DISP=SHR
```

2. Create the source-level extract file:

```
//jobname JOB
//ASMMSAMP EXEC PGM=ASMLANGX,PARM='ASMMSAMP'
//SYSADATA DD DSN=<myid>.SYSADATA,DISP=SHR
//ASMLANGX DD DSN=<myid>.ASMLANGX,DISP=SHR
```

Invoking IDF

At the TSO/E READY prompt allocate the required data sets:

```
ALLOC FI (ASMLANGX) DS('<myid>.ASMLANGX') SHR
TSOLIB ACT DS('<myid>.LOAD')
```

Invoke IDF by entering:

```
  ASMIDF ASMMSAMP
```

Wait until the initial IDF panel is displayed, and then continue at “Running IDF on a sample program” on page 12.

Invoking IDF with a TSO batch job

To debug the sample program ASMMSAMP, submit the following job:

```
//jobname JOB
//IDFT     EXEC PGM=IKJEFT01,DYNAMNBR=100,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 TSOLIB ACT  DSN( '<myid>.LOAD'      +
                 '<hlq>.SASMMOD1'   +
                 '<hlq>.SASMMOD2' )
 ALLOCATE DD(ASMLANGX) DSN( '<myid>.ASMLANGX' ) SHR REUSE
 ALLOCATE DD(ASM)      DSN( '<myid>.ASM'      ) SHR REUSE
 TSOEXEC ASMIDF ASMMSAMP ( LUNAME <luid>
 TSOLIB DEACT
 FREE FI(ASM ASMLANGX)
/*
```

Notes:

1. The IDF parameter LUNAME (or LU) must be specified
2. The IDF VTAM APPLIDs ASMTL001 to ASMTL*nnn* must be ACTIVE
3. The VTAM `luid` specified in the LUNAME parameter must be ACTIVE

Sample debug session on z/OS

This section provides a simple example of debugging an assembler language program. It assumes that both the High Level Assembler and the IDF components of the Toolkit were installed on your system and you have access to them.

Invoking IDF with a batch job

To debug sample program ASMMSAMP, prepared in the section “Sample debug session on TSO” on page 9, submit the following job:

```
//jobname JOB
//IDFB    EXEC PGM=ASMIDFB,
//             PARM='ASMMSAMP ( NOSVC97 LUNAME <luid> '
//STEPLIB  DD DISP=SHR,DSN=<myid>.LOAD
//         DD DISP=SHR,DSN=<hlq>.SASMMOD1
//          DD DISP=SHR,DSN=<hlq>.SASMMOD2
//ASMLANGX DD DISP=SHR,DSN=<myid>.ASMLANGX
//ASM      DD DISP=SHR,DSN=<myid>.ASM
//SYSTSPRT DD SYSOUT=*
```

Notes:

1. The IDF parameters NOSVC97 and LUNAME must be specified
2. The IDF VTAM APPLIDs ASMTL001 to ASMTL*nnn* must be ACTIVE
3. The VTAM `luid` specified in the LUNAME parameter must be ACTIVE
4. TSO Command Processor programs are not supported

Sample debug session on CMS

This section provides a simple example of preparing and debugging an assembler language program. It assumes that both the High Level Assembler and the IDF components of the Toolkit were installed on your system and you have access to them.

Sample program preparation

1. Assemble the sample program:
   ```
   ASMAHL ASMMSAMP (ADATA
   ```
2. Generate the load module and rename the LOAD MAP file:
   ```
   LOAD ASMMSAMP (RLDSAVE
   GENMOD ASMMSAMP
   RENAME LOAD MAP A ASMMSAMP = =
   ```
3. Create the source-level extract file:
   ```
   ASMLANGX ASMMSAMP
   ```

Invoking IDF

Invoke IDF by entering:

```
ASMIDF ASMMSAMP
```

Wait until the initial IDF panel is displayed, and then continue at “Running IDF on a sample program” on page 12.

Sample debug session on z/VSE

This section provides a simple example of preparing and debugging an assembler language program. It assumes that both the High Level Assembler and the IDF components of the Toolkit were installed on your system and you have access to them.

Sample program preparation

```
* $$ JOB JNM=ASMMSAMP,CLASS=0,DISP=L,LDEST=*,PDEST=*
// JOB ASMMSAMP
// LIBDEF *,SEARCH=(PRD2.PROD,yourlib.sublib)
// DLBL SYSADAT,'SYSADATA',0,VSAM,                                X
              CAT=VSESPUC,RECSIZE=8192,                           X
              DISP=(,KEEP),RECORDS=(500,500)
// LIBDEF PHASE,CATALOG=yourlib.sublib
// OPTION NODECK,CATAL
   PHASE ASMMSAMP,*
// EXEC ASMA90,SIZE=ASMA90,PARM='ADATA'
 COPY ASMMSAMP
/*
// EXEC ASMLKEDT
/*
```

```
// EXEC ASMLANGX,PARM='ASMMSAMP'
/*
/&
* $$ EOJ
```

Invoking IDF

Invoke IDF by running:

```
* $$ JOB JNM=ASMMSAMP,CLASS=0,DISP=L,LDEST=*,PDEST=*
// JOB ASMMSAMP
// SETPFIX LIMIT=24K
// LIBDEF *,SEARCH=(PRD2.PROD,yourlib.sublib)
// EXEC ASMIDF,PARM='ASMMSAMP (LU luname'
/*
/&
* $$ EOJ
```

Wait until the initial IDF panel is displayed on *luname*, and then continue at "Running IDF on a sample program."

Running IDF on a sample program

1. Open the Current Registers window by pressing PF2.

   The Current Registers window shows the PSW, 16 general purpose registers and 4 floating point registers.
2. Open the Disassembly window by pressing PF9.

   The Disassembly window shows the sample program in object and disassembled forms.
3. Execute the first instruction by pressing PF1. This will cause IDF to automatically load the language extract file for source-level debugging.
4. Open windows to display source variables:

   ```
   open var vpacked
   open var vhalf
   open var vfull
   open var vchar
   ```

   Tip: Use PF12 to retrieve the previous command and then overtype the last few characters.
5. Press PF1 until the end of the program, observing the changing values in the Current Registers window, and values being set in storage as displayed in the LSM Variable Information windows.
6. Press PF3 twice to exit IDF.

Part 2. Guide to using IDF

Chapter 3. Using ASMLANGX to extract source-level information

Source-level debug support extends IDF by adding source code and variable display.

ASMLANGX collects information about the program sections in the program being debugged, from the SYSADAT(A) file created by the High Level Assembler.

Assembly requirements

The first step is to assemble the program.

The ADATA assembler option is *needed* to generate a file containing detailed information about the assembly.

Program build requirements

The next step is to build the executable module, as appropriate for the operating system environment in which you intend to run IDF.

Building a module on z/OS

IDF uses information from the Load Module file to determine locations of the program CSECTs and external symbols.

Building a module on CMS

The LOAD MAP file that results from the LOAD/GENMOD operation must be retained.

- Rename this file so that its name matches the file name of the executable module.
- This file is used by IDF to determine locations of the program CSECTs and external symbols.
- If the assembler TEST option is specified, this file also contains records that describe many of the assembler internal symbols. If present, IDF uses these records to help in the disassembly of the program. Since they are not essential, and tend to make the final MAP file quite large, you may suppress them by specifying the LOAD NOINV option.

Building a phase on z/VSE

IDF uses the librarian member `phasename.`MAP created by ASMLKEDT, which is a linkage editor *front-end* program.

- This member is used by IDF to determine locations of the program CSECTs and external symbols.
- The assembler TEST option does not apply.

If IDF cannot locate a MAP for a referenced phase then a dummy map is created. The dummy map has one CSECT with the same name as the phase and one entry point name **ENTRY**.

To produce a map for IDF On z/VSE, replace the program LNKEDT in the link-edit step with program ASMLKEDT. ASMLKEDT calls the linkage editor and produces a librarian member `phasename.`MAP on the PHASE catalog library.

To invoke ASMLKEDT, adjust the JCL, replacing

```
 // EXEC LNKEDT,PARM='...'
```

with
```
 // EXEC ASMLKEDT,PARM='...'.
```

The PARM values remain the same.

For example:
```
// LIBDEF  PHASE,CATALOG=MY.SUBLIB
// LIBDEF  *,SEARCH=(MY.SUBLIB,...)
// OPTION  CATAL
   PHASE   MYPROG,*
   INCLUDE MYPROG
/*
* Invoke linkage editor and create MYPROG.MAP
// EXEC ASMLKEDT,PARM='AMODE=24'
/*
```

Running ASMLANGX

The following sections describe how to run ASMLANGX to create a file to allow source-level debugging.

ASMLANGX handles the case where several programs are included in a single source file and the assembler BATCH option is used to process this composite program.

For all operating systems, the input file must be a SYSADATA file created by High Level Assembler Release 2 or higher.

Extraction file allocation on z/OS

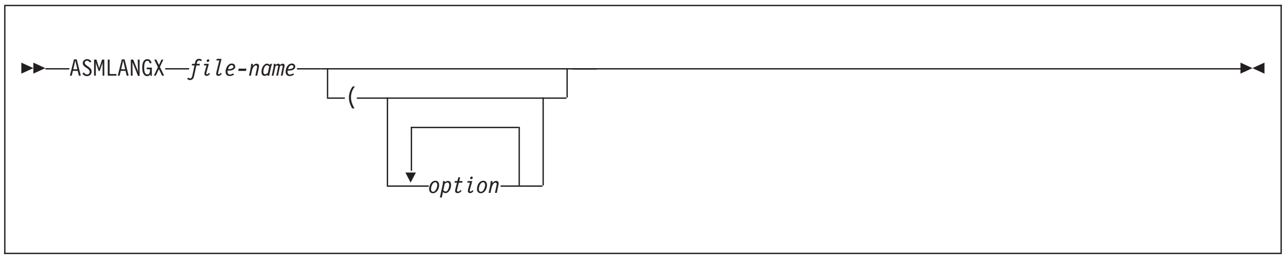
You may use ASMLANGX to create the extract files as sequential files on z/OS, but normally you create a PDS to contain the extracted data for actually debugging the target program using IDF's Language Support.

The recommended format of the extract file is:
```
RECFM(VB) LRECL(1562) BLKSIZE(27998)
```

Memory above the 16 MB line is exploited for extract information.

ASMLANGX does not perform any dynamic allocation. You will need to allocate the SYSADATA data set to DDname SYSADATA and the ASMLANGX data set to DDNAME ASMLANGX before you invoke the ASMLANGX command processor.

Online language extraction on TSO



file-name
: The PDS member name of the input and output files.

  For TSO sequential files (DSORG(PS)), the name is ignored.

  The default input file DDname is SYSADATA. Please see "PFT" on page 261.

The default output file DDname is ASMLANGX. Please see “OFT” on page 260.

option
: Options are described in Appendix A, “ASMLANGX options,” on page 257.

Example TSO commands

```
alloc dd(sysadata) ds(my.adata) shr
alloc dd(asmlangx) ds(my.langx) old
asmlangx myprog (asm loud error
```

Batch language extraction on z/OS

```
►►──//stepname──EXEC──PGM=ASMLANGX──,PARM='file-name(─┬────────────┬─'───────────────────►◄
                                                      │ ┌────────┐ │
                                                      └─▼─option─┴─┘
```

file-name
: The PDS member name of the input and output files.

For TSO sequential files (DSORG(PS)), the name is ignored.

The default input file DDname is SYSADATA. Please see “PFT” on page 261.

The default output file DDname is ASMLANGX. Please see “OFT” on page 260.

options
: Options are described in Appendix A, “ASMLANGX options,” on page 257.

Example JCL

```
//*                                                                   */
//*     --ASMLANGX -- EXTRACTION --
//*
//*     Replace member with the correct member name
//*     Replace hlq. with the correct high-level qualifiers
//*
//ASMLANGX EXEC PGM=ASMLANGX,REGION=4096K,
// PARM='member (ASM LOUD ERROR'
//SYSADATA DD  DISP=SHR,DSN=hlq..SYSADATA
//ASMLANGX DD  DISP=OLD,DSN=hlq..ASMLANGX
```

Online language extraction on CMS

```
►►──ASMLANGX──file-name──(─┬────────────┬──────────────────────────────────►◄
                           │ ┌────────┐ │
                           └─▼─option─┴─┘
```

file-name
: The file name (FN) of the input and output files.

The default input file type is SYSADATA. Please see “PFT” on page 261.

The default output file type is ASMLANGX. Please see “OFT” on page 260.

option
Options are described in Appendix A, “ASMLANGX options,” on page 257.

Batch language extraction on z/VSE

```
►►──//──EXEC──ASMLANGX──,PARM='output-file-name──┬──────────────────┬─'──────────────────────────►◄
                                                  └─(──┬──────────┬─┘
                                                       │ ┌──────┐ │
                                                       │ ▼      │ │
                                                       └──option┴─┘
```

output-file-name
The librarian member name of the ASMLANGX member that is either created or replaced on the specified PHASE CATALOG library.

The default input file DLBL name is SYSADAT. Please see “PFT” on page 261.

The default output file name can be overridden. Please see “OFN” on page 260.

option
Options are described in Appendix A, “ASMLANGX options,” on page 257.

JCL example

```
* Create language member ASMSAMP.ASMLANGX on MY.LIB
* for source level debug
// LIBDEF *,SEARCH ...
// LIBDEF *,CATALOG=MY.LIB  <======= Required
// DLBL SYSADAT,'my.adata',0,VSAM,CAT=VSESPUC,DISP=(,KEEP)
// EXEC ASMLANGX,PARM='myprog (ASM LOUD ERROR'
/*
```

Which files to keep

Once you have generated an extraction file, you can choose to discard or keep the input files. IDF needs only the extraction file and the module file, and the LOAD MAP on CMS.

z/VM and z/OS
If you have several separately compiled programs which are linked into a single module, you can either retain an extraction file for each of them, or you can copy them into a single appended file for convenience. It is recommended you keep them separate, in case you want to change and recompile one of the separate compiles.

Return codes

0	Operation successful, output file was written.	
0xxx	Error discovered while parsing arguments or options, values for *xxx* are:	
	1	Token too long
	2	Left parenthesis found inside options
	3	Unknown option
	4	No primary input file name (PDS member name) was specified
2xxx	Error occurred during scan of ADATA file.	
3xxx	Error occurred while writing output file.	

For return codes 2xxx, and 3xxx, the values for *xxx* are:

0yy	yy is the RC from File_Write
1yy	yy is the RC from File_Open

2yy	yy is the RC from File_Read
3yy	yy is one of the following ASMLANGX codes: 10 = file not in expected format 11 = maximum mumber of statements exceeded
4yy	yy is the RC from File_Point
5yy	yy is the RC from Mem_Allocate
6yy	yy is the RC from Mem_Free
7yy	yy is the RC from File_Close
8yy	yy is the RC from File_Note

Chapter 4. Invoking IDF to debug your program

This chapter describes how to invoke IDF to debug your program.

When preparing to use IDF, you should understand:
- How the program is packaged
- How the program is to be loaded into storage
- The conventions used to pass parameters to the program

This section explains how to invoke IDF to perform different kinds of debugging jobs.

For a complete list of options, see “IDF options at invocation” on page 25.

When you invoke IDF, double-check that your options are correct. If you make mistakes setting up a debugging session, you are making the session more complicated.

Running IDF on TSO and CMS

```
►►──ASMIDF──module-name──┬──────────────────────────────────────────────────────────────┬──────►◄
                         │      ┌──────────────────────────┐                            │
                         │      ▼                          │                            │
                         └─(────┬─idf-option───────────┬───┴──┬────────────────────────┬┘
                                └─*profile-macro-option┘      └─/──module-parameters───┘
```

module-name
: The name of the module to be debugged.

idf-option
: An option directed to IDF.

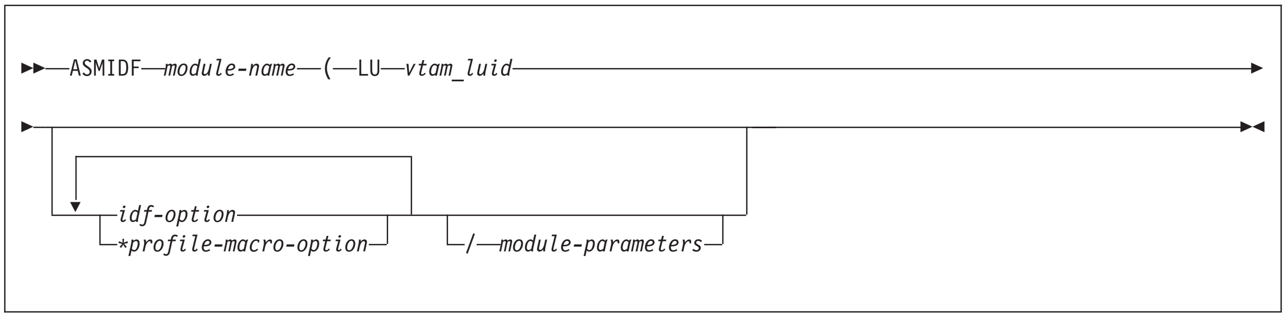
  See “IDF options at invocation” on page 25 for details on the options.

**profile-macro-option*
: An option that begins with an asterisk (*) is not examined by IDF. Instead, it is made available for processing by the PROFILE macro. Retrieve these options by issuing the EXTRACT PLIST command within the macro and parsing the result.

module-parameters
: Parameters directed to the module that is to be debugged.

IDF only operates as a TSO command processor on a 3270 terminal, either real or emulated.

Running IDF via TSO batch job



module-name
The name of the module to be debugged.

vtam_luid
The VTAM logical unit name of the terminal to be used by IDF.

idf-option
An option directed to IDF.

See "IDF options at invocation" on page 25 for details on the options.

The LUNAME option is required.

**profile-macro-option*
An option that begins with an asterisk (*) is not examined by IDF. Instead, it is made available for processing by the PROFILE macro. Retrieve these options by issuing the EXTRACT PLIST command within the macro and parsing the result.

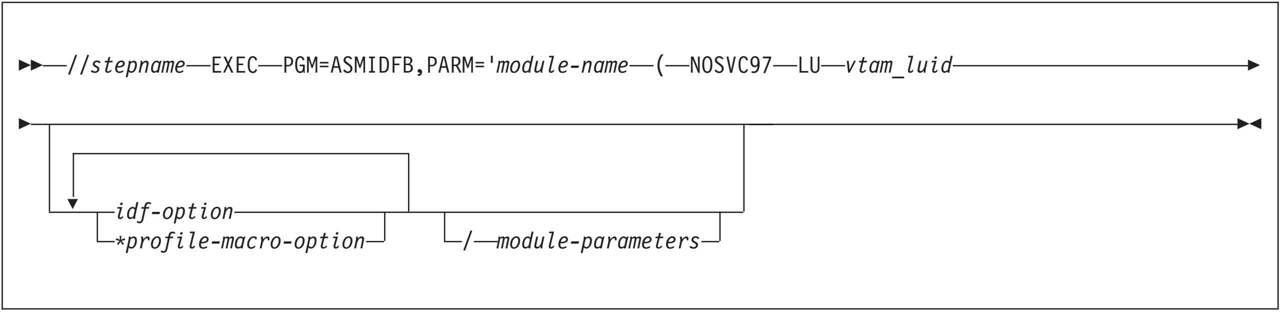
module-parameters
Parameters directed to the module that is to be debugged.

Example JCL

Note: This example assumes the HLASM Toolkit target library (SASMMOD2) is in the linklist.

```
//jobname JOB <job parameters>
//IDFT     EXEC PGM=IKJEFT01,DYNAMNBR=100,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 TSOLIB ACT  DSN( '<myid>.LOAD' )
 ALLOCATE DD(ASMLANGX) DSN( '<myid>.ASMLANGX' ) SHR REUSE
 ALLOCATE DD(ASM)      DSN( '<myid>.ASM'      ) SHR REUSE
 TSOEXEC ASMIDF MYPROG ( LU MYterm / module-parameters
 TSOLIB DEACT
 FREE FI(ASM ASMLANGX)
/*
```

Running IDF via z/OS batch job



stepname
: The name of the job step.

module-name
: The name of the module to be debugged.

vtam_luid
: The VTAM logical unit name of the terminal to be used by IDF.

idf-option
: An option directed to IDF.

  See “IDF options at invocation” on page 25 for details on the options.

  The NOSVC97 option and the LUNAME option are required.

**profile-macro-option*
: An option that begins with an asterisk (*) is not examined by IDF. Instead, it is made available for processing by the PROFILE macro. Retrieve these options by issuing the EXTRACT PLIST command within the macro and parsing the result.

module-parameters
: Parameters directed to the module that is to be debugged.

Example JCL

```
//jobname JOB <job parameters>
//IDFB    EXEC PGM=ASMIDFB,
//             PARM='MYPROG ( NOSVC97 LUNAME MYterm '
//STEPLIB  DD DISP=SHR,DSN=<myid>.LOAD
//         DD DISP=SHR,DSN=ASM.SASMMOD1    HLASM TARGET LIBRARY
//         DD DISP=SHR,DSN=ASM.SASMMOD2    TOOLKIT TARGET LIBRARY
//ASMLANGX DD DISP=SHR,DSN=<myid>.ASMLANGX
//ASM      DD DISP=SHR,DSN=<myid>.ASM
//SYSTSPRT DD SYSOUT=C
```

Running IDF on z/VSE

```
                                   ┌──────────────────────────┐
                                   ▼                          │
►►──//──EXEC──ASMIDF,PARM='phasename──(──┬─idf-option──────────┬─┴─┬─────────────────────┬──────►
                                         └─*profile-macro-option┘   └─/──phase-parameters─┘

►─'──────────────────────────────────────────────────────────────────────────────────────────►◄
```

phasename
: The name of the phase to be debugged.

idf-option
: An option directed to IDF.

  See "IDF options at invocation" on page 25 for details on the options.

  The LUNAME option is required.

**profile-macro-option*
: An option that begins with an asterisk (*) is not examined by IDF. Instead, it is made available for processing by the PROFILE macro. Retrieve these options by issuing the EXTRACT PLIST command within the macro and parsing the result.

phase-parameters
: Parameters directed to the phase that is to be debugged.

Example JCL

```
// SETPFIX LIMIT=24K
// LIBDEF PHASE,SEARCH=(MY.LIBRARY,HLASM.LIBRARY)
// LIBDEF PROC,SEARCH=(MY.PROCLIB,HLASM.LIBRARY)
// EXEC ASMIDF,PARM='MYPROG (LU MYterm /phase-parameters'
/*
```

These JCL statements are needed:

SETPFIX LIMIT
: Sets the page fix limit. This must be set to 24K for the product exit to function correctly.

LIBDEF PHASE
: Defines the library that contains the phase map created with the ASMLKEDT, or the dummy map, and the phase to be debugged.

LIBDEF PROC
: Defines the procedure library which contains any REXX macros.

EXEC ASMIDF
: Executes the program ASMIDF (IDF) using the parameters specified with the PARM option.

IDF options at invocation

You can set most of these options after invocation using the SET OPTION command (see“SET OPTION” on page 178). However some options only make sense if they are specified at invocation.

The options only available at invocation are:

FASTPATH ISA   LIBE LINE   LUNAME NOPROFIL   PROFILE RLOG

When you set options in a profile, other restrictions apply. For more information, see “Command restrictions related to PROFILE execution” on page 204.

1ADSTOP (CMS only)

```
►►──1ADStop──────────────────────────────────────────────────────────────►◄
```

When PER is enabled, four address ranges are provided and storage modifications in any of these ranges should cause IDF to present a message. Because of the way the PER hardware works, IDF must partially disassemble the instruction being executed to obtain an address to check against the address ranges. In some cases storage modification events are missed.

However, if the 1ADSTOP option is set, the four address ranges are treated as a single address range, beginning at the lowest address in any range and ending at the highest address in any range. In this case events are recognized without partial disassembly.

AMODE24 | AMODE31 | AMODE64 (z/OS only)

```
   ┌─AMODE24─┐
►►─┼─────────┼───────────────────────────────────────────────────────────►◄
   ├─AMODE31─┤
   └─AMODE64─┘
```

AMODE24 is valid on any XA or ESA system. It initializes the target program as AMODE24, regardless of the CMS, z/OS, or z/VSE defaults.

AMODE31 is also valid on any XA or ESA system. It initializes the target program as AMODE31, regardless of the CMS, z/OS, or z/VSE defaults.

AMODE64 is only valid on a z/OS system running on a z/Architecture® machine. It initializes the target program as AMODE64, regardless of the z/OS default.

Note: On CMS, if the target program is *not* a nucleus extension, and is AMODE(ANY), this option is needed to enable IDF to recognize the program as AMODE31.

ASCII

```
►►──ASCii──────────────────────────────────────────────────────────►◄
```

ASCII tells IDF to display the character portion of dumped storage in ASCII rather than in EBCDIC. Data overtyped in these areas is also in ASCII, rather than EBCDIC.

AUTOLOAD | NOAUTOLD

```
      ┌─AUTOLoad─┐
►►────┤          ├─────────────────────────────────────────────────►◄
      └─NOAUTOLd─┘
```

AUTOLOAD tells IDF to issue LANGUAGE LOAD commands for you if you STMTSTEP to a location that is not within a code section for which IDF Language extract data was loaded. This is done in an attempt to automatically load IDF Language extract files for you. **This only works if the code section name matches the file name of the extract file**.

NOAUTOLD tells IDF not to issue LANGUAGE LOAD commands for you if you STMTSTEP to a location that is not within a code section for which IDF Language extract data was loaded. For more information about the LANGUAGE LOAD command, see “LANGUAGE LOAD” on page 129.

AUTOSIZE | NOAUTOSZ

```
      ┌─AUTOSize─┐
►►────┤          ├─────────────────────────────────────────────────►◄
      └─NOAUTOSz─┘
```

AUTOSIZE tells IDF to automatically size Disassembly windows, Dump windows, and LSM Information windows when a window is opened or closed. IDF also automatically places all opened windows so that each window's upper border overlays the lower border of the window last opened.

NOAUTOSZ tells IDF to not automatically size Disassembly windows, Dump windows, and LSM Information windows when a window is opened or closed. When a window is opened, IDF places it at the top of the screen.

BCX | NOBCX

```
   ┌─BCX───┐
►►─┤       ├─────────────────────────────────────────────────►◄
   └─NOBcx─┘
```

When the instruction displayed on the main panel is a BC or BCR, BCX shows an extended branch mnemonic instead of the CC mask when disassembling the instruction. Mnemonics used are those for use after compare instructions. This option has no effect on branch relative instructions.

NOBCX does not show BC and BCR instructions as their extended mnemonics, but shows the CC mask instead.

CKSUBCM

```
►►──CKSubcm──────────────────────────────────────────────────►◄
```

When the CKSUBCM option is set, IDF checks that its subcommand environment is intact before executing a REXX exit routine. The CKSUBCM option is intended for use in hostile environments which cut off the subcommand chain.

CMDLOG

```
►►──CMDLog───────────────────────────────────────────────────►◄
```

When the CMDLOG option is set, IDF logs each operator command in the command log. For the location of the command log, and more information about logging, see "Command record and playback features" on page 80.

CMPEXIT

```
►►──CMPExit──────────────────────────────────────────────────►◄
```

CMPEXIT indicates that the exit routine defined by the SET EXITEXEC command is a compiled-code routine, not a REXX exit routine.

COLORS | COLOURS

```
►►──┬─COLors──┬──mhti──────────────────────────────────────────────────────►◄
    └─COLours─┘
```

COLORS sets the display colors. You can specify up to four colors.

```
                                        (Default)
 m = color for messages                  white
 h = color for headings                  turquoise
 t = color for text                      blue
 i = color for input data                green
```

The first is always messages, the second is always headings, and so on. Valid color letters are:

```
    B = Blue            G = Green           P = Pink            R = Red
    T = Turquoise       W = White           Y = Yellow
```

For example, COLORS RYGB specifies messages red, headings yellow, text green, and input blue.

COMMAND

```
►►──COMmand──module-parameters─────────────────────────────────────────────►◄
```

Use the *module-parameters* as a command to start the debugging operation. If the command is an EXEC, you must prefix the command name with "EXEC".

DMSO (CMS only)

```
►►──DMSO───────────────────────────────────────────────────────────────────►◄
```

Most symbols that start with "DMS0" are generated by system macros and are a nuisance. By default, they are ignored. When the DMS0 option is set, they are loaded.

Note: Even if DMS0 is not set, "DMS0" symbols are loaded if they occur in DSECTs.

EXITEXEC

```
►►──EXITEXEC──name─────────────────────────────────────────────────────────►◄
```

EXITEXEC supplies the name of the exit routine.

FASTPATH | PATH | PATHFILE

```
►►──┬─FASTPath─┬───────────────────────────────────────────────────────►◄
    ├─PATH─────┤
    └─PATHFile─┘
```

FASTPATH displays the number of times each instruction was executed, at the right side of the disassembly listing. Equivalent to the PATH option in function, but provides enhanced performance.

PATH produces the same display.

PATHFILE produces the same display as PATH, but at the end of operation, writes the collected data to:
- On CMS, file "ASM PATHDATA *fm*", where *fm* is specified by the MODE option (and defaults to file mode A1).
- On z/OS, the data set pointed to by the PATHDATA DD name.
- On z/VSE, to SYSLST.

See "The PATH, FASTPATH, and PATHFILE options" on page 38 for more information.

FULLQUAL

```
►►──FULLQual──────────────────────────────────────────────────────────►◄
```

FULLQUAL always displays and returns symbolic addresses with the module name included. The default is to include only the module name, if the module containing the address is not the currently qualified module. See "Addresses displayed by IDF" on page 82 for more information.

HEXDISP

```
►►──HEXDisp───────────────────────────────────────────────────────────►◄
```

When the HEXDISP option is set, hexadecimal notation is used in disassembled instructions for all displacements from symbols, or displacements which cannot be related to symbols.

HEXINPUT

```
►►──HEXInput──────────────────────────────────────────────────────────►◄
```

When the HEXINPUT option is set, IDF assumes that numbers input without any explicit indication of base are hexadecimal. If the first digit of a hexadecimal constant is not zero, IDF interprets it as the name of a symbol.

IMPMACRO | NOIMPMAC

```
     ┌─IMPMacro─┐
►►───┤          ├──────────────────────────────────────────────────────────►◄
     └─NOIMPMac─┘
```

IMPMACRO permits implied macros to be executed from the command line.

NOIMPMAC disallows the implied execution of macros from the command line. With this option in effect, macro invocations must be prefixed by the MACRO keyword if they are to be found and executed.

INVPSW | NOINVPSW

```
     ┌─NOINVPsw─┐
►►───┴─INVPsw───┴──────────────────────────────────────────────────────────►◄
```

When the INVPSW option is set, it lets you set the PSW to any combination of hexadecimal digits through the SET PSW command.

When the NOINVPSW option is set, the PSW may only be set to a valid PSW through the SET PSW command.

ISA (CMS only)

```
►►──ISA──address───────────────────────────────────────────────────────────►◄
```

ISA defines the address of a 16-byte doubleword-aligned interrupt save area that is to be used by all of IDF's first level interrupt handlers.

Valid forms for *address* are X'nn', F'nn', and nn. The address must reside in the first 4k of storage. The default value is X'500.'.

LIBE (z/OS and CMS)

```
►►──LIBE──fn───────────────────────────────────────────────────────────────►◄
```

- On z/OS, the LIBE option specifies that the target load module should be loaded from an alternate DD name. A "$" indicates that the target should be loaded from the standard OS load module search order.
- On CMS, the LIBE option specifies that the target program is to be loaded from an OS-style LOADLIB rather than from a CMS-style MODULE file. The *fn* provided is "$", which indicates a CMS GLOBAL command has specified the necessary LOADLIB, or the file name of the LOADLIB, which is added to the list of GLOBAL LOADLIBs as the first library.

LINE (CMS only)

```
►►──LINE──line-address──────────────────────────────────────────────►◄
```

The LINE option provides a means of telling IDF to use a terminal other than the virtual console, and is useful when debugging full-screen applications. Specify the line address in explicit hexadecimal or explicit decimal notation.

LUNAME (z/VSE and z/OS)

```
►►──LUname──lu-unit─────────────────────────────────────────────────►◄
```

The LUNAME option defines the *lu-unit*, the VTAM logical unit name of the terminal to be used by IDF. This is *required* to run IDF in z/VSE.

LSMDEBUG

```
►►──LSMDebug────────────────────────────────────────────────────────►◄
```

Not for general use.

LSMDEBUG enables the internal diagnostic trace of the IDF Language Support subsystem.

MACROLOG

```
►►──MACROLog────────────────────────────────────────────────────────►◄
```

When the MACROLOG option is set, all IDF commands that are issued by macros or exit routines are written to:
- On CMS, "ASM MACROLOG *fm*", where *fm* is specified by the MODE option (and defaults to file mode A1).

- On z/OS, the commands are written to the data set defined by the MACROLOG DD name.
- On z/VSE, the commands are written to the data set defined by the MACROLG DLBL.

MODE (CMS only)

```
►►──MODE──file-mode──────────────────────────────────────────────────────►◄
```

When the MODE option is set, the "CMDLOG" and "PATHDATA" files are read from or written to the minidisk at the specified *file-mode*. The "MACROLOG" file is written to the minidisk at the specified *file-mode*.

MODMAP | NOMODMAP (CMS only)

```
     ┌─MODMap───┐
►►───┤          ├─────────────────────────────────────────────────────────►◄
     └─NOMODMap─┘
```

When the MODMAP option is set, IDF prefers the "fn MAP *" file over a "LOAD MAP *" file for symbol information. If no "fn MAP *" is found, IDF checks for a "LOAD MAP *" file.

When the NOMODMAP option is set, IDF prefers the "LOAD MAP *" file over a "fn MAP *" file for symbol information. If no "LOAD MAP *" is found, IDF checks for a "fn MAP *" file.

NODSECTS

```
►►──NODSects──────────────────────────────────────────────────────────────►◄
```

The default is to load symbols that occur in DSECTs. The NODSECTS option ignores these symbols.

NUCEXT (CMS only)

```
►►──NUCext────────────────────────────────────────────────────────────────►◄
```

When the NUCEXT option is set, the program is supposed to run as a CMS nucleus extension and is already loaded.

OFFSET

```
►►──OFFSet──────────────────────────────────────────────────────────────────►◄
```

When the OFFSET option is set, IDF shows dump and disasm addresses in terms of the currently set offset value. Data at the address specified by the current offset is shown as +00000000.

OLDBREAK

```
►►──OLDBREAK────────────────────────────────────────────────────────────────►◄
```

The OLDBREAK option sets an alternative mode of operation for the BREAK command. This mode of operation does not "toggle" a breakpoint. If the BREAK command is issued against an address where a breakpoint is already set, IDF issues an error message.

PASSPGM

```
►►──PASspgm─────────────────────────────────────────────────────────────────►◄
```

The PASSPGM option makes IDF pass program interruptions that are not the result of a PER event to the CMS, z/OS, or z/VSE program interrupt handler. Used for debugging exit routines.

Warning: If the target program has not activated an exit routine to trap the interrupt, IDF abends. You may need to re-IPL CMS or logon to TSO again following the debugging session. On z/VSE, `PASSPGM` occurs immediately on STXIT PC entry. If the STXIT is not present, then IDF traps the program check.

PROFILE | NOPROFIL

```
►►──┬─PROfile──┬───────────────┬─┬──────────────────────────────────────────►◄
    │          └─profile-macro─┘ │
    └─NOPROfil───────────────────┘
```

PROFILE specifies the name of the initial profile macro to be run.

NOPROFIL stops IDF executing an initial profile macro.

If neither PROFILE nor NOPROFIL are specified, IDF attempts to run an initial profile macro called PROFILE. If PROFILE is not found, then no initial profile macro is run.

QWDUMP

```
►►──QWDump──────────────────────────────────────────────────────────►◄
```

QWDUMP applies to unformatted dump displays only; it forces all displays to begin on a "quadword" boundary (the low order hex digit is zero).

RISK

```
►►──RISk────────────────────────────────────────────────────────────►◄
```

When the RISK option is set, IDF ignores as many error indications as possible. This may result in an IDF crash.

Setting the option may allow tracing of CMS services or DCSS-resident code under some circumstances.

RLOG

```
►►──RLog────────────────────────────────────────────────────────────►◄
```

When the RLOG option is set, as soon as the PROFILE macro has completed and the target program is ready for execution, all the commands in the command log file are executed. Then the CMDLOG option is set. This provides a means of resuming an interrupted earlier debugging session.

For more information about command logging, see “Command record and playback features” on page 80.

ROWSTYLE

```
►►──ROWstyle────────────────────────────────────────────────────────►◄
```

When the ROWSTYLE option is set, IDF uses the "traditional" row-style arrangement for the register display. The default is to display in columns.

SBORDER

```
►►──SBORDer──────────────────────────────────────────────────────────────►◄
```

When the SBORDER option is set, IDF does not use APL characters in window borders. This is useful when running IDF on a terminal emulator that does not support APL characters.

SCDACTIV

```
►►──SCDACTIV─────────────────────────────────────────────────────────────►◄
```

When the SCDACTIV option is set, IDF collapses its subcommand environment before activating the target program, and re-creates it when control returns to IDF. This option is not expected to be useful in general.

SELFNUCX (CMS only)

```
►►──SELFNucx──symbol─────────────────────────────────────────────────────►◄
```

The code is self-nucxloading, so IDF does not check the length shown in the SCBLOCK to see that it matches the length in the MODULE.

symbol defines the start of the nucxloaded code.

For example: `selfnucx freego`

STOPNOP | NOSTOPNP

```
►►──┬─STOPNOP──┬─────────────────────────────────────────────────────────►◄
    └─NOSTOPNp─┘
```

STOPNOP tells IDF to stop on NOP and NOPR instructions that follow BAL, BALR, BAS, and BASR instructions.

NOSTOPNP prevents IDF from stopping on NOP and NOPR instructions that follow BAL, BALR, BAS, and BASR instructions.

STOPSTMT | NOSTOPST

```
   ┌STOPSTmt─┐
►►─┤         ├──────────────────────────────────────────────────────────►◄
   └NOSTOPSt─┘
```

STOPSTMT tells IDF to stop single-stepping for a STMTSTEP or STEP command if it reaches a location that is within a code section for which IDF Language extract data was not loaded. This is done in an attempt to avoid what otherwise appears to be a loop.

NOSTOPST tells IDF to continue single-stepping for a STMTSTEP or STEP command if it reaches a location that is not within a code section for which IDF Language extract data was loaded. For more details see "Controlling single-stepping your program" on page 86

SVC97 | NOSVC97 (z/OS only)

```
   ┌SVC97───┐
►►─┤        ├───────────────────────────────────────────────────────────►◄
   └NOSVC97─┘
```

When the SVC97 option is set, IDF uses SVC 97 for breakpoints.

When the NOSVC97 option is set, IDF does not use SVC 97 for breakpoints. Instead, it uses invalid opcodes. NOSVC97 *must* be specified if running IDF in a z/OS (non-TSO) batch environment.

The SVC97 and NOSVC97 options can be set only at invocation, or within the profile macro prior to the load of the user application.

For more details see section "Breakpoint method selection (TSO)" on page 41.

SWAP

```
►►──SWAp────────────────────────────────────────────────────────────────►◄
```

SWAP tells IDF to capture the target's screen. You need this option when you are debugging full-screen applications, and need to enter information on the screen. You switch between the IDF debugging screen and the target application through the SWAP command.

SYSTEM (CMS only)

```
►►──SYStem──────────────────────────────────────────────────────────────►◄
```

When the SYSTEM option is set, the program runs in system key (key=0).

TRACEALL

```
►►──TRACeall────────────────────────────────────────────────────────────►◄
```

When the TRACEALL option is set, and single-step mode is on, IDF traces executed instructions as much as possible.

This *does not* include tracing in CMS services (though the RISK option may provide access) or other code that resides above the address in the VMSIZE word of NUCON. It *does* include tracing in nucleus extensions called through BALR linkage if they fall below the VMSIZE limit.

The default is to limit tracing to the target program's defined limits.

This option is set off through the SET OPTION command.

TRANS (CMS only)

```
►►──TRANs───────────────────────────────────────────────────────────────►◄
```

When the TRANS option is set, the program runs as a transient.

UNFTDUMP

```
►►──UNFtdump────────────────────────────────────────────────────────────►◄
```

When the UNFTDUMP option is set, the memory dump does not show symbols. The formatted dump display may still be obtained with the DUMPMODE command.

Initialization of general-purpose registers (GPRs)

The GPRs are initialized by IDF to the value X'FEFE*nn*0F' (where *nn* is the register number, 00 through 0F), with the following exceptions:

- R12 and R15 contain the target program's entrypoint address.
- R13 points to a 24-word save area (doubleword aligned).

z/OS

- R1 contains the parameter pointer.

  If the COMMAND option is *not* specified, R1 points to a standard z/OS CALL format parameter list.

  If the COMMAND option is specified, R1 points to a standard TSO Command Processor Parameter List (CPPL). For more details, see “Loading programs (TSO)” on page 44.
- If SVC 97 is being used for breakpoints, R14 points to an SVC 97; otherwise, it points to an X'02FF'. IDF uses either of these to determine if control was returned by the target program.
- If option AMODE64 is in effect, the first word of each register is initialized by IDF to X'00000000'.

z/VM

- R0 and R1 contain parameter pointers in the usual CMS fashion (for more information, see “How to specify parameters for your program” on page 51).
- If the program is a nucleus extension, R2 contains the address of its SCBLOCK.
- R14 points to an X'02FF' used by IDF to determine that the target program has returned control.

z/VSE

- R1 contains the parameter pointer.
- R14 points to an X'01FF' used by IDF to determine if control was returned by the target program.

Initialization of floating point registers

The FPRs are initialized by IDF to X'0000000000000000'.

Initialization of access registers

Access registers AR0 and AR2 to AR15 are initialized by IDF to X'00000000'. AR1 is initialized by IDF to X'00000001'.

The PATH, FASTPATH, and PATHFILE options

Use these options to display how many times an executed instruction was executed. From the display you can determine:

- Sections of code that were not executed

  These sections are either sections not exercised by testing, or "dead code" that will never be executed, and hence you can remove from the program.
- Sections of code that were executed many times

  These are called "hotspots" in your program: sections where by refining the program algorithm or implementation, you can speed up program execution.

For example, to execute the CMS program "MYPROG" with the PATH support enabled, issue the command:

```
ASMIDF MYPROG (PATH /Parms for MYPROG
```

If you specified the PATH or PATHFILE options when IDF was invoked, you can choose an alternative algorithm for collecting the instruction execution count information. You can set the FASTPATH option only after the target program is loaded, and before any execution counts are collected.

Using the PATH option

If you specify the PATH option, the number of times each instruction was executed is displayed in a column on the right of the open Disassembly windows. You must also have the disassembled text displayed (SHOW DISASM), as the numbers are displayed to the right of these disassembled lines. The same applies if the HISTORY command is used with the PATH option.

If the count exceeds 7 digits, the leftmost digits are truncated. This is unlikely, as executing individual instructions 10 million times through IDF imposes quite a load on the system.

Using the PATHFILE option

If you want to capture the PATH information for more detailed analysis, you can specify the PATHFILE option rather than PATH. When you leave IDF, this information is saved in a file:

- On CMS, the file "ASM PATHDATA A1" is written to disk.
- On z/OS, data is written to the data set pointed to by the "PATHDATA" DD statement.
- On z/VSE, data is written to SYSLST.

The columns in this file are, from left to right:

1. Execution count
2. Absolute memory address of the instruction
3. Name of the module containing the address
4. Address of the instruction within the module (offset)
5. Symbolic name of the instruction

If you plan to use this option, make sure you have plenty of free disk space. Also be aware that each instruction needs eight bytes of free storage, plus 20 bytes of control information for every 64K block of information.

Using the FASTPATH option

You can also consider the FASTPATH option, which records execution counts in a different way. Where the PATH option uses eight bytes of storage for every instruction executed, FASTPATH uses only four, but allocates a contiguous block of storage to record information by assuming that the maximum number of instructions in your program is the program size divided by 2. Thus the FASTPATH option may, or may not, take more storage than the PATH option. However, it does not have to scan a list to find the instruction count information, so it executes in *much* less time.

The FASTPATH option is really a "mode" type of setting. The type of storage allocation and recording performed is determined by whether you issue a SET OPTION ON FASTPATH or a SET OPTION ON PATH command first. Once you have set the FASTPATH option to ON, you can use SET OPTION ON/OFF PATH to turn execution recording on and off; the allocation method does not change once it is initially set. Because the allocation mechanism assumes a given program size and start address, the SET BASE and SET SIZE commands may not be used after the FASTPATH option is set.

When using the FASTPATH option, instruction count information is only collected for the original target module. All other instructions executed are ignored.

Excluding called subroutines

If you have subroutines in your program that you do not want to be included in PATH or FASTPATH processing, use the SKIPSTEP command. This causes IDF to skip PATH and FASTPATH processing when you call a subroutine on the list of subroutines being skipped. That is, for the purposes of PATH and FASTPATH processing, the skipped subroutine is treated as one instruction: the subroutine call itself.

Chapter 5. Debugging programs on z/OS

On z/OS, IDF supports the following:
- User-Area Programs
- Programs in the z/OS Link List

This chapter describes how to debug each of these.

IDF supports the debugging of programs with these formats:
- Load Module members
- Programs in storage
- Program Objects

Note: IDF is not able to debug CICS® or IMS™ transactions directly.

Data set naming conventions

Some IDF commands cause data to be written to or read from files. Since IDF was originally written to run on CMS, the commands are oriented towards the naming conventions used by the CMS file system.

The mapping of the CMS file conventions to z/OS is:

CMS	**Equivalent on z/OS**
fn	PDS member name (ignored if using sequential file)
ft	DDNAME, which in turn points to the z/OS data set name
fm	Not used on z/OS

Note: You must allocate the DDs using TSO ALLOC commands, or place them in your TSO startup procedure JCL, or place them in the batch IDF JCL. No dynamic allocation is performed.

Optional data set file allocations

The following allocations are required if the relevant options are specified at IDF invocation:

```
CMDLOG    RECFM(VB) LRECL(124) BLKSIZE(29393)
MACROLOG  RECFM(VB) LRECL(6148) BLKSIZE(29393)
PATHDATA  RECFM(VB) LRECL(79) BLKSIZE(29393)
```

Breakpoint method selection (TSO)

On TSO, IDF has two methods of setting breakpoints:
- Using SVC 97, the TSO/E TEST breakpoint SVC
- Using invalid opcodes.

Both methods have advantages and disadvantages.

SVC 97 breakpoints

The advantage of using SVC 97 for breakpoints is that IDF can then debug code that uses z/OS subtasking or sets up (E)STAE or (E)SPIE exits. Also, IDF can debug ISPF dialogs.

The disadvantage is that you must install IDF in the z/OS Link List or your STEPLIB, or via the TSOLIB command. You cannot install it in a TASKLIB or ISPLLIB concatenation. The reason for this is that IDF must be invoked directly from TSO/E READY or from a CLIST that it is invoked from TSO/E READY. You *cannot* invoke IDF from a REXX EXEC or under ISPF except by using the TSOEXEC command. For an example of how to debug ISPF dialogs, see "ISPF applications (TSO)" on page 47.

Invalid opcode breakpoints

The advantages of not using SVC 97 for breakpoints are:

1. You can install IDF in a TASKLIB or an ISPLLIB.
2. You can more easily invoke IDF under ISPF.

This is useful if your LOGON PROC does not have a STEPLIB that you can write to.

The disadvantages of not using SVC 97 for breakpoints are:

1. IDF cannot debug code that uses z/OS subtasking or sets up (E)STAE or (E)SPIE exits. (When IDF is invoked with the NOSVC97 option to test a program running under the LE/370 environment, the LE/370 NOSTAE and NOSPIE options must be used.)
2. IDF cannot debug ISPF dialogs.
3. DBREAK does not function for modules not in storage.

Specifying the breakpoint method

Options allow you to select the method used for setting breakpoints:

SVC97
Tells IDF to use SVC 97 for breakpoints. This is the default.

NOSVC97
Tells IDF not to use SVC 97 for breakpoints.

Specify an option at invocation, or with a SET command in the PROFILE macro.

The SVC97 and NOSVC97 options cannot be set after IDF has loaded your program. As a result, if the profile macro issues a LOAD or a LOAD MODULE command and you do not wish to use the default breakpoint method, you must set the NOSVC97 option before issuing the LOAD or LOAD MODULE command.

Specifying the SVC97 or NOSVC97 option at invocation overrides the setting of NOSVC97 or SVC97 in the PROFILE macro.

Breakpoint method (z/OS batch)

In z/OS batch, IDF has only one method of setting breakpoints:

- Using invalid opcodes

How to specify parameters for your program (TSO)

To pass parameters to your program, append them to the ASMIDF command after a slash (/). IDF interprets anything that follows a slash as parameters that should be passed to the target program.

The parameter string is passed to the target in standard z/OS `EXEC PGM` style. That is, R1 points to a word that points to a halfword length field that is followed by the parameter string itself.

```
* z/OS PROGRAM Parameter List -------------------------------
PLIST  DC    A(PARMS+X'80000000')
                            ┬
                            │
┌───────────────────────────┘
│
▼
PARMS  DC    AL2(L'CMDSTR)
CMDSTR DC    C'original command string'
```

Figure 1. Specifying parameters for your program (TSO)

The COMMAND option (TSO)

To pass a parameter string as a standard TSO Command Processor Parameter List (CPPL), use the COMMAND option. The first token after the slash must be the command name, which is then followed by its parameter string.

```
* TSO COMMAND Parameter List -------------------------------
CPPL   DC    A(command buffer)
       DC    A(UPT)         │     copied from IDF invocation
       DC    A(PSCB)        │
       DC    A(ECT)         │
                            │
┌───────────────────────────┘
│
▼
CMDBUF DC    AL2(length of command buffer)
       DC    AL2(offset to first non-blank byte after verb)
CMDSTR DC    C'original command string'
```

The COMMAND option does not work exactly like the CP option of the TSO/E TEST command. The examples in Appendix E, “Migrating from TSO/E TEST to IDF,” on page 289 compare the use of these two options.

You can use the COMMAND option for other special environmental setups. For more details, see “Programs requiring environmental setup (TSO)” on page 45.

A warning about leading blanks

All blanks between the slash (/) and the parameters to your program are included in the information passed to your program. If you do not want initial blanks passed to your program, you must place the parameters immediately after the slash, without intervening blanks.

Programs that use IKJSCAN

When invoking IDF from another TSO program that calls the TSO service IKJSCAN, such as ISPF, an extra right parenthesis is added to the end of the command used to invoke IDF.

To prevent this extra parenthesis from becoming part of the parameter passed to your program, add a close parenthesis before the slash or the command option. This extra parenthesis is ignored by IDF and prevents IKJSCAN from adding one at the end of the IDF invocation command.

How to specify parameters for your program (z/OS Batch)

To pass parameters to your program, include them in the PARM= option on the EXEC ASMIDF statement.

```
►►──//stepname──EXEC──PGM=ASMIDFB,PARM='module-name──(──NOSVC97──LU──luname──────────────────►

►──┬──────────────────────────────────────┬──'───────────────────────────────────────────────►◄
   │ ┌──────────────┐                     │
   └─▼──idf-options─┴──┬─────────────────────┬─┘
                       └─/──module-parameters─┘
```

stepname
The name of the job step.

module-name
The name of the module to be debugged.

luname
The VTAM LU name of the terminal used to debug this module.

idf-options
Options to be passed to IDF.

module-parameters
Parameters to be passed to your module. These are passed to the program by IDF in standard z/OS EXEC PGM style. That is, R1 points to a word that points to a halfword length field that is followed by the parameter string itself.

Loading programs (TSO)

How IDF loads your target program is dependent on whether the LIBE option is used or not.

If the LIBE option is NOT specified, then IDF uses the standard z/OS load module search order. However after the load completes, IDF will attempt to read the module to determine the module map. This read will fail unless the module was loaded from a STEPLIB, TASKLIB or ISPLLIB (when in ISPF) data set.

If the LIBE option is specified, the next option token specifies the DD name of a pre-allocated DD concatenation to be used in loading the target. If the target is not in the specified concatenation of libraries the target is *not* loaded using the standard z/OS search order.

TASKLIBs and the SVC97 option

If the NOSVC97 option is specified, the TSO TEST SVC 97 support is not used for breakpoints.

In this case, IDF does *not* set up a TASKLIB, contrary to what is normally done when TSO/E CALL or TEST commands are used.

Programs that dynamically load other modules and programs that are link-edited as overlay programs may get z/OS System 106 ABENDs under IDF, unless the library in which the target program resides is in the STEPLIB, ISPLLIB, or other TASKLIB DD concatenation.

If MYPROG is a program that is normally invoked directly from the TSO/E READY prompt, you could debug it with the following IDF command:

```
ASMIDF myprog /Parms for MYPROG
```

if myprog is loaded from a TASKLIB or STEPLIB.

File allocation requirements

If MYPROG is a program that is normally executed in z/OS Batch, you must use the TSO ALLOC command to allocate needed data sets before invoking MYPROG under IDF. When you exit IDF, use the TSO FREE command to release these data sets.

Also remember to allocate and free the DDs that are needed for IDF operation. These include:

ASM Used to access the PDSs holding the IDF REXX EXECs used to customize IDF for a particular task.

ASMLANGX
Used to access the PDSs holding the IDF Language extract data files, if you want source-level debug.

If you plan to execute the program more than once under IDF, write these commands into a REXX EXEC or CLIST.

The TSOEXEC command

Invoke IDF with the TSO/E TSOEXEC command.

- This is *optional* when IDF is invoked:
  - Directly from the TSO/E READY prompt
  - From a CLIST that was invoked directly from the TSO/E READY prompt
- This is *needed* when IDF is invoked:
  - From a REXX EXEC
  - From a CLIST other than directly from the TSO/E READY prompt
  - Under ISPF.

Programs requiring environmental setup (TSO)

If RXMYPROG is a REXX function package that permanently loads itself as a system extension when first executed, and you are interested in debugging it when it is called by a REXX exec called TESTER, to which you must pass a parameter, then load the REXX function into storage, and invoke IDF:

```
ASMIDF rxmyprog (COMMAND/exec tester a
```

The COMMAND option

The COMMAND option tells IDF that the string following a slash (/) is not an argument string to be passed to your program, but is instead a command that it should issue to begin the debugging operation.

This COMMAND is LINKed to, and if it is an EXEC, you must precede it with a percent (%) sign. The percent sign signifies an implicit EXEC, and the command string is passed to the EXEC command for execution.

You can use this technique to debug user-area programs that need environmental setup.

When the COMMAND option is specified:

- IDF starts, but the registers and PSW are not displayed if the Current Registers window or Old Registers window is open.

- Set a breakpoint at the start of your program, or set a deferred breakpoint with the DBREAK command, and then press the RUN key.
- IDF then issues the command you specified.
- When the breakpoint is reached, IDF issues a message to tell you that it has reached a breakpoint.

  The registers and PSW are now available for inspection and modification. Before the first breakpoint, IDF does not display the registers and PSW.

TSO batch and z/OS batch job requirements

When IDF is started as a TSO batch or z/OS batch job, it needs a terminal (defined by its VTAM logical unit (LU) name) to run. The terminal is specified through the LUNAME option. The LUNAME is the name used by VTAM to define the LU to the network, and is used by IDF to address the terminal.

IDF uses VTAM application IDs that are defined according to the convention:
- The first five characters must be ASMTL
- The last three characters must be consecutive 3-digit numbers, starting at 001

There must be at least as many VTAM application IDs defined as there are concurrent IDF sessions. For example, if five IDF sessions are required at the same time, then there must be the following (at least) five VTAM application IDs defined: ASMTL001 ASMTL002 ASMTL003 ASMTL004 ASMTL005.

Dynamically loaded programs (TSO)

The IDF DBREAK command provides a powerful deferred breakpoint facility to allow the simple debugging of programs that are dynamically loaded with standard system calls. See “DBREAK” on page 109 for details.

Note: If your program is dynamically loaded, and then deleted from storage, any existing breakpoints are invalidated. Use the IDF DROP MODULE command to notify IDF that the module definition is no longer valid.

DROP MODULE removes all breakpoints that are currently defined for locations within this module. Any deferred breakpoints (from previous DBREAK commands) are reactivated in case this module is loaded into storage again, possibly at a different location.

If your program is dynamically loaded in a manner that is not supported by DBREAK (such as having your own routine that loads and relocates an object module), you need another solution.
- The basic need is for IDF to gain control when your program begins execution.
  - If you are using the SVC97 option, you should insert an SVC 97 (X'0A61') at the entry point of your program using one of the following methods:
    - If your program is written in assembler, insert an SVC 97 opcode in your program source.
    - If a Load Module editor is available, change the opcode at the entry point to an SVC 97. Record the original opcode for later restoration.
    - Use a binary editor (such as the ISPF (PDF) editor in HEX mode), to edit the Object Module to change the opcode at the entry point to an SVC 97. Record the original opcode for later restoration.
  - If you are using the NOSVC97 option, you should insert an invalid opcode (such as X'0000') at the entry point of your program using one of the following methods:
    - If your program is written in assembler, a DC X'0000' data directive in your program source.
    - If a Load Module editor is available, change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.
    - Use a binary editor (such as the ISPF (PDF) editor in HEX mode), to edit the object Module to change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.

See also “LANGUAGE LOAD” on page 129.

- Invoke IDF (use the COMMAND option, as described in “Programs requiring environmental setup (TSO)” on page 45), and press the RUN key.
- IDF issues the command you specified with the COMMAND option, and your program is dynamically loaded into memory. It then executes the instruction (SVC 97 or X'0000') that you inserted.
- IDF issues a message saying that either an operation exception occurred or a breakpoint was reached.
- At this point:
  - If an extra instruction (X'0000') or the SVC 97 was inserted, you need to open the Current Registers window, and use IDF's register typeover capability to update the PSW to advance the current execution address by 2.

    You can optionally replace the SVC 97 (or X'0000') by a NOPR R0 (X'0700') instruction. This stops IDF receiving control at this location if the instruction is executed again.
  - If an existing instruction was replaced (by an SVC 97 or X'0000'), you can restore the modified instruction to its original value by opening a Disassembly window or Dump window and using IDF's storage typeover capability.
- You should now be able to continue debugging the program as usual.

For information about source level support for dynamically loaded programs, see “Source level support” on page 90.

Programs invoked by REXX (TSO)

Some programs, particularly REXX function packages, need environmental setup for debugging because they are invoked by the REXX interpreter. There are other special considerations that apply to any program invoked by REXX.

REXX functions must return an EVALBLOK. If you quit from IDF while in the middle of a REXX function, an error message is issued by the REXX interpreter, and you may have to LOGOFF and then LOGON to TSO again.

There is no practical way for IDF to stop this.

ISPF applications (TSO)

You can use IDF to debug an application program that needs an ISPF environment present, if SVC 97 is used for breakpoints.

Notes:

1. The IDF session cannot be invoked from within ISPF. It must be invoked from the TSO/E ready prompt.
2. If the library containing the program to be debugged does not reside in the STEPLIB or ISPLLIB allocations, use TSOLIB ACTIVATE to include the load library, or allocate the library and invoke IDF using the LIBE option to specify the allocated DDNAME.

The following examples describe a number of ISPF application debug scenarios.

- To debug an ISPF dialog program which resides in a data set in the STEPLIB allocation:

  From the TSO/E READY prompt:
  ```
   TSOEXEC ASMIDF XYZZY  (COMMAND/ISPSTART PGM(XYZZY)
  ```
  Once in IDF,
  ```
  BREAK (XYZZY)
  ```
  then
  ```
  RUN
  ```

- To debug an ISPF dialog program which resides in a data set in the ISPLLIB allocation:

  From the TSO/E READY prompt:

  ```
      TSOEXEC ASMIDF XYZZY  (LIBE ISPLLIB COMMAND/ISPSTART PGM(XYZZY)
  ```

  Once in IDF,

  ```
    BREAK (XYZZY)
  ```

  then

  ```
    RUN
  ```

DB2 applications (TSO)

IDF may be used to debug an application program using DB2 EXEC SQL statements. There are a number of environmental requirements which must be met:

- The IDF NOSVC97 option must be used.
- If running application under the Language Environment, then the LE options NOSTAE and NOSPIE must be used.
- The DB2 commands must be placed on the TSO stack to be executed. The preferred method is to use a CLIST, and place the DB2 commands within DATA and ENDDATA statements.

```
PROC 0 LIBE('ASMIDF.SQL.LOAD') MEMBER(SQLPROG1) OPT('PARMS') IDF
/* */
CONTROL NOCONLIST FLUSH NOLIST
PROFILE NOPAUSE
/* */
ALLOC FI(ASMLANGX) REUSE DA('DMS.SQL.ASMLANGX') SHR
ALLOC FI(ASM) REUSE DA('ASMIDF.PROD.IDF') SHR
/* */
ALLOC FI(LIBEDD) REUSE DA('&LIBE') SHR
/* */
IF &IDF = IDF THEN +
  DO
    DATA
    DSN
    RUN CP PLAN(&MEMBER)
    TSOEXEC ASMIDF &MEMBER (LIBE LIBEDD NOSVC97 / &OPT
    END
    ENDDATA
  END
ELSE +
  DO
    DATA
    DSN
    RUN CP PLAN(&MEMBER)
    &MEMBER &OPT
    END
     ENDDATA
   END
 FREE FI(ASMLANGX LIBEDD ASM)
 EXIT CODE(0)
```

Causing a break-in event (TSO)

A "break-in event" is an event that causes IDF to break in to the target program's execution and regain control, even though a breakpoint was not reached. For example, you might need to cause a break-in event if your program is in an infinite loop that does not contain a breakpoint.

When IDF initially invokes your program, it establishes an ESTAE exit. If your program has not established its own ESTAE exit (which overrides the IDF ESTAE), IDF should receive control when the ATTENTION key is pressed.

When a break-in event is recognized, IDF displays the IDF user interface screen and issues a message acknowledging the event. You can then continue your debugging session.

Notes:
1. This requires SVC97 to operate normally.
2. There is no break-in event if the LUNAME option is specified when IDF is invoked.

Your program's defined limits

IDF considers "your program" to be any location within your initial program module, and any extra program modules that you have defined.
- When your program is loaded, IDF determines its limits from the operating system.
- You can *implicitly* define extra modules as the result of triggering a deferred breakpoint established with a DBREAK command.
- You can *explicitly* define extra modules with the IDF MODULE command:
  - If the module is described by a Contents Directory Entry (CDE) you can establish the module definition with a MODULE CDE command.
  - You can define an explicit module origin and size with the MODULE BASE and MODULE SIZE commands.
  - You can establish an explicit module's CSECT structure with a LOAD SYMBOLS command (see "LOAD" on page 138).

The origin and size of the programs known to IDF are displayed in the Target Status window, when open.

Whenever possible, IDF notifies you if your program is preparing to branch to a location outside its defined limits. This is often useful in locating a "wild branch", for example to location zero.

If you have specified the TRACEALL option, IDF considers the defined limits of your program to begin at location zero and extend upward to the end of storage. Thus when you specify TRACEALL, you can trace through all of virtual memory.

Care should be taken if you attempt to trace through protected (read only) storage.

Programs performing full-screen I/O (TSO)

Debugging programs that perform full-screen I/O is generally a difficult task, because when the debugger reaches a breakpoint, the screen image created by your program is lost. You may become frustrated if you need to look closely at the screen image to determine if the program is working properly.

IDF provides the SWAP option to make this kind of debugging easier. If MYPROG was a user-area program that performed full-screen I/O, you could debug it with the following IDF command:

```
IDF myprog (SWAP /Parms for MYPROG
```

When the SWAP option is in effect, IDF attempts to respect the screen image created by your program. In single-step mode, IDF does nothing special. However, if you set a breakpoint in your program and press the RUN key to run the target program to that point, before turning control over to your program, IDF restores any saved screen image. When the breakpoint is reached, IDF captures the contents of the screen and saves it before presenting its own display. The same process occurs when you use the UNTIL command.

To capture the screen image, IDF uses the Read Buffer command in Character Mode, so even program symbol sets should be saved.

To see the saved screen at a breakpoint, issue the SWAP command. To return to the IDF screen, just press ENTER.

Applications that use z/OS subtasking

To debug applications that make use of z/OS subtasking, IDF must be invoked with the SVC97 option (see “Breakpoint method selection (TSO)” on page 41).

Debugging z/OS applications that make use of subtasking can be unreliable. IDF cannot handle more than one event at a time. For example, a program check in one subtask while statement stepping in another subtask causes an ABEND within IDF.

Breakpoints can be set in different subtasks at the same time, but this is only reliable when only one subtask is “active” at a time (that is, the other subtask remains non-dispatchable). This requires judicious use of breakpoints at the appropriate points in the application. It is strongly recommended that you do not place breakpoints in code which can be executed by multiple tasks if it is possible for the breakpoints to be encountered simultaneously. Failure to do this can cause unpredictable results.

Chapter 6. Debugging programs on CMS

Program preparation on CMS

After all program sections are prepared and the process of building the final module is complete, save the LOAD MAP file. This gives IDF the location of program sections within the module. If multiple programs are being debugged, rename the file to "*modname* MAP" (where *modname* is the file name of the MODULE).

By default, any local symbols which are present in SYM records within the program object code are placed in the MAP file, with the prefix `Invalid card -` IDF uses this information if, for some reason, you cannot use the ASMLANGX files for source-level debugging. Suppress these records with the NOINV option of the CMS LOAD command if you wish to minimize the size of the MAP file.

On CMS, IDF debugs:
- User-Area Programs
- Transients
- Nucleus Extensions Loaded Explicitly
- Self-Loading Nucleus Extensions

IDF supports debugging programs in the following formats:
- CMS 5.5 (and later) format MODULE
- LOADLIB members
- OBJECT (TEXT) files
- Programs in storage

How to specify parameters for your program

To pass parameters to your program, include them at the end of the IDF command after a slash (/). IDF interprets anything that follows a slash as parameters that should be passed to your program.

These parameters are parsed into tokenized and extended parameter lists. The flag byte in R1 has the same contents as it has if your program is invoked directly. If IDF is invoked from an EXEC-1 program, the contents of R0 at the time IDF was invoked are propagated to your program's R0.

```
* CMS Parameter Lists --------------------------------------
* A) Extended Parameter List
EPLIST DC    A(start of command verb)
       DC    A(start of first nonblank byte following verb)
       DC    A(first byte past the end of the argument list)
       DC    A(0)
       DS    0D               align next part on doubleword
* B) Tokenized Parameter List
TPLIST DC    CL8'command verb'
       DC    CL8'token'
       ...                                        ...
       DC    CL8'token'
       DC    8X'FF'
CMDSTR DC    C'original command string'
```

A warning about leading blanks: All blanks between the slash (/) and the parameters to your program are included in the information passed to your program. If you do not want initial blanks passed to your program, you must place the parameters immediately after the slash, with no intervening blanks.

Unless your program is a nucleus extension (see “CMS nucleus extensions loaded explicitly”), IDF loads it into the appropriate area (user/transient) with the LOADMOD command.

For example, to debug MYPROG, normally invoked by:

```
MYPROG fn ft fm (abcd
```

You issue:

```
ASMIDF myprog / fn ft fm (abcd
```

If you want to specify more options for IDF, for example to set display colors, you specify:

```
ASMIDF myprog (COLORS RWGY / fn ft fm (abcd
```

User-area programs

By default, CMS programs are built to execute in the *User Program Area* at location X'20000'.

Unless told otherwise, IDF assumes that you want to debug a user-area program. There are no mandatory options to specify when debugging a user-area program (although there are options you can specify to enable special features).

CMS transient programs

Transient-area programs are special non-relocatable programs that execute in the *Transient Program Area* at location X'00E000'.

Under IDF, these programs are debugged in the same way as user-area programs but the TRANS option must be specified so that IDF knows that you want to debug a transient.

For example, if MYPROG shown above is a transient, you specify:

```
ASMIDF myprog (TRANS / fn ft fm (abcd
```

When debugging a transient-area program, you do not have access to CMS SUBSET during the debugging session. Running another transient erases your program.

CMS nucleus extensions loaded explicitly

Nucleus extension programs are typically relocatable programs. They are loaded into "high" storage, typically with the CMS NUCXLOAD command. These programs become an extension to the CMS nucleus, and remain resident until removed by the CMS NUCXDROP command.

If MYPROG is a nucleus extension that you have loaded with the NUCXLOAD command you invoke IDF:

```
ASMIDF myprog (NUCEXT /fn ft fm (abcd
```

As far as IDF is concerned, this is the only difference between debugging a nucleus extension and a user-area program. You must load the nucleus extension before invoking IDF.

If the nucleus extension was loaded from a LOADLIB and not a MODULE file, use the LIBE $ option as well as the NUCEXT option. This tells IDF to not check the MODULE file.

Self-loading CMS nucleus extensions

Some nucleus extensions are self-loading. They obtain nucleus free storage themselves, move in the code, and declare the nucleus extension. In this case, you must specify the symbol where the code that is moved into nucleus storage begins.

For example, if your program is a REXX function package, there is a short piece of code at the start of the program that loads the remainder as a nucleus extension. The real program begins at symbol FREEGO. You invoke IDF:

```
ASMIDF rxmyprog (SELFNUCX FREEGO/fn ft fm (abcd
```

You must pre-load the nucleus extension before invoking IDF. Since the program is self-loading and may not yet be operational, you may have to use a two-run technique with IDF:

1. First run:

   For the first run, do *not* declare the program as self-loading. Invoke IDF as if the program was only going to allocate nucleus storage. Trace through the program up to the point where it normally branches to the NUCXLOADed code, then QUIT from IDF. At this point, you have effectively pre-loaded the nucleus extension.
2. Second run:

   For the second run, invoke IDF with the SELFNUCX option as shown above. Since you pre-loaded it in the first run, you can now continue debugging in the NUCXLOADed code.

Note: It may also be possible to use the SET BASE and SELFNUCX VALUE commands to trace a self-loading nucleus extension in a single run.

Programs requiring environmental setup

RXMYPROG is a REXX function package that permanently loads itself as a system extension when first executed (for example, a self-loading nucleus extension on CMS). You want to debug it when it is called by a REXX exec called TESTER, to which you must pass a parameter.

Load the REXX function into storage with the command:

```
NUCXLOAD rxmyprog
```

Next, issue the IDF command:

```
ASMIDF rxmyprog (SELFNUCX FREEGO COMMAND/exec tester a
```

The COMMAND option

The COMMAND option tells IDF that the string following a slash (/) is not an argument string to be passed to the target program, but a command that it should issue to begin the debugging operation.

This command is issued with a CMSCALL (SVC 204), so if it is an exec, you must specify that, as shown in the example above.

You can use the same technique to debug user-area programs that need environmental setup.

When the COMMAND option is specified:

- IDF starts as usual but PER is disabled, and the registers and PSW are not displayed if the Current Registers window or Old Registers window is open.
- Set a breakpoint at the start of your program, or set a deferred breakpoint with the DBREAK command, and then press the RUN key.
- IDF then issues the command you specified.
- As part of that command's execution, it should invoke the target program (RXMYPROG in this example), which is already in storage, defined to IDF, and with the breakpoint installed at or near its entry-point.
- When the breakpoint is reached, IDF issues a message to you.

  PER is now available for use, and the registers and PSW are available for inspection and modification. Before the first breakpoint IDF keeps PER disabled and does not display the registers and PSW.

If practical, when using the COMMAND option, allow the target program to execute through to the point of its return to its caller (for example, to the EXEC that was invoked by the COMMAND string and which then invoked the target program). This “normal flow of control” allows any usual “house-cleaning” needed by the operating system for these call/return linkages. If instead you QUIT from the target program before its normal return, it may leave a “dangling” call linkage to the target program, without a corresponding return.

Quitting from a target program before a normal return is a particular problem on CMS. The CMSCALL (SVC 204) that was used from within IDF to call the COMMAND string is not returned to. If this “un-paired” SVC linkage remains undetected, you may experience difficulties in CMS.

IDF termination contains special logic to check for this condition (that is, a QUIT from the target program that was invoked with a COMMAND string), and if detected, IDF:

1. Issues a linemode message warning of the problem.
2. Executes one or more `CMSRET RC=999` macros until the original SVC nesting (as saved just before IDF's CMSCALL to the COMMAND string) is restored.

If the original nesting cannot be restored within 20 calls to CMSRET, or if a CMS ABEND occurs during this processing, IDF issues another warning message (suggesting that CMS be re-IPLed) and terminates.

Dynamically loaded programs

The IDF DBREAK command provides a powerful deferred breakpoint facility to allow the simple debugging of programs that are dynamically loaded with standard system calls. See “DBREAK” on page 109 for details.

Attention: If your program is dynamically loaded, and then deleted from storage, any existing breakpoints are invalidated. Use the IDF DROP MODULE *module-name* command to notify IDF that the module definition is no longer valid.

DROP MODULE removes all breakpoints that are currently defined for locations within this module. Any deferred breakpoints (from previous DBREAK commands) are reactivated in case this module is loaded into storage again, possibly at a different location.

If your program is dynamically loaded in a manner that is not supported by DBREAK, such as having your own routine that loads and relocates an object module, you need an alternative solution.

IDF must be able to gain control when your program starts executing.

- Insert an invalid opcode (such as X'0000') at the entry point of your program using one of the following methods:
  - If your program is in assembler, insert a DC X'0000' data directive in its source.
  - If a MODULE editor is available, change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.
  - Use a binary editor (such as the ISPF (PDF) editor in HEX mode), to edit the object Module to change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.

Invoke IDF (you need to use the COMMAND option, as described in “Programs requiring environmental setup” on page 53), and press the RUN key.

IDF issues the command you specified with the COMMAND option, and your program is dynamically loaded into memory. It then executes the invalid instruction (X'0000') that you inserted.

IDF issues a message saying that either an operation exception has occurred, or that a breakpoint was reached.

At this point:

- If an extra instruction (X'0000') was inserted, you need to open the Current Registers window, and use IDF's register typeover capability to update the PSW to advance the current execution address by 2.

  You can optionally replace the X'0000' by a NOPR R0 (X'0700') instruction. This stops IDF receiving control at this location if the instruction is executed again.
- If an existing instruction was replaced (by an X'0000'), you can restore the modified instruction to the original value by opening a Disassembly window or Dump window and using IDF's storage typeover capability.

You should now be able to continue debugging the program as usual.

For information about source level support for dynamically loaded programs, see "Source level support" on page 90.

Programs invoked by REXX

Some programs, particularly REXX function packages, need environmental setup for debugging because they are invoked by the REXX interpreter. There are other special considerations that apply to any program that is invoked by REXX.

REXX functions must return an EVALBLOK. If you quit from IDF while in the middle of a REXX function, an error message is issued by the REXX interpreter, and you may have to re-IPL CMS.

There is no practical way for IDF to prevent this from happening.

Programs declaring interrupt routines

If your program declares interrupt routines with ABNEXIT, HNDEXT, or their z/OS equivalents read this section.

Remember that some interrupt routines are not automatically cleared by CMS at end of command. ABNEXIT is one of these. If you are in a debugging session and your program has declared an ABNEXIT routine, if you quit from IDF through PF3 before your program explicitly clears the exit routine, the exit routine remains in effect even when your program is no longer in memory. The next time you run a program or invoke IDF for another debugging session, a system abend is most likely, after which you need to re-IPL CMS.

It is safest if you re-IPL CMS after a debugging session where exit routines were tested. The results after IDF completes are unpredictable.

Causing a break-in event

A "break-in event" is an event that causes IDF to break in to the target program's execution and regain control, even though no breakpoint was reached. For example, you might need to cause a break-in event if your program has gone into an infinite loop that does not contain a breakpoint.

To cause a break-in event, you need some preparation:

- You need to know the address of IDF's ISA (interrupt save area). If you have not specifically set it with the ISA option when invoking IDF, the default value for the ISA location is in effect, (that is, 16 bytes at X'500').
- You can cause a break-in event by obtaining the CP command prompt (typically by pressing the PA1 key), and storing any value in the first doubleword of IDF's ISA. For example: "STORE S500 00".

  Do not modify the second doubleword of the ISA. Any store operation that changes the first doubleword (or any part of it) causes a break-in event at the next interrupt.

If your program has gone into an infinite loop however, there may not be any interrupts. You can ensure a steady supply of interrupts by setting the PATH option. This makes IDF take an interrupt after each target program instruction.

When a break-in event is recognized, IDF displays the IDF user interface screen and issues a message acknowledging the break-in event. You can then continue the debugging session.

Your program's defined limits

IDF considers "your program" to be any location within the initial target program module, and any more program modules that you have defined.

- When the target is loaded, IDF determines its limits by examining the module file.
- Extra modules are implicitly defined as the result of the triggering of a deferred breakpoint that was established with a DBREAK command.
- Extra modules are explicitly defined with the IDF MODULE command:
  - If the module is a nucleus extension, establish the module definition with a MODULE NUCEXT command.
  - If the module is a transient program, establish the module definition with a MODULE TRANS command.
  - Define an explicit module origin and size with MODULE BASE and MODULE SIZE commands.
  - You can establish an explicit module's CSECT structure with a LOAD SYMBOLS command (see "LOAD" on page 138).

The origin and size of the programs known to IDF are displayed in the Target Status window, when open.

Whenever possible, IDF tells you if your program is about to branch to a location outside its defined limits. This is often useful in finding a "wild branch", for example to location zero.

If you have specified the TRACEALL option, IDF considers the defined limits of your program to begin at location zero and extend upward to the address specified in the VMSIZE word of NUCON. Thus when you specify TRACEALL, you can trace through all of virtual memory.

Take care if you attempt to trace through protected (read only) storage.

If you specify the RISK option, IDF considers all of memory to be within your program's defined limits. Thus you could attempt to step through code in a DCSS that is actually above the address specified in the VMSIZE word of NUCON.

PER versus non-PER mode

The Break panel provides a means of enabling or disabling what is referred to as "PER mode" operation. The method of enabling and disabling PER mode is discussed later; this section explains the difference between the two modes of operation.

PER is an acronym for "Program Event Recorder". It is a feature of the z/Architecture hardware, and is only available in EC (Extended Control) mode.

When PER mode is enabled through the Break window (the default is for IDF to disable it), IDF puts the processor in EC mode before turning control over to the target program. This lets IDF use the PER instruction fetch, register alteration, and storage alteration features to monitor your program's execution. Single-stepping and breakpoints are available in either mode. Register stops and address stops (storage alteration stops) are available only in PER mode.

Whenever possible, IDF implements breakpoints by means of inserting an invalid opcode at the break address. In the case of a read-only DCSS, this is not possible. In this case IDF needs to use PER instruction fetch events to implement breakpoints. This can only be done when you have set PER=Y.

PER mode's benefits are:
- Register alteration stops (RegStops) are available
- Storage alteration stops (AdStops) are available

Its disadvantage is:
1. the number of instructions executed by IDF when PER=Y is slightly higher than when PER=N.

The disadvantage of non-PER mode is that RegStops and AdStops are not available.

When you debug your program, allowing for these benefits and disadvantages leads to better results and fewer surprises.

You are not committed to either mode for the entire debugging session. Whenever IDF reaches a breakpoint and is therefore able to display the Break window, you can change modes.

Programs performing full-screen I/O

Debugging programs that perform full-screen I/O is generally a difficult task, because when the debugger reaches a breakpoint the screen image created by your program is lost. This can be extremely frustrating, since you may need to look closely at the screen image to determine whether the program is operating correctly.

IDF provides the SWAP option to make this kind of debugging easier. If MYPROG was a user-area program that performed full-screen I/O, you could debug it with:

```
ASMIDF myprog (SWAP /fn ft fm (abcd
```

When the SWAP option is in effect, IDF attempts to respect the screen image created by your program. In single-step mode, IDF does nothing special when SWAP is in effect. However, if you set a breakpoint in your program and press the RUN key to run the target program to that point, before turning control over to your program, IDF restores any saved screen image. When the breakpoint is reached, IDF captures the contents of the screen and saves it before presenting its own display. The same process occurs when you use the UNTIL command.

To capture the screen image, IDF uses the READ BUFFER command in Character Mode, so even program symbol sets should be saved.

If you have two terminal sessions available, then you can use the LINE option. The LINE option tells IDF to use the GRAF at the specified address for its I/O, instead of using the virtual console.

A sample procedure to do this is:

```
CP DEFINE GRAF xxx
DIAL userid xxx        <-- (from the alternate session)
ASMIDF MYPROG (LINE X'xxx'
```

If you use the LINE option instead of the SWAP option, your debug session is faster, because there is no need to constantly capture and re-write your program's screen.

To see the saved screen at a breakpoint, issue the SWAP command. To return to the IDF screen, just press ENTER.

Using a message-trapping tool

Many users run a message-trapping tool as a normal part of their working environment.

IDF sets up a dummy HNDEXT routine while it is in control, so that if a clock or other external interrupt occurs, the "DEBUG ENTERED" scenario is avoided.

If the target program sets up an HNDEXT routine, that new routine supersedes IDF's dummy handler without interfering with IDF operation. If IDF was invoked by another program that had set up a HNDEXT routine, IDF's dummy handler supersedes it.

Therefore, if you use a message-trapping tool that relies on HNDEXT processing to gain control and thus intercept messages, you should disable it before starting IDF to make sure you do not lose any messages.

Many message trappers "steal" the external new PSW and perform their own interrupt handling at that level. If the message trapper you use does this, it is likely that it can coexist with IDF. It is suggested that you test it to make sure, if you are concerned about the possibility of losing messages that arrive during a debugging session.

IDF does not affect the MSG setting, so if you do not use a message-trapping tool, none of this should affect you.

Chapter 7. Debugging programs on z/VSE

On z/VSE, IDF supports debugging of phases that are loaded into the same partition as IDF.

Note: Phases loaded into the SVA or Logical Transients cannot be debugged.

Data set naming conventions

Certain IDF commands cause data to be written to or read from files. Since IDF was originally written to run on CMS, the commands are oriented towards the naming conventions used by the CMS file system.

The mapping of the CMS file conventions to z/VSE is:

CMS	**Equivalent on z/VSE**
fn	z/VSE librarian member name.
ft	DLBL name, which in turn points to the z/VSE data set name
fm	Not used on z/VSE

On z/VSE, you must reference files using DLBL statements (JCL), and libraries using LIBDEF statements.

How to specify parameters for your program

To pass parameters to your program, include them in the PARM= option on the EXEC ASMIDF statement.

```
 // EXEC ASMIDF,PARM='phasename (LU luname idf-options/phase-parameters'
```

where:

phasename
Is the name of the phase to be debugged

luname
Is the VTAM LU name of the terminal you use to debug this phase

idf-options
Are options to be passed to IDF

phase-parameters
Are parameters to be passed to your phase

Loading programs

The target program being debugged is loaded from the first sublibrary in the LIBDEF PHASE SEARCH list in which it occurs.

JCL requirements

Because IDF runs from JCL, you need to specify any files that your program needs in that JCL. You need to allocate optional IDF files, such as REXX customization macros and IDF Language extract files. Even though IDF is started as a batch job, it needs a terminal (defined by its VTAM *logical unit* (LU) name) to run. This terminal is specified through the LUTERM option in the PARM value of the `// EXEC ASMIDF` statement. The LUTERM is the name used by VTAM to define the LU to the network, and is used by IDF to address the terminal.

JCL example:

```
// SETPFIX LIMIT=24K
// LIBDEF PHASE,SEARCH=(my.library,HLASM.LIBRARY)
// LIBDEF PROC,SEARCH=(MY.PROCLIB,HLASM.LIBRARY)
// EXEC ASMIDF,PARM='MYPROG (LU MYterm /phase-parameters'
/*
```

The following JCL statements are required:

SETPFIX LIMIT
Sets the page fix limit. IDF needs this set to 24K to allow the product exit to function correctly.

LIBDEF PHASE
Defines the library which contains the phase map created by ASMLKEDT and the target phase.

LIBDEF PROC
Defines the procedure library which contains any REXX procedures that might be used.

EXEC ASMIDF
Executes the program ASMIDF (IDF) using the parameters specified with the PARM option.

Dynamically loaded programs

The IDF DBREAK command provides a powerful deferred breakpoint facility to allow the simple debugging of programs that are dynamically loaded using standard system calls. See "DBREAK" on page 109 for details.

Attention: If your program is dynamically loaded, and then deleted from storage, any existing breakpoints are invalidated. Use the IDF DROP MODULE command to notify IDF that the module definition is no longer valid.

DROP MODULE removes all breakpoints that are currently defined for locations within this module. Any deferred breakpoints (from previous DBREAK commands) are reactivated in case this module is loaded into storage again, possibly at a different location.

If your program is dynamically loaded in a manner that is not supported by DBREAK (such as having your own routine that loads and relocates an object module), you need to use an alternative solution.

- IDF needs to gain control when your program starts executing.

  You should insert an invalid opcode (such as X'0000') at the entry point of your program using one of the following methods:
  - If your program is written in assembler, insert a DC X'0000' data directive in your program source.
  - If a phase editor is available, change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.
  - Use a binary editor to edit the object Module to change the opcode at the entry point to X'0000'. Record the original opcode for later restoration.
- Invoke IDF and press the RUN key, on the terminal that you have specified in the *LUNAME* parameter on the EXEC ASMIDF statement.
- Your program is dynamically loaded into memory and executes the instruction (X'0000') that you inserted.
- IDF issues a message saying that an operation exception has occurred.
- At this point:
  - If an extra instruction (X'0000') was inserted, you need to open the Current Registers window, and use IDF's register typeover capability to update the PSW to advance the current execution address by 2.

    You may optionally replace the X'0000' by a NOPR R0 (X'0700') instruction. This stops IDF receiving control at this location if the instruction is executed again.

– If an existing instruction was replaced (by X'0000'), you can restore the modified instruction to its original value by opening a Disassembly window or Dump window and using IDF's storage typeover capability.

- You should now be able to continue debugging the program as usual.

For information about source level support for dynamically loaded programs, see "Source level support" on page 90.

Running with subtasks

IDF detects program loads, program checks and breakpoints defined by commands for all the tasks running within a partition. To achieve this IDF uses a locking mechanism to single thread each event detected. The lock is in effect from the time the event is detected and is released when control is returned to the respective task. If IDF is interrupted while the LOCK is in effect by a STXIT IT, or STXIT OC for example, a deadlock situation may occur. Take care, when you are debugging subtasks and interruption routines are in effect, that you do not interrupt the IDF session.

Running with CICS

CICS uses subtasks, therefore to avoid the deadlocks described in the previous section you should run CICS with:

1. Run away task inoperative (that is, SIT option ICVR=0).
2. Stall Purge should be disabled for all transactions that run during the debugging session.

When issuing `MSG CICS` partition console commands on debug sessions, do not interrupt the IDF session.

Using ASMIDF to debug a CICS/VSE application

ASMIDF does not run under CICS. It is possible, however, to run CICS/VSE under ASMIDF as for a batch application. Here's how:

1. Ensure the CICS job's LIBDEF search list includes the Toolkit and HLASM sublibraries plus any libraries that contain REXX procs, ASMLANGX file, and so on, to be used in the IDF session.
2. Replace the EXEC DFHSIP statement with the following:

   ```
   // SETPFIX LIB=144K
   // EXEC ASMIDF,SIZE=9M,PARM=DFHSIP (LU luid/SI'                    X
                 DSPACE=2M
   SIT=??
   DCT=??
   GRPLIST=VSELST
   START=AUTO
   EXTSEC=NO
   SVD=NO
   ICVR=0
   TCT=??
   APPLID=??
   $END
   ```

   where **??** are installation dependent **luid** is the LU of the terminal at which the IDF session is to be run **ICVR=0** is required
3. When the IDF command line appears, issue a deferred breakpoint for the application program that is to be debugged: `DBREAK (program-name.)+x'20'`
4. Allow IDF to RUN so that CICS can start.
5. Once CICS is up, run the transaction that involves the program for which the DBREAK has been issued. IDF should then display a breakpoint at entry to the program.

Debugging STXIT code

STXIT PC
IDF traps program checks before any STXIT code is invoked. This lets you analyze the initial program check. If you have an active STXIT PC you can then set breakpoints in your STXIT PC code and use the IDF RUN command to break in the STXIT PC code.

Note: You may need to issue the RUN command twice when a branch address is invalid, before the STXIT PC code is invoked.

STXIT AB
IDF lets you debug your STXIT AB routines if you set breakpoints in them. Care should be taken when debugging STXIT AB routines with subtasks. If another event is trapped by IDF at the same time as the STXIT AB breakpoint, IDF may terminate the session due to a possible deadlock situation.

STXIT IT, OC
IDF lets you debug your STXIT IT, OC routines if you set break points in them. Refer to "Running with subtasks" on page 61 for considerations when running with subtasks.

Causing a break-in event

A break-in event is an event that causes IDF to break in to the target program's execution and regain control, even though no breakpoint was reached. For example, you might need to cause a break-in event if your program has gone into an infinite loop that does not contain a breakpoint.

When IDF initially invokes your program, it has established a STXIT OC exit. If your program has not established its own STXIT OC exit (which overrides the IDF STXIT OC), IDF should receive control when the console command 'MSG partition ID' is entered. This may cause corruption of any open data sets

When a break-in event is recognized, IDF displays the IDF user interface screen as usual and issues a message acknowledging the event. You can then continue your debugging session.

Your program's defined limits

IDF considers "your program" to be any location within the initial target phase, and any more phases that you have defined.

- When the target is loaded, IDF determines its limits by examining the directory entry for the phase.
- Extra modules are implicitly defined as the result of the triggering of a deferred breakpoint that was established with a DBREAK command.
- Extra phases are explicitly defined with the IDF MODULE command:
  - If the module is described in the phase load trace table, establish the module definition with a MODULE command.
  - Define an explicit module origin and size with MODULE BASE and MODULE SIZE commands.
  - You can establish an explicit module's CSECT structure with a LOAD SYMBOLS command (see "LOAD" on page 138).

The origin and size of the programs known to IDF are displayed in the Target Status window, when open.

Whenever possible, IDF notifies you if your program is preparing to branch to a location outside its defined limits. This is often useful in locating a "wild branch", for example to location zero.

If you have specified the TRACEALL option, IDF considers the defined limits of your program to begin at location zero and extend upward to the end of storage. Thus when you specify TRACEALL, you can trace through all of virtual memory.

Care should be taken if you attempt to trace through protected (read only) storage.

Programs performing full-screen I/O

Debugging programs that perform full-screen I/O is generally a difficult task, because when the debugger reaches a breakpoint, the screen image created by your program is lost. This can be extremely frustrating, since you may need to look closely at the screen image to determine whether the program is operating correctly.

IDF provides the SWAP option to make this kind of debugging easier. If MYPROG was a program that performed full-screen I/O, you could debug it with the following IDF command:

```
 // EXEC ASMIDF PARM='myprog (LU myterminal SWAP /Parms for MYPROG'
```

When the SWAP option is in effect, IDF attempts to respect the screen image created by your program. In single-step mode, IDF does nothing special. However, if you set a breakpoint in your program and press the RUN key to run the target program to that point, before turning control over to your program, IDF restores any saved screen image. Then when the breakpoint is reached, IDF captures the contents of the screen and saves it before presenting its own display. The same process occurs when you use the UNTIL command.

To capture the screen image, IDF uses the READ BUFFER command in Character Mode, so even program symbol sets should be saved.

To see the saved screen at a breakpoint, issue the SWAP command. To return to the IDF screen, just press enter.

Chapter 8. Windows, PF keys, cursor positioning, and other operational details

This chapter provides more detail on the operation of IDF.

Windows

IDF has many different windows:
- AdStops window
- Additional Floating-Point Registers window
- Break window
- Current Registers window
- Disassembly window
- Dump window
- Entry Point Names window
- LSM Information window
- Minimized Windows Viewer
- Options window
- Old Registers window
- Skipped Subroutines window
- Target Status window

There is also a Command window which is always displayed. It contains the command input area, the message display areas, and, by default, the settings of PF keys 1 to 12. Use the SET PFKDISP command to customize the number of PF key settings displayed. The size of the Command window varies, depending on the number of PF key settings displayed. This is the only window displayed by default when IDF is started.

Each window when displayed is surrounded by a border that includes a title describing that window. The title includes the window's window number. The window number is an integer assigned in sequence as windows are opened. When a window is closed, the window numbers of open windows are reassigned.

By default, when windows are opened, IDF positions them on the screen so that their top border overlays the bottom border of the window last opened. The exceptions to this rule are:
- AdStops window
- Break window
- Skipped Subroutines window

These windows are placed on the right of the screen.

As windows are opened they are added to the bottom of the screen. Use the ORDER command to move a window to the front of the list of windows being displayed.

The Additional Floating-Point Registers window, Current Registers window, Old Registers window, the Options window, and Target Status window are of a fixed size when opened.

By default, any open Disassembly windows, Dump windows, and LSM Information windows share the space on the screen not used by the other windows. They are automatically re-sized every time some window opens or closes. LSM Information windows use at most enough space to contain the information generated by the command.

Change this default behavior with the NOAUTOSZ option (AUTOSIZE is the default), or with the SET OPTION OFF AUTOSIZE command.

When AUTOSIZE is off, all open windows are placed in the upper left corner of the screen.

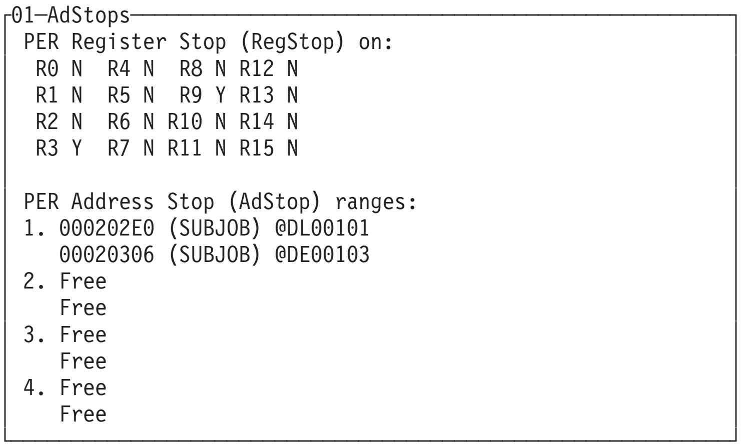
You can position windows manually with the MOVE command.

AdStops window (CMS only)

The AdStops window can only be opened when PER operation is active. It is opened by issuing the ADSTOPS command without providing any arguments. To close the window, issue the ADSTOPS command again without any arguments or issue the CLOSE command against that window. When the AdStops window is opened, it is positioned at the bottom right corner of the screen, and overlays any information there.

Alternatively, you can use the REGSTOPS command.

The AdStops window lets you set and reset register alteration stops (RegStops) or clear storage alteration stops (AdStops).



```
┌01─AdStops──────────────────────────────────────────────┐
│ PER Register Stop (RegStop) on:                        │
│  R0 N  R4 N  R8 N R12 N                                │
│  R1 N  R5 N  R9 Y R13 N                                │
│  R2 N  R6 N R10 N R14 N                                │
│  R3 Y  R7 N R11 N R15 N                                │
│                                                        │
│ PER Address Stop (AdStop) ranges:                      │
│ 1. 000202E0 (SUBJOB) @DL00101                          │
│    00020306 (SUBJOB) @DE00103                          │
│ 2. Free                                                │
│    Free                                                │
│ 3. Free                                                │
│    Free                                                │
│ 4. Free                                                │
│    Free                                                │
└────────────────────────────────────────────────────────┘
```

Figure 2. AdStops window

To clear a storage alteration stop, just place the cursor on the field describing the stop you want to clear, and either press the ERASE-EOF key or overtype the field with blanks. You must clear both the start and the end address.

Change the register stop settings by overtyping the values shown.

Additional Floating-Point Registers window

Open the Additional Floating-Point Registers window by issuing the AFPR command. Close the window by issuing the AFPR command again, or by issuing the CLOSE command against that window.

The window lists the active contents of the Floating-Point Control register, and the Additional Binary Floating-Point registers, that is, FPR1, 3, 5, 7 to 15.

```
┌02─Additional Floating Point Registers ├──────────────────────────────────────┐
│ Floating Point Control Register 00000000                                     │
│ FPR01 0000000000000000  FPR08 0000000000000000  FPR09 000000000000000        │
│ FPR03 0000000000000000  FPR10 0000000000000000  FPR11 000000000000000        │
│ FPR05 0000000000000000  FPR12 0000000000000000  FPR13 000000000000000        │
│ FPR07 0000000000000000  FPR14 0000000000000000  FPR15 000000000000000        │
└──────────────────────────────────────────────────────────────────────────────┘
```

Figure 3. An example of the Additional Floating-Point Registers window

You can change the contents of the registers by overtyping the hex values displayed.

Break window

Open the Break window by issuing the BREAK command without providing any arguments. Close the window by issuing the BREAK command again without any arguments or by issuing the CLOSE command against that window. When the Break window is opened, it is positioned at the top right corner of the screen, overlaying any existing information. It uses only as many rows as needed to display the breakpoint information.

The window lists the active breakpoints and watchpoints. For watchpoints, the address is preceded by a "w" and the condition are displayed on the next line. Any commands associated with the breakpoint or watchpoint are shown on the following line. If there are more breakpoints or watchpoints than fit on the screen the NEXT and PREVIOUS commands scroll forwards and backwards through the list.

The Break window also lets you clear breakpoints. Breakpoints remain in effect until you explicitly clear them.

To clear a breakpoint, place the cursor on the field describing the breakpoint you want to clear, and either press the ERASE-EOF key or overtype the field with blanks.

```
┌01─Current Registers───┬05─Break Points──────────────────────────────────┐
│ (TCAT) @PROLOG+44     │ w00057752 (TCAT) @DL00029                       │
│  R0 00009025  R1 0001 │   Condition:  = c r3,=f'3'                      │
│  R4 FEFE040F  R5 FEFE │  00057788 (TCAT) LOCRET                         │
├  R8 FEFE080F  R9 0005 └─────────────────────────────────────────────────┤
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
├02─Old Registers─────────────────────────────────────────────────────────┤
│ (TCAT) @PROLOG+40                      PSW 078D10008005771E (CC mask= 4 L) │
│ 0005771E 9620 C0BA                 OI    CTGOPTN3,32                     │
│  R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000 │
│  R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000 │
│  R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000 │
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
├03─Disassembly───────────────────────────────────────────────────────────┤
│ (TCAT) @PROLOG+32                                                       │
│  00057716 92C1 C0CA                 MVI   CTGTYPE,193                   │
│  0005771A 943F C0BA                 NI    CTGOPTN3,63                   │
│  0005771E 9620 C0BA                 OI    CTGOPTN3,32                   │
│  00057722 4190 07D8                 LA    R9,2008                       │
│  00057726 5090 C108                 ST    R9,CATWRK                     │
│  0005772A 1F88                      SLR   R8,R8                         │
│  0005772C 5080 C10C                 ST    R8,CATWRKUS                   │
│  00057730 4190 C108                 LA    R9,CATWRK                     │
│  00057734 5090 C0C4                 ST    R9,CTGWKA                     │
│  00057738 4170 0002                 LA    R7,2                          │
├04─Storage Dump──────────────────────────────────────────────────────────┤
│ (TCAT) CTGOPTN3                                                         │
│  0005779A      21                              .                        │
│ (TCAT) CTGOPTN4                                                         │
│  0005779B      00                             .                         │
│ (TCAT) CTGENT                                                           │
│  0005779C 000577BC                         ....                         │
│ (TCAT) CTGCAT                                                           │
│  000577A0 00000000                         ....                         │
│ (TCAT) CTGWKA                                                           │
│  000577A4 00000000                         ....                         │
│ (TCAT) CTGDSORG                                                         │
└─────────────────────────────────────────────────────────────────────────┘

==>

1 Stmtstep   2 Regs       3 Quit       4 Until       5 Run        6 Dump
7 Previous   8 Next       9 Disasm    10 Break      11 Step       12 Retrieve
```

Figure 4. An example of the Break window overlaying other windows

z/VM The Break window lets you enable or disable SVC trapping and PER operation. Change the SVC and PER settings by overtyping the values shown.

Current Registers window

The Current Registers window is opened by entering the REGS or REGS64 command. It is closed by issuing the REGS or REGS64 command again or by issuing the CLOSE command against that window.

By default it displays the current PSW, the current general purpose and floating point registers. If the CREGS command was issued the current control registers are displayed. If the AREGS command was issued the current access registers are displayed.

```
┌01─Current Registers──────────────────────────────────────────────────────┐
│(TCAT) @PROLOG+84                         PSW 078D20008005774A (CC mask=2 H)│
│ R0 00009025  R1 000120D4  R2 00057FC0  R3 00000001 FPR0 0000000000000000 │
│ R4 FEFE040F  R5 FEFE050F  R6 00000001  R7 00000002 FPR2 0000000000000000 │
│ R8 00000000  R9 000577E8 R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000 │
│R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
└──────────────────────────────────────────────────────────────────────────┘
```

Figure 5. Current Registers window, with General Purpose and Floating Point Registers

```
┌01—Current Registers──────────────────────────────────────────────────────┐
│ Access Registers:                       PSW 03E800008005774A (CC mask=2 H)│
│  A0 00000000  A1 00000001  A2 00000000  A3 00000000                       │
│  A4 00000000  A5 00000000  A6 00000000  A7 00000000                       │
│  A8 00000000  A9 00000000 A10 00000000 A11 00000000                       │
│ A12 00000000 A13 00000000 A14 00000000 A15 00000000                       │
└───────────────────────────────────────────────────────────────────────────┘
```

Figure 6. Current Registers window, with Access Registers

```
┌01—Current Registers──────────────────────────────────────────────────────┐
│ Control Registers:                      PSW 03E800008005774A (CC mask=2 H)│
│  C0 000100E0  C1 00000000  C2 00000000  C3 00000000                       │
│  C4 00000000  C5 00000000  C6 FF000000  C7 00000000                       │
│  C8 00000000  C9 00000000 C10 00000000 C11 00000000                       │
│ C12 00000000 C13 00000000 C14 1F000000 C15 00000000                       │
└───────────────────────────────────────────────────────────────────────────┘
```

Figure 7. Current Registers window, with Control Registers

```
┌01—Current Registers──────────────────────────────────────────────────────┐
│ (TESTVAR) TESTVAR                                                         │
│              EPSW FF00000000000000 000000000000ACC0 (CC mask=8 E)         │
│  R0 00000000 FEFE000F  R8 00000000 FEFE080F FPR0 0000000000000000         │
│  R1 00000000 0000D024  R9 00000000 FEFE090F FPR2 0000000000000000         │
│  R2 00000000 FEFE020F R10 00000000 FEFE0A0F FPR4 0000000000000000         │
│  R3 00000000 FEFE030F R11 00000000 FEFE0B0F FPR6 0000000000000000         │
│  R4 00000000 FEFE040F R12 00000000 0000ACC0                               │
│  R5 00000000 FEFE050F R13 00000000 0000D058                               │
│  R6 00000000 FEFE060F R14 00000000 00090466                               │
│  R7 00000000 FEFE070F R15 00000000 0000ACC0                               │
└───────────────────────────────────────────────────────────────────────────┘
```

Figure 8. Current Registers window, as opened with REGS64

You can modify the contents of the PSW or registers whenever they are displayed by overtyping the current information with the new value.

Disassembly window

Open a Disassembly window with the DISASM command or the OPEN DISASM command. Close it by issuing the DISASM command without an address or by issuing the CLOSE command against that window. Many Disassembly windows can be open. Each Disassembly window can display storage and source at a different address.

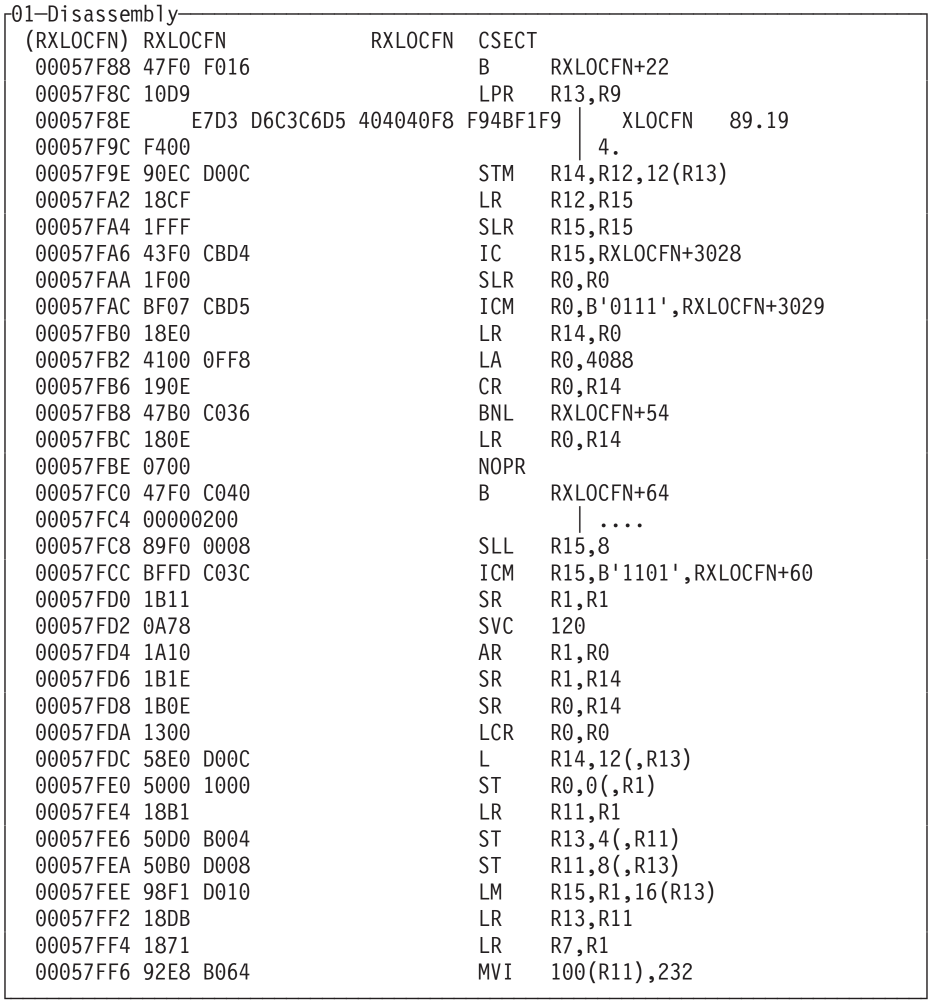
In a Disassembly window, instructions are shown disassembled to their approximate assembler syntax. You can change the contents of the locations displayed by overtyping the hex values shown on the left side of the screen. When storage that cannot be disassembled is displayed, up to 48 bytes (three rows) are displayed in dump format in both hexadecimal and character form. The character form is EBCDIC unless the ASCII option is ON. Modify this storage by overtyping either the hexadecimal or the character portions. Characters overtyped in the character portion are EBCDIC unless the ASCII option is ON. If you have specified the PATH or PATHFILE options, a column of numbers is displayed at the right side of the screen. These numbers are execution counts for the instructions they are next to.

To change the location counter such that it reflects the original location counter of the CSECT as it appears in the assembler listing, use the OFFSET command to set the offset to the starting location of the CSECT within the modules. It may be necessary to set the OFFSET option on (use the OPTIONS command to review the current options).

CSECT names and external symbols are shown intensified (or in the heading color) and the next instruction to be executed is shown intensified (or in the message color). If a breakpoint was set for an instruction, then its address (at the left side of the screen) is shown intensified (or in the heading color). If a watchpoint was set for an instruction, then its address is shown intensified (or in the heading color) and is preceded by a 'w'. If an instruction starts a subroutine that is being skipped, then its address is shown intensified (or in the heading color) and is preceded by an 's'.

Branch instructions are disassembled to their appropriate extended mnemonics unless the NOBCX option is specified.

All open Disassembly windows share any space on the screen not occupied by the Additional Floating-Point Registers window, the Current Registers window, the Old Registers window, the Options window, and the Target Status window with any open Dump windows, and LSM Information windows. Use the SIZE and MOVE commands to change the size and location of any open window.



```
┌01─Disassembly──────────────────────────────────────────────────────────────┐
│ (RXLOCFN) RXLOCFN            RXLOCFN  CSECT                                 │
│  00057F88 47F0 F016                   B     RXLOCFN+22                      │
│  00057F8C 10D9                        LPR   R13,R9                          │
│  00057F8E     E7D3 D6C3C6D5 404040F8 F94BF1F9 │   XLOCFN   89.19            │
│  00057F9C F400                                │ 4.                          │
│  00057F9E 90EC D00C                   STM   R14,R12,12(R13)                 │
│  00057FA2 18CF                        LR    R12,R15                         │
│  00057FA4 1FFF                        SLR   R15,R15                         │
│  00057FA6 43F0 CBD4                   IC    R15,RXLOCFN+3028                │
│  00057FAA 1F00                        SLR   R0,R0                           │
│  00057FAC BF07 CBD5                   ICM   R0,B'0111',RXLOCFN+3029         │
│  00057FB0 18E0                        LR    R14,R0                          │
│  00057FB2 4100 0FF8                   LA    R0,4088                         │
│  00057FB6 190E                        CR    R0,R14                          │
│  00057FB8 47B0 C036                   BNL   RXLOCFN+54                      │
│  00057FBC 180E                        LR    R0,R14                          │
│  00057FBE 0700                        NOPR                                  │
│  00057FC0 47F0 C040                   B     RXLOCFN+64                      │
│  00057FC4 00000200                              │ ....                      │
│  00057FC8 89F0 0008                   SLL   R15,8                           │
│  00057FCC BFFD C03C                   ICM   R15,B'1101',RXLOCFN+60          │
│  00057FD0 1B11                        SR    R1,R1                           │
│  00057FD2 0A78                        SVC   120                             │
│  00057FD4 1A10                        AR    R1,R0                           │
│  00057FD6 1B1E                        SR    R1,R14                          │
│  00057FD8 1B0E                        SR    R0,R14                          │
│  00057FDA 1300                        LCR   R0,R0                           │
│  00057FDC 58E0 D00C                   L     R14,12(,R13)                    │
│  00057FE0 5000 1000                   ST    R0,0(,R1)                       │
│  00057FE4 18B1                        LR    R11,R1                          │
│  00057FE6 50D0 B004                   ST    R13,4(,R11)                     │
│  00057FEA 50B0 D008                   ST    R11,8(,R13)                     │
│  00057FEE 98F1 D010                   LM    R15,R1,16(R13)                  │
│  00057FF2 18DB                        LR    R13,R11                         │
│  00057FF4 1871                        LR    R7,R1                           │
│  00057FF6 92E8 B064                   MVI   100(R11),232                    │
└────────────────────────────────────────────────────────────────────────────┘
```

Figure 9. Disassembly window

Dump window

Open a Dump window by issuing the DUMP command or the OPEN DUMP command. Close it by issuing the DUMP command without an address or by issuing the CLOSE command against that window. Many Dump windows can be open. Each Dump window can display storage at a different address. The Dump window provides either a symbolic or an unformatted dump of storage. The dump format is toggled by the DUMPMODE command.

If a non-zero ALET was provided for a Dump window, then that window displays storage from the dataspace identified by the ALET. The storage in the dataspace is dumped in the unformatted style regardless of the current dump mode selected. The ALET specified is displayed on the first row of the Dump window.

You specify an ALET for a Dump window with the SET ALET command, by using an access register in an expression, or by placing the cursor in an access register in the Current Registers window or Old Registers window.

All open Dump windows share any space on the screen not occupied by the Additional Floating-Point Registers window, the Current Registers window, the Old Registers window, the Options window, and the Target Status window with any open Disassembly windows, and LSM Information windows. Use the SIZE and MOVE commands to change the size and location of any open window.

The symbolic format shows data areas with their symbolic names indicated. In addition, the symbolic dump format shows only the memory area that applies.

For example, if a variable is a halfword that is not aligned on a fullword boundary, IDF uses two lines to show it. The first line shows the CSECT and name of the variable. The second line shows the content of the variable in both hexadecimal and character formats. The character format is EBCDIC unless the ASCII option is ON. Since the variable is not aligned on a fullword boundary, the first two bytes of the word in which the variable is found are not shown, and the two bytes that are shown are shifted right on the screen by that amount.

The advantage of this dump format is that you see the variable and its limits, without hunting for them. The disadvantage is that less is shown on each screen, since a line is taken for each new variable's name.

```
┌01─Storage Dump ─────────────────────────────────────────────────────────────┐
│ (EXAMPLE1) EXAMPLE1                                                          │
│  00010408 47F0F0E8 D3898385 95A28584 40D481A3 │ å00YLicensed Mat │          │
│  00010418 85998981 93A24060 40D79996 978599A3 │ erials - Propert │          │
│  00010428 A8409686 40C9C2D4 40C5E7C1 D4D7D3C5 │ y of IBM EXAMPLE │          │
│  00010438 F1404DC3 5D40C396 97A89989 8788A340 │ 1 (C) Copyright  │          │
│  00010448 C9C2D440 C3D6D9D7 40F1F9F9 F56BF2F0 │ IBM CORP 1995,20 │          │
│  00010458 F0F44B40 C1939340 D9898788 A3A240D9 │ 04. All Rights R │          │
│  00010468 85A28599 A585844B 40E4E240 C796A585 │ eserved. US Gove │          │
│  00010478 99959485 95A340E4 A28599A2 40D985A2 │ rnment Users Res │          │
│  00010488 A3998983 A3858440 D9898788 A3A24060 │ tricted Rights - │          │
│  00010498 40E4A285 6B4084A4 97938983 81A38996 │  Use, duplicatio │          │
│  000104A8 95409699 408489A2 839396A2 A4998540 │ n or disclosure  │          │
│  000104B8 9985A2A3 998983A3 85844082 A840C7E2 │ restricted by GS │          │
│  000104C8 C140C1C4 D740E283 888584A4 938540C3 │ A ADP Schedule C │          │
│  000104D8 9695A399 8183A340 A689A388 40C9C2D4 │ ontract with IBM │          │
│  000104E8 40C39699 974B4000 90ECD00C 18CF41E0 │  Corp. .°Ö}..õ \ │          │
│  000104F8 F11050D0 E00450E0 D00818DE 58F0F158 │ 1.&}\.&\}..úì01ì │          │
│  00010508 05EF58D0 D00450F0 D01098EC D00C07FE │ .Õì}}.&0}.qÖ}..Ú │          │
│  00010518 00000000 00000000 00000000 00000000 │ ................ │          │
│  00010528 00000000 00000000 00000000 00000000 │ ................ │          │
│  00010538 00000000 00000000 00000000 00000000 │ ................ │          │
│  00010548 00000000 00000000 00000000 00000000 │ ................ │          │
│  00010558 00000000 00000000 00010568 00000000 │ ...........Ç.... │          │
│ (EXAMPLE2) EXAMPLE2                                                          │
│  00010568 47F0F016 C5E7C1D4 D7D3C5F2 40F2F0F0 │ å00.EXAMPLE2 200 │          │
│  00010578 F4F0F3F0 F30090EC D00C18CF 41E0F114 │ 40303.°Ö}..õ \1. │          │
│  00010588 50D0E004 50E0D008 18DE17FF 58D0D004 │ &}\.&\}..ú..ì}}. │          │
│  00010598 50F0D010 98ECD00C 07FE90EC D00C18CF │ &0}.qÖ}..Ú°Ö}..õ │          │
│  000105A8 41E0F114 50D0E004 50E0D008 18DE17FF │  \1.&}\.&\}..ú.. │          │
│  000105B8 58D0D004 50F0D010 98ECD00C 07FE90EC │ ì}}.&0}.qÖ}..Ú°Ö │          │
│  000105C8 D00C18CF 41E0F114 50D0E004 50E0D008 │ }..õ \1.&}\.&\}. │          │
│  000105D8 18DE17FF 58D0D004 50F0D010 98ECD00C │ .ú..ì}}.&0}.qÖ}. │          │
│  000105E8 07FE90EC D00C18CF 41E0F114 50D0E004 │ .Ú°Ö}..õ \1.&}\. │          │
│  000105F8 50E0D008 18DE17FF 58D0D004 50F0D010 │ &\}..ú..ì}}.&0}. │          │
│  00010608 98ECD00C 07FE90EC D00C18CF 41E0F114 │ qÖ}..Ú°Ö}..õ \1. │          │
│  00010618 50D0E004 50E0D008 18DE17FF 58D0D004 │ &}\.&\}..ú..ì}}. │          │
│  00010628 50F0D010 98ECD00C 07FE90EC D00C18CF │ &0}.qÖ}..Ú°Ö}..õ │          │
│  00010638 41E0F114 50D0E004 50E0D008 18DE17FF │  \1.&}\.&\}..ú.. │          │
└──────────────────────────────────────────────────────────────────────────────┘
```

Figure 10. Formatted Dump window

The unformatted dump provides the "traditional" memory dump display.

```
┌01─Storage Dump──────────────────────────────────────────────────────────┐
│  00010408 47F0F0E8 D3898385 95A28584 40D481A3 │ å00YLicensed Mat │      │
│  00010418 85998981 93A24060 40D79996 978599A3 │ erials - Propert │      │
│  00010428 A8409686 40C9C2D4 40C5E7C1 D4D7D3C5 │ y of IBM EXAMPLE │      │
│  00010438 F1404DC3 5D40C396 97A89989 8788A340 │ 1 (C) Copyright  │      │
│  00010448 C9C2D440 C3D6D9D7 40F1F9F9 F56BF2F0 │ IBM CORP 1995,20 │      │
│  00010458 F0F44B40 C1939340 D9898788 A3A240D9 │ 04. All Rights R │      │
│  00010468 85A28599 A585844B 40E4E240 C796A585 │ eserved. US Gove │      │
│  00010478 99959485 95A340E4 A28599A2 40D985A2 │ rnment Users Res │      │
│  00010488 A3998983 A3858440 D9898788 A3A24060 │ tricted Rights - │      │
│  00010498 40E4A285 6B4084A4 97938983 81A38996 │  Use, duplicatio │      │
│  000104A8 95409699 408489A2 839396A2 A4998540 │ n or disclosure  │      │
│  000104B8 9985A2A3 998983A3 85844082 A840C7E2 │ restricted by GS │      │
│  000104C8 C140C1C4 D740E283 888584A4 938540C3 │ A ADP Schedule C │      │
│  000104D8 9695A399 8183A340 A689A388 40C9C2D4 │ ontract with IBM │      │
│  000104E8 40C39699 974B4000 90ECD00C 18CF41E0 │  Corp. .°Ö}..õ \ │      │
│  000104F8 F11050D0 E00450E0 D00818DE 58F0F158 │ 1.&}\.&\}..úì01ì │      │
│  00010508 05EF58D0 D00450F0 D01098EC D00C07FE │ .Õì}}.&0}.qÖ}..Ú │      │
│  00010518 00000000 00000000 00000000 00000000 │ ................ │      │
│  00010528 00000000 00000000 00000000 00000000 │ ................ │      │
│  00010538 00000000 00000000 00000000 00000000 │ ................ │      │
│  00010548 00000000 00000000 00000000 00000000 │ ................ │      │
│  00010558 00000000 00000000 00010568 00000000 │ ...........Ç.... │      │
│  00010568 47F0F016 C5E7C1D4 D7D3C5F2 40F2F0F0 │ å00.EXAMPLE2 200 │      │
│  00010578 F4F0F3F0 F30090EC D00C18CF 41E0F114 │ 40303.°Ö}..õ \1. │      │
│  00010588 50D0E004 50E0D008 18DE17FF 58D0D004 │ &}\.&\}..ú..ì}}. │      │
│  00010598 50F0D010 98ECD00C 07FE90EC D00C18CF │ &0}.qÖ}..Ú°Ö}..õ │      │
│  000105A8 41E0F114 50D0E004 50E0D008 18DE17FF │  \1.&}\.&\}..ú.. │      │
│  000105B8 58D0D004 50F0D010 98ECD00C 07FE90EC │ ì}}.&0}.qÖ}..Ú°Ö │      │
│  000105C8 D00C18CF 41E0F114 50D0E004 50E0D008 │ }..õ \1.&}\.&\}. │      │
│  000105D8 18DE17FF 58D0D004 50F0D010 98ECD00C │ .ú..ì}}.&0}.qÖ}. │      │
│  000105E8 07FE90EC D00C18CF 41E0F114 50D0E004 │ .Ú°Ö}..õ \1.&}\. │      │
│  000105F8 50E0D008 18DE17FF 58D0D004 50F0D010 │ &\}..ú..ì}}.&0}. │      │
│  00010608 98ECD00C 07FE90EC D00C18CF 41E0F114 │ qÖ}..Ú°Ö}..õ \1. │      │
│  00010618 50D0E004 50E0D008 18DE17FF 58D0D004 │ &}\.&\}..ú..ì}}. │      │
│  00010628 50F0D010 98ECD00C 07FE90EC D00C18CF │ &0}.qÖ}..Ú°Ö}..õ │      │
│  00010638 41E0F114 50D0E004 50E0D008 18DE17FF │  \1.&}\.&\}..ú.. │      │
│  00010648 58D0D004 50F0D010 98ECD00C 07FE90EC │ ì}}.&0}.qÖ}..Ú°Ö │      │
│  00010658 D00C18CF 41E0F114 50D0E004 50E0D008 │ }..õ \1.&}\.&\}. │      │
└─────────────────────────────────────────────────────────────────────────┘
```

Figure 11. Unformatted Dump window

Regardless of the dump format, you can modify memory by overtyping the data shown, in either the hexadecimal or the character portions of the display. Characters overtyped in the character portion are EBCDIC unless the ASCII option is ON.

Entry Point Names window

Open an Entry Point Names window by issuing the EPNAMES command. Close it by issuing the EPNAMES command.

The Entry Point Names window has a fixed size. Use the PREVIOUS and NEXT commands or PF keys to scroll to the next or previous page, provided the cursor is in the Entry Point Names window when the command is issued or the PF key pressed. The window lists the program name, entry point name, load address and long entry point name for each entry point in the module being debugged.

```
┌03─Entry point name────────────────────────────────────────────More:+-─┐
│  Program TESTIDF   Entry short name TESTIDF  Address 00018EF8          │
│Long name TESTIDF                                                       │
└────────────────────────────────────────────────────────────────────────┘
```

Figure 12. Entry Point Names window

You can change the short entry point name by overtyping it.

LSM Information window

Open an LSM Information window after IDF Language extract data is loaded. Open it with an ARRAY, STRUCTURE, or VARIABLE command, a LANGUAGE command that causes IDF to open this window, or the OPEN command. Close it with an ARRAY, STRUCTURE, or the VARIABLE command without any arguments or by issuing the CLOSE command against that window. Many LSM Information windows can be opened. Each LSM Information window can display different information.

All open LSM Information windows share any space on the screen not occupied by the Additional Floating-Point Registers window, the Current Registers window, the Old Registers window, the Options window, and the Target Status window with any open Disassembly windows, and Dump windows. Use the SIZE and MOVE commands to change the size and location of any open window.

The title and contents of an LSM Information window varies depending on the command being used and the IDF Language options and settings.

```
Var: BThing             > This is the b-thing here               <
```

Figure 13. LSM Information window, with VARIABLE command output. In this case, the variable attributes are hidden due to the COMPACT ON (default) Language option.

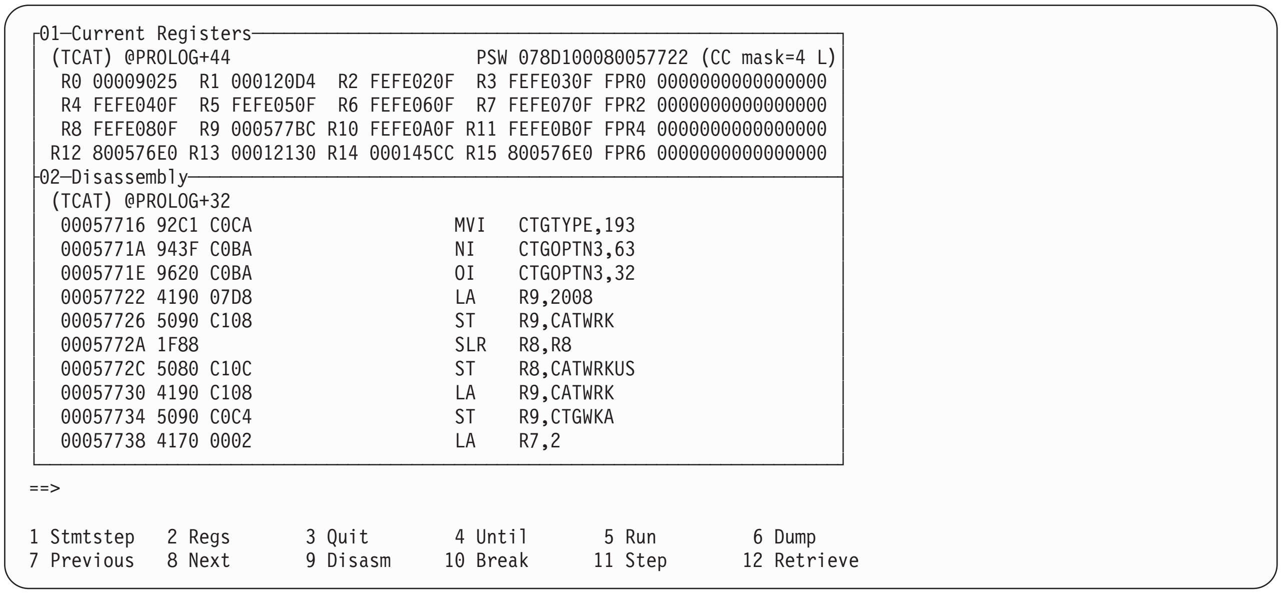
```
┌01─Language Options──────────────────────────────────────────────────────────┐
  Show: Both, Stmt
  Dcls: ON    Comments: ON       Macs: ON   Generated: ON      Nocode: OFF
  Space: OFF     Brief: OFF    Compact: OFF
  Char: EBCDIC   Fixed: Decimal  Float: Std        Bit: Bit
  Enum: Decimal Packed: Decimal  Zoned: Decimal
  VAR checking: Bounds: ON    Negative: ON   Substring: ON    Optimize: ON
  Audit: OFF
  SAREGS: OFF  SALIMIT: 100
  Major: OFF     PadID: OFF     Detail: MIN,3
  Nest: 0       Scroll: MAX
  Debug: OFF
  Stem: LSM.                  EXLIMIT: 20000
  XPATH: ASMLANGX
└─────────────────────────────────────────────────────────────────────────────┘
```

Figure 14. LSM Information window, with LANGUAGE OPTIONS command output

Minimized Windows Viewer

The MINIMIZE command "shrinks" a window, and so frees up space on the display screen. The first MINIMIZE command opens the Minimized Windows Viewer below the Command window (command line and PF Keys). An entry representing the minimized window is placed in the Minimized Windows Viewer. You can select the window to be minimized with the cursor.

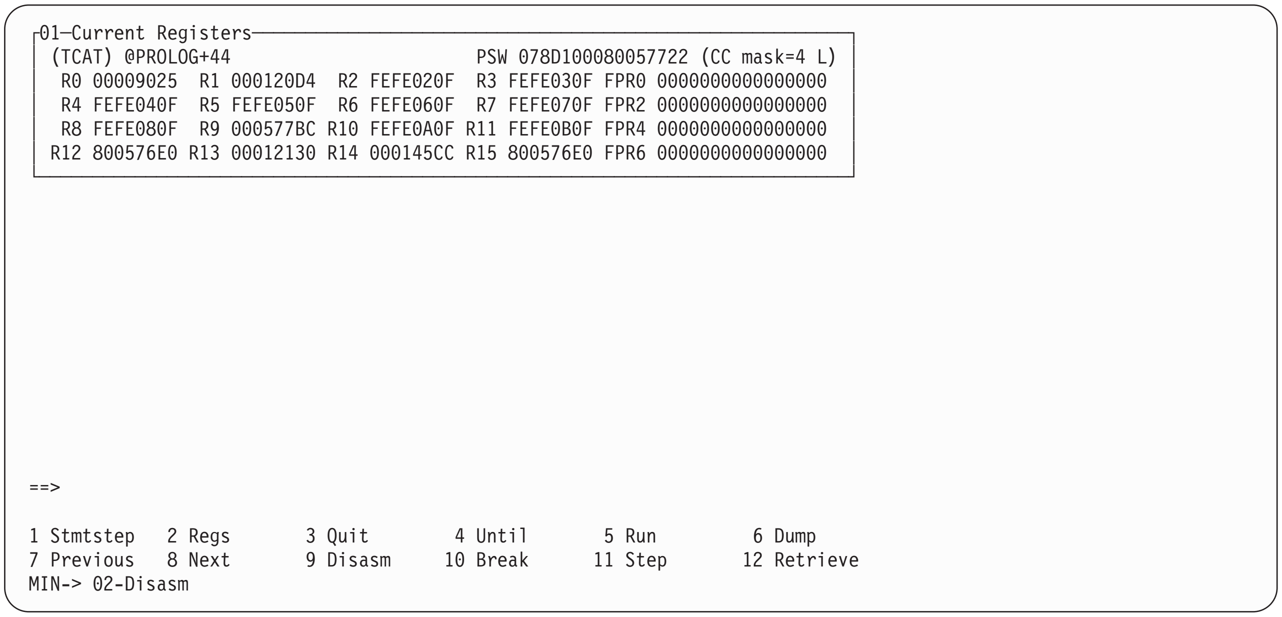
The MAXIMIZE command restores a minimized window to its previous position on the display screen. When the last minimized window is maximized, the Minimized Windows Viewer closes. You can select the window to be maximized with the cursor.



```
┌01─Current Registers──────────────────────────────────────────────────────┐
│ (TCAT) @PROLOG+44                       PSW 078D100080057722 (CC mask=4 L)│
│  R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000 │
│  R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000 │
│  R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000 │
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
├02─Disassembly────────────────────────────────────────────────────────────┤
│ (TCAT) @PROLOG+32                                                         │
│  00057716 92C1 C0CA                   MVI   CTGTYPE,193                   │
│  0005771A 943F C0BA                   NI    CTGOPTN3,63                   │
│  0005771E 9620 C0BA                   OI    CTGOPTN3,32                   │
│  00057722 4190 07D8                   LA    R9,2008                       │
│  00057726 5090 C108                   ST    R9,CATWRK                     │
│  0005772A 1F88                        SLR   R8,R8                         │
│  0005772C 5080 C10C                   ST    R8,CATWRKUS                   │
│  00057730 4190 C108                   LA    R9,CATWRK                     │
│  00057734 5090 C0C4                   ST    R9,CTGWKA                     │
│  00057738 4170 0002                   LA    R7,2                          │
└───────────────────────────────────────────────────────────────────────────┘
==>

1 Stmtstep   2 Regs       3 Quit       4 Until       5 Run         6 Dump
7 Previous   8 Next       9 Disasm    10 Break      11 Step       12 Retrieve
```

Figure 15. A sample screen before any windows are minimized



```
┌01─Current Registers──────────────────────────────────────────────────────┐
│ (TCAT) @PROLOG+44                       PSW 078D100080057722 (CC mask=4 L)│
│  R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000 │
│  R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000 │
│  R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000 │
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
└───────────────────────────────────────────────────────────────────────────┘










==>

1 Stmtstep   2 Regs       3 Quit       4 Until       5 Run         6 Dump
7 Previous   8 Next       9 Disasm    10 Break      11 Step       12 Retrieve
MIN-> 02-Disasm
```

Figure 16. The same screen after the Disassembly window is minimized

Options window

Open an Options window with the OPTIONS command. Close it by issuing the OPTIONS command again, or by issuing the CLOSE command against that window.

The Options window has a fixed size. Use the PREVIOUS and NEXT commands or PF keys to scroll to the next or previous page of settings, provided the cursor is in the Options window when the command is issued or PF key pressed.

```
┌04─Debugger Options─────────────────────────────────────────────More: + - ┐
│APROGMSG: ON      ASCii: OFF  AUTOLoad: ON   AUTOSize: ON       BCX: ON   │
│ CKSubcm: OFF    CMDLog: OFF   CMPExit: OFF   COMmand: OFF      DMS0: OFF │
│  DSECts: ON   EXItexec: OFF  FASTPath: OFF  FULLQual: OFF   HEXDisp: ON  │
│HEXInput: ON   IMPMacro: ON     INVPsw: OFF  MACROLog: OFF    MODMap: ON  │
└──────────────────────────────────────────────────────────────────────────┘
```

Figure 17. Options window

Some IDF options settings cannot be modified after IDF has completed initialization. For these, the option setting is shown in the same color as the option name.

Modify the remaining options by overtyping the current setting with an appropriate options keyword.

Old Registers window

Open the Old Registers window with the OREGS command. Close it by issuing the OREGS command again or by issuing the CLOSE command against that window.

```
┌01─Old Registers─────────────────────────────────────────────────────────┐
│(TCAT) @PROLOG+82                      PSW 078D200080057748 (CC mask=2 H)│
│00057748 1836                      LR    R3,R6                           │
│ R0 00009025  R1 000120D4  R2 00057FC0  R3 FEFE030F FPR0 0000000000000000│
│ R4 FEFE040F  R5 FEFE050F  R6 00000001  R7 00000002 FPR2 0000000000000000│
│ R8 00000000  R9 000577E8 R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000│
│R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 000000000000000 │
└─────────────────────────────────────────────────────────────────────────┘
```

Figure 18. Old Registers window

If your program has executed one or more instructions, by default, a copy of the PSW, next instruction, and registers as of the last time IDF was in control are also shown (after this these are generally the previous contents of the PSW, general registers, and so on). If the CREGS command or the AREGS command was issued, the control registers or the access registers, as of the last time IDF was in control, are displayed. If your program has not executed any instructions, this window is empty.

Skipped Subroutines window

Open the Skipped Subroutines window by issuing the SKIPSTEP command without providing any arguments. Close the window, by issuing the SKIPSTEP command again without any arguments or issue the CLOSE command against that window. When the Skipped Subroutines window is opened, it is positioned at the top right corner of the screen, overlaying any existing information. It uses as many rows as needed to display the information.

The window lists the subroutines for which single-stepping, statement stepping, or the PATH or FASTPATH options are bypassed. If there are more skipped subroutines than fit on the screen, the NEXT and PREVIOUS commands scroll forward and backward through the list.

```
┌01─Subroutines Skipped───────────────────────────────────────┐
│  00057712 (RXLLSTDD) XCOM                                   │
│  00057A88 (RXLLISTD) RXLLISTD                               │
└─────────────────────────────────────────────────────────────┘
```

Figure 19. Skipped Subroutines window

The Skipped Subroutines window also lets you stop the "skipping" of a subroutine by placing the cursor on the field describing the subroutine in question, and either pressing the ERASE-EOF key or overtyping the field with blanks.

Target Status window

Open the Target Status window with the STATUS command. Close it by issuing the STATUS command again or by issuing the CLOSE command against that window.

```
Program VARMVSXA  Origin 00057920  Entrypoint 80057920
Symbols 66           End 00057FFF   Size(hex) 000006E0   Size(dec) 1760
```

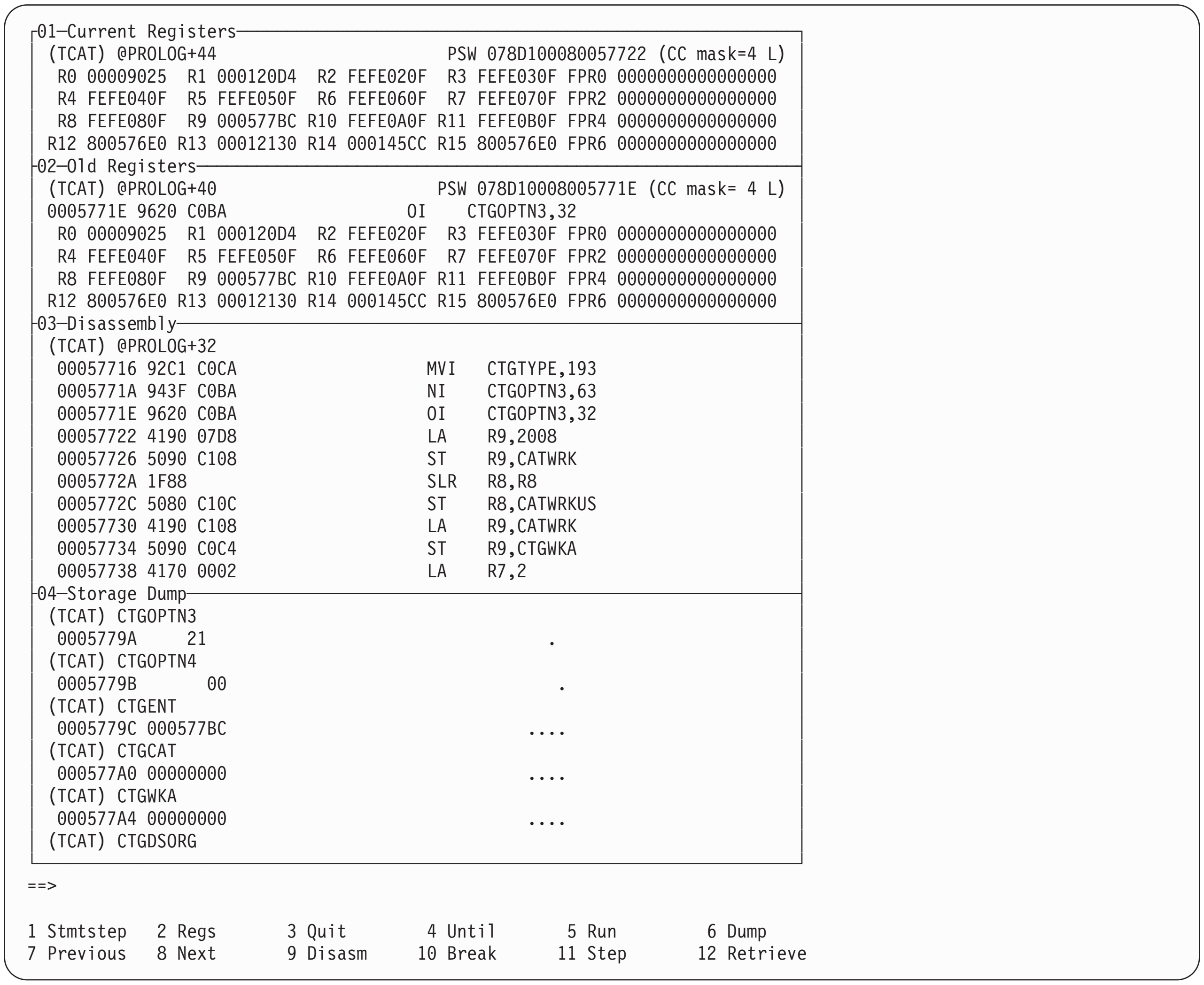
Figure 20. Target Status window

The Target Status window provides information about the programs known to IDF. This includes the number of symbols in the program known to IDF, where the program was loaded in memory, how large it is, and, for the original target, its entrypoint address. The name of the currently qualified module is shown in the message color. The window shows the information about one module at a time. To display information about the other modules known to IDF, place the cursor in the Target Status window and issue a NEXT or PREVIOUS command.

z/OS For multi-segment program objects the window shows information about one segment at a time. For example: PROGA with AMODE24 and AMODE31 classes bound with the SPLIT option will have two entries, one for each segment, with the same Program name.

Some examples of actual screens

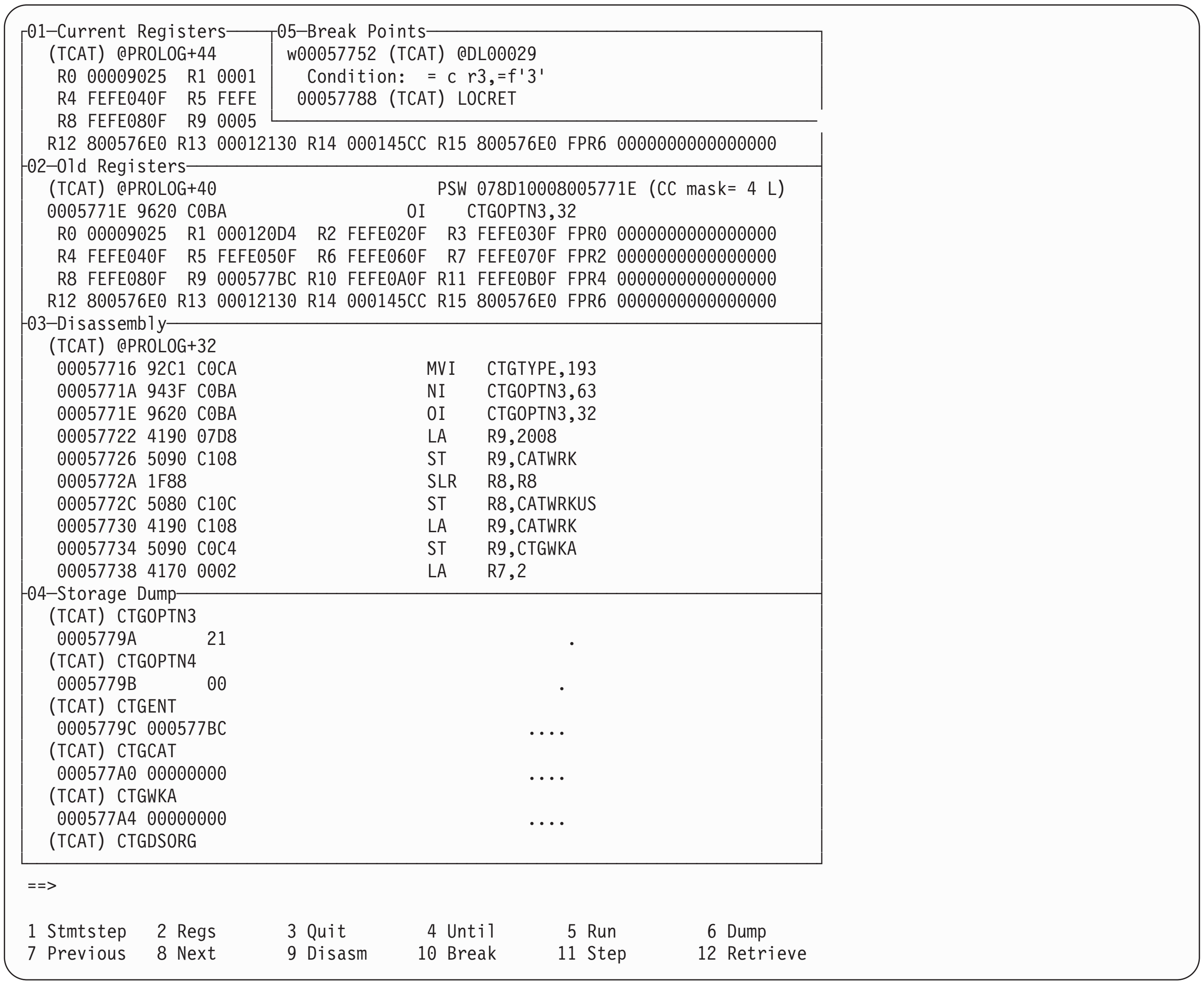
Different windows can be open at the same time in various combinations. The larger the screen the more windows you can have open at one time. This example shows a combination of open windows. It combines the Current Registers window, the Old Registers window, a Disassembly window and a Dump window all on one screen. The example was created on a screen with 43 lines.


```
┌01─Current Registers──────────────────────────────────────────────────────────┐
│ (TCAT) @PROLOG+44                         PSW 078D100080057722 (CC mask=4 L) │
│  R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000    │
│  R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000    │
│  R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000    │
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000    │
├02─Old Registers──────────────────────────────────────────────────────────────┤
│ (TCAT) @PROLOG+40                        PSW 078D10008005771E (CC mask= 4 L) │
│ 0005771E 9620 C0BA                   OI    CTGOPTN3,32                       │
│  R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000    │
│  R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000    │
│  R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000    │
│ R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000    │
├03─Disassembly────────────────────────────────────────────────────────────────┤
│ (TCAT) @PROLOG+32                                                            │
│  00057716 92C1 C0CA                   MVI   CTGTYPE,193                      │
│  0005771A 943F C0BA                   NI    CTGOPTN3,63                      │
│  0005771E 9620 C0BA                   OI    CTGOPTN3,32                      │
│  00057722 4190 07D8                   LA    R9,2008                          │
│  00057726 5090 C108                   ST    R9,CATWRK                        │
│  0005772A 1F88                        SLR   R8,R8                            │
│  0005772C 5080 C10C                   ST    R8,CATWRKUS                      │
│  00057730 4190 C108                   LA    R9,CATWRK                        │
│  00057734 5090 C0C4                   ST    R9,CTGWKA                        │
│  00057738 4170 0002                   LA    R7,2                             │
├04─Storage Dump───────────────────────────────────────────────────────────────┤
│ (TCAT) CTGOPTN3                                                              │
│  0005779A     21                                 .                           │
│ (TCAT) CTGOPTN4                                                              │
│  0005779B       00                                .                          │
│ (TCAT) CTGENT                                                                │
│  0005779C 000577BC                             ....                          │
│ (TCAT) CTGCAT                                                                │
│  000577A0 00000000                             ....                          │
│ (TCAT) CTGWKA                                                                │
│  000577A4 00000000                             ....                          │
│ (TCAT) CTGDSORG                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
==>

1 Stmtstep   2 Regs       3 Quit       4 Until       5 Run         6 Dump
7 Previous   8 Next       9 Disasm    10 Break      11 Step       12 Retrieve
```

Figure 21. An example of several open windows on one screen

If your screen display is like the one shown above, and you open the Break window, the Break window overlays part of the screen as described above and shown in Figure 22 on page 79.



```
┌01─Current Registers─────┬05─Break Points─────────────────────────────────┐
│  (TCAT) @PROLOG+44      │ w00057752 (TCAT) @DL00029                      │
│   R0 00009025  R1 0001  │   Condition:  = c r3,=f'3'                     │
│   R4 FEFE040F  R5 FEFE  │  00057788 (TCAT) LOCRET                        │
│   R8 FEFE080F  R9 0005  └────────────────────────────────────────────────┘
│  R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
├02─Old Registers──────────────────────────────────────────────────────────┤
│  (TCAT) @PROLOG+40                      PSW 078D10008005771E (CC mask= 4 L) │
│  0005771E 9620 C0BA                  OI    CTGOPTN3,32                    │
│   R0 00009025  R1 000120D4  R2 FEFE020F  R3 FEFE030F FPR0 0000000000000000 │
│   R4 FEFE040F  R5 FEFE050F  R6 FEFE060F  R7 FEFE070F FPR2 0000000000000000 │
│   R8 FEFE080F  R9 000577BC R10 FEFE0A0F R11 FEFE0B0F FPR4 0000000000000000 │
│  R12 800576E0 R13 00012130 R14 000145CC R15 800576E0 FPR6 0000000000000000 │
├03─Disassembly────────────────────────────────────────────────────────────┤
│  (TCAT) @PROLOG+32                                                       │
│   00057716 92C1 C0CA                   MVI   CTGTYPE,193                  │
│   0005771A 943F C0BA                   NI    CTGOPTN3,63                  │
│   0005771E 9620 C0BA                   OI    CTGOPTN3,32                  │
│   00057722 4190 07D8                   LA    R9,2008                      │
│   00057726 5090 C108                   ST    R9,CATWRK                    │
│   0005772A 1F88                        SLR   R8,R8                        │
│   0005772C 5080 C10C                   ST    R8,CATWRKUS                  │
│   00057730 4190 C108                   LA    R9,CATWRK                    │
│   00057734 5090 C0C4                   ST    R9,CTGWKA                    │
│   00057738 4170 0002                   LA    R7,2                         │
├04─Storage Dump───────────────────────────────────────────────────────────┤
│  (TCAT) CTGOPTN3                                                         │
│   0005779A       21                              .                       │
│  (TCAT) CTGOPTN4                                                         │
│   0005779B       00                             .                        │
│  (TCAT) CTGENT                                                           │
│   0005779C 000577BC                          ....                        │
│  (TCAT) CTGCAT                                                           │
│   000577A0 00000000                          ....                        │
│  (TCAT) CTGWKA                                                           │
│   000577A4 00000000                          ....                        │
│  (TCAT) CTGDSORG                                                         │
└──────────────────────────────────────────────────────────────────────────┘
 ==>

 1 Stmtstep   2 Regs       3 Quit       4 Until       5 Run         6 Dump
 7 Previous   8 Next       9 Disasm    10 Break      11 Step       12 Retrieve
```

Figure 22. An example of the Break window overlaying other windows

Specifying a window

Many commands affect what is displayed in a particular window. There can be more than one Disassembly window, Dump window, or LSM Information window, open at once. IDF needs to be told which one is the target of commands like DISASM, DUMP, FOLLOW, LANGUAGE, SEARCH, and SET ALET. Similarly, with the CLOSE, MOVE, NEXT, PREVIOUS, and SIZE commands IDF needs to be told which window to use.

To tell IDF which window to use, you can use a Window Specification in the appropriate command. The Window Specification is an equal sign followed by the window's window number, in the form "=n". The Window Specification, if present, must precede any operands of the command. For example:

```
DISASM =2 address
FOLLOW =3 R9
```

You can also indicate the window IDF should use by placing the cursor in the window. As the cursor can also specify the argument of a command, a Window Specification, if present, is used in preference to the position of the cursor to specify the window to be used.

If you neither use a Window Specification nor place the cursor in a window of the appropriate type (Disassembly window for DISASM command, and so on), then:

1. The DISASM and LANGUAGE commands use the first open Disassembly window.
2. The DUMP and FOLLOW commands use the first open Dump window.
3. The ARRAY, STRUCTURE, and VARIABLE commands use the first open LSM Information window.
4. The NEXT and PREVIOUS commands scroll all the open windows.
5. The SIZE and MOVE commands need the window type as an extra argument.
6. The SEARCH command starts searching from the start address of the first open Disassembly window or Dump window.

PF keys

The default PF key settings provide the functions anticipated to be most useful.

As distributed on CMS and TSO, the ENTER key is set to the COMMAND command. This means that you can invoke any IDF command by typing it on the command line and pressing ENTER, instead of invoking it by pressing a PF key.

You can redefine the PF keys by issuing a SET command on the command line, by issuing SET commands in your PROFILE (the usual method), or by writing a macro to reset them (occasionally useful). The SET PFK command is described in “PFK” on page 160.

The ENTER key and PF1 through PF24 can be set to any IDF command or to any IDF macro.

By default, PF13 through PF24 are undefined and undisplayed. To display use “PFKDISP” on page 160. Any of those keys that remain undefined are mapped into PF1 through PF12. This makes it easier to use some terminals.

Command record and playback features

IDF provides a command level record and playback facility. IDF records commands to a command log. These commands can then be replayed. They can all be replayed at the start of a debugging session, or else they can be recalled one by one, and then executed, modified and executed, or skipped.

To record a debugging session on the command level, specify or set the CMDLOG option.

The location of the command log is:

- On CMS, the file "ASM CMDLOG *fm*", where *fm* is specified by the MODE option (and defaults to file mode A1).
- On z/OS the data set defined by the CMDLOG DD name.
- On z/VSE, the data set defined by the CMDLOGO DLBL name.

Turn logging off with the SET OPTION CMDLOG command.

You can specify command arguments with the cursor position, and you can overtype data on the screen. IDF logs all operations in a way that lets you repeat them later with some modification to the file.

To play back all or part of a session recorded at command level, use the RLOG command, or specify the RLOG option at invocation.

Address expressions

You can specify the addresses you provide to IDF either by the cursor placement (for information about this see “Arguments and cursor positioning” on page 83) or by typing an expression on the command line before pressing a PF key.

An IDF expression is made up of terms. If an expression consists of two or more terms, a plus (+) or minus (-) sign (operator) indicates that the fully resolved values of the terms should be added or subtracted.

A term can consist of a program symbol, a statement number, a hex constant, a decimal constant, or a character constant that is one character in length. Program symbols are of the form "(module.csect) symbol". If supported by the LSM, they may also be of the form "(module.csect) STMT#nnnnn". The following are examples of basic terms:

```
LOOP
STMT#5
X'2005E'
C'b'
247
F'235'
```

Source statement numbers can be used as symbolic names. They are specified in the form "STMT#*nnnnn*", where *nnnnn* is the statement number involved.

Numeric values that are input without an explicit hexadecimal (X'999') or decimal (F'999') indication, for example "246", are interpreted by IDF as determined by the current base setting. The default base is decimal. It can be redefined to hexadecimal with HEXINPUT ON or SET OPTION ON HEXINPUT. (The arrow beside the command line indicates the default base; an arrow of the form "-->" is used if the default base is decimal, and of the form "==>" is used if the default base is hex.)

Implicitly specified hexadecimal numbers must begin with a numeric digit from 0 to 9. If you attempt to input a hexadecimal number such as A34 implicitly, IDF interprets it as the name of a symbol, not a number. Thus, 0A34 is recognized (assuming the default base is hexadecimal) but A34 is taken as a symbol name.

In some cases you may wish to specify a CSECT name or a module name. For example, if symbol LOOP occurs in two assemblies, you have to specify the CSECT name of the symbol you are referring to. If the symbol is not in the currently QUALIFIED module, you have to specify the module of the symbol you are referring to. Specify the module name and the CSECT name by prefixing the term with them, and enclosing them in parentheses. The module name, when present, precedes the CSECT name and is followed by a period. When a module name is specified, a CSECT name need not be specified. Here are some examples:

```
(TEST)LOOP
(VARMVSXA.VARASM) BTHING
(TCAT.) LOCRET
```

Intersperse blanks as you like, provided they do not interrupt items such as a symbol or hex constant. For example, the following expressions are equivalent to the previous ones:

```
( TEST   )      LOOP
( VARMVSXA  . VARASM   ) BTHING
( TCAT .  ) LOCRET
```

The SET QUALIFY command tells IDF which module's symbols should be searched when a module name is not specified in a symbolic name.

In some cases you may wish to follow a term with a register designator. A register designator consists of the strings 'PSW', 'R0' through 'R15', or 'AR0' through 'AR15', enclosed in parentheses. If 'AR0' through 'AR15' are specified then those commands that can use an ALET use the ALET contained in that access register in addition to the contents of the associated general purpose register. For example, location LOOP in CSECT TEST indexed by the current contents of R4 is specified:

```
(TEST)LOOP(R4)
```

The byte at the location addressed by R4 in the dataspace identified by the ALET in AR4 is specified:

```
    0(AR4)
```

You can combine terms, for example:

```
    (TEST) LOOP+20 (R2)

    24(R2)+0(R3)
```

As another example, to find the difference between two labels in the same CSECT, LOOP, and LOOPEND, you could use:

```
    (TEST)LOOPEND-(TEST)LOOP
```

If you have a translate table, you might want to use an expression with a character term:

```
    TRTABLE+C't'
```

Character terms in addressing expressions are restricted to a single character. If you need to specify an apostrophe, place two apostrophes within the enclosing apostrophes: for example C''''.

Terms and register designators can be followed by indirection operators (`%, :>, &, +>, ?, =>, ->`). If an indirection operator follows a term, IDF uses the contents of the word pointed to by the expression evaluated thus far. Similarly, if an indirection operator follows a register designator, IDF is being told how to interpret the contents of the register. The word or register is treated as:

- A 24-bit address if the `%` or `:>` operators are used.
- A 31-bit address if the `?` or `=>` operators are used.
- A 64-bit address if the `&` or `+>` operators are used.
- The appropriate size (31-bit or 24-bit or 64-bit) depending on the AMODE of the PSW if the `->` operator was used.

Note: If a register designator is not followed by an indirection operator it is interpreted as a 24-bit, 31-bit or 64-bit value depending on the AMODE of the PSW.

You can refer to the current offset value (set with the OFFSET command) by starting an expression with a unary plus or minus. If the expression begins with a unary plus, it is interpreted as the current OFFSET value plus the remainder of the expression. If the expression begins with a unary minus, it is interpreted as the current OFFSET value minus the remainder of the expression. Examples of an expression of this type are:

```
+X'40'
-23
```

If the current offset value is X'20000', the expression +X'40' gives the value X'20040'.

Addresses displayed by IDF

In many situations IDF displays, or returns, addresses in symbolic form. Normally, these "symbolic addresses" are of the form "(module.csect) symbol+offset". If the address is within a code section for which IDF Language extract data was loaded, the symbolic address is of the form:

```
 "(module.csect) STMT#nnnnn+offset".
```

If the address is within the limits of the currently qualified module, the module name and the separating period are normally left out. The FULLQUAL option forces IDF to always include the module name in generated symbolic names.

Source statement numbers can be used as symbolic names. They are specified in the form `"STMT#nnnnn"`, where `nnnnn` is the statement number involved. `nnnnn` can be any value between 1 and 65535 inclusive.

Arguments and cursor positioning

A number of IDF commands accept arguments. For example, the BREAK command accepts an argument, which is the address at which a breakpoint is to be set. This section describes the method used by IDF to determine what argument was provided.

IDF uses "intelligent" cursor sensing logic to let you specify a "verb" by pressing a PF key, and specify the argument by either typing an expression, or "pointing" with the cursor. Instead of typing an expression and a command, you can place the cursor on a display field, then press a PF key. This tells IDF to perform the requested task at the current cursor location.

The following examples show how this works in practice:
- When a register contains the address of an area you want to dump or disassemble, place the cursor in the register and press the DUMP or DISASM PF key.
- If a Disassembly window is open, you can place the cursor in the first input field where the instruction's hexadecimal value is shown, and press the BREAK key to set a breakpoint on that instruction.
- If you have dumped a control block that contains a "link" word, place the cursor in the link word and press the DUMP key. This makes the link word the first one shown. If you press DUMP again, IDF "follows" the link and displays the next control block.

z/VM

- If you have dumped a storage area that is being altered unintentionally, you can place the cursor in the Dump window and issue the ADSTOPS command to set the start of an address modification range, then move the cursor to the end of the area and issue the ADSTOPS command again to set the end of the range shown.

If you find the rules described here difficult to remember, experiment with IDF to see if you can get the "feel" of it. If not, you can always specify arguments by typing them on the command line.

Here is how IDF determines the argument:
1. If the command does not accept an argument, no argument is sought.
2. If an expression is entered on the command line, that expression is evaluated as the supplied argument.
3. If the cursor is on the command line, but no expression is entered there, IDF considers that no argument is supplied.
4. If the cursor is in an unprotected field that is not the command line, IDF attempts to use that cursor position as an indication of the argument that the operator needs, as follows:
   a. If the cursor is in a floating-point register, or any field shown on the Break window, IDF considers that no argument is supplied.
   b. If the cursor is in a general-purpose register, IDF considers the low-order 24 bits, or the low-order 31 bits if in AMODE31, or 64-bits if in AMODE64, of the contents of that register is the supplied argument. If the access registers are displayed, then the DUMP and OPEN DUMP commands use the ALET contained in the access register in addition to the address contained in the associated general-purpose register. This applies to the registers displayed in both the Current Registers window and the Old Registers window.
   c. If the cursor is in the PSW, IDF considers the address part of the PSW is the supplied argument. This applies to the PSW displayed in both the Current Registers window and the Old Registers window.
   d. If the cursor is in a disassembled instruction, the following rules apply:
      1) All commands except DISASM and OPEN DISASM use the address of the halfword the cursor is in.

2) The DISASM and OPEN DISASM commands use the address of the halfword the cursor is in, *unless* the field the cursor is in is both the first field disassembled and a branch instruction, when the commands use the effective address of the branch instruction.

e. If the cursor is in the protected portion of a disassemble line, the starting address of the instruction disassembled is used.

f. If the cursor is in a dump field, the following rules apply:

   1) All commands except DUMP and OPEN DUMP use the address of the beginning of the hexadecimal field the cursor is in, or the exact address of the character the cursor is on if it is in the character portion of the display.

   2) The DUMP and OPEN DUMP commands use the address of the beginning of the hexadecimal field the cursor is in, *unless* the field is both a fullword field and the first field in the dump display, in which case the commands use the low-order 24 bits, the low-order 31 bits if in AMODE31, or the first two words if in AMODE64, of the *contents* of the field.

g. If the cursor is in the protected portion of a dump line, the starting address of the dump line is used.

Chapter 9. Source-level debug additional capabilities

IDF provides a significant debug capability for application programs at the disassembly (object code) level without any more preparation. For trivial debug situations, this is often enough.

Programmers write programs at the source level. It follows that when the reason for a problem is not immediately visible, it is advantageous to also debug at the source level. The addition of the ASMLANGX step to the program development process means that IDF Language extract data is available for the program compile units. When this data is loaded using the IDF LOAD LANGUAGE or STEP or STMTSTEP commands, you can perform these extra functions:

-
  - Selectively enable and disable the display of variable declaration statements with SHOW DCLS and HIDE DCLS commands.
  - Selectively enable and disable the display of block comments with SHOW COMMENTS and HIDE COMMENTS commands.
  - Selectively enable and disable the display of macro definitions and macro expansions with SHOW MACROS and HIDE MACROS commands.
- Perform source text searches
  - You can find and display the desired area of the source code.
  - The search can be up or down.
  - Both LOCATE (like the XEDIT Locate command) and FIND (like the ISPF Editor Find command) commands are supported.
- Set breakpoints on program statements
  - The breakpoint can be at the start of, or within, any program statement.
- Single-step your program
  - You can use the STMTSTEP command to single-step at the program statement level.
  - You can use the STEP command to single-step at the program instruction level.
- Program variable data display (with optional typeover alteration)
  - Supported for any program variable that is "known" in the currently executing program block.
  - There are many types of variables. These are displayed with different IDF Language Support commands:
    - To display simple variables, the VARIABLE command is used.
    - To display structures and their components, the STRUCTURE command is used.
    - To display array elements, the ARRAY command is used.
    - To display variables type attributes, the TYPE command is used.
    - To display the names of variables for which a pointer (or locator) variable is a valid base address, the PLOCATES command is used.

      For more details, see “Displaying and changing items” on page 86.
  - The various forms of program variable display support complex expressions. See “Variable expressions” on page 87 for more details.
- Program variable name display
  - The names of the program variables matching a particular pattern can be displayed.
  - The names of all program variables can be displayed.

  See “Displaying variable names” on page 90 for more details.
- Program caller hierarchy display
  - For each generation:

- For those programs for which IDF Language information is available, the program module, code section, source statement number, and source statement number are shown
- For those programs without IDF Language information, the program module, code section, and offset from the start of this code section are shown.

See "Displaying CALLERS" on page 90 for more details.

Controlling single-stepping your program

If STOPSTMT OFF is in effect, the ASMLANGX data is not automatically loaded while the STMTSTEP process is in control. If STOPSTMT OFF is in effect when STMTSTEP is at a code location for which no extract data is loaded, then language extract AUTOLOAD is not performed. At the start of the session with no LANGUAGE LOAD done, when the STMTSTEP is issued - it will load the extract data for the current CSECT and then go into STMTSTEP process mode. Once in this process no further loads will occur. This allows for all CSECTS to have language extract data available at all times. When running a debug session in which only particular CSECTs are to be examined by STMTSTEPping, then set STOPSTMT OFF and LANGUAGE LOAD the CSECTs required prior to starting - there is no need to manipulate the members available in the ASMLANGX data set.

With STOPSTMT OFF, a STMTSTEP on a call to another CSECT will still perform single-stepping on each instruction in the called routine (without updating the display) until an instruction for which language extract data was loaded is encountered. This may involve many thousands of instructions before such an instruction is encountered, and may give the appearance that the program is in a loop because the display is locked during this process.

If the STEP command is issued by a macro, it does not take effect until the macro exits. The MSTEP command can be used to execute target instructions before returning control to the invoking macro.

If you have subroutines within the program which you do not want to step through, use the SKIPSTEP command. The SKIPSTEP command causes IDF to skip stepping when it comes to a subroutine call to a subroutine that was added to the list of subroutines being skipped. For the purposes of stepping, the skipped subroutine is treated as one instruction, the subroutine call instruction itself. If a breakpoint or a watchpoint whose condition is true is placed within the execution path of the subroutine being skipped, execution stops at that breakpoint or watchpoint.

Displaying and changing items

Items are displayed using these commands:

Table 1. Commands Displaying items

Item	Command
variables	VARIABLE
structures	STRUCTURE
array elements	ARRAY
type attributes for a variable	TYPE
pointer locates information for a variable	PLOCATES

The VARIABLE command lets you look at successive elements of an array. However, the need to manually update the array indices to view the desired array element becomes tiresome.

If the array is a structure component, use the STRUCTURE command to scroll through all elements of the array.

If the array is actually a substructure with multiple components you must spend the time to scroll past the components which are not relevant to the problem at hand. The ARRAY command helps in these situations.

When you use the ARRAY command, you select the initial element to be displayed using the same syntax as the VARIABLE command. The significant differences are:

- When the LSM Information window is scrolled forwards past the end of the current variable information display, the next element in the array is displayed.
- When the LSM Information window is scrolled backwards past the beginning of the current variable information display, the previous element in the array is displayed.

The display of the information about items persists until you:

- Issue the same command, but without item arguments.
- Issue a CLOSE command against the window.
- Update the information in the window, with a command such as VARIABLE, ARRAY, CALLERS, EVALUATES, PLOCATES, LANGUAGE STATUS or MAP.
- The target program completes execution.
- The target program execution progresses beyond the item's defined scope.

You can change the data displayed by a VARIABLE, STRUCTURE, UNION or ARRAY command by overtyping it.

Variable expressions

Variable expressions are supported for those programs for which the program that generates the extraction file (ASMLANGX) has made variable information available. Variables are displayed in PL/I-like format, with appropriate extensions where needed.

Variable expressions cannot be substituted for Address expressions in commands that expect an address to be supplied. For example, 'BREAK variable-expression' does not evaluate the variable-expression as an address. However, an IDF macro can be written that uses EXTRACT VLOC to determine the address of a variable which is then used as an operand on any IDF command:

```
/* REXX Variable Breakpoint                      */
/* Supply the variable name to be set breakpoint at */
ARG vbn
  IF vbn ¬= '' THEN DO
    'EXTRACT VLOC ' vbn
    IF LSM.0 = 1 THEN DO
      PARSE VAR LSM.1 . varaddr .
      'BREAK 'varaddr
    END
  END
```

Variable scope

The source statement which corresponds to the current instruction address in the PSW is used to determine the current program scope.

Only variables that are active within the current program scope can be accessed.

Variable names

Variable names are 1 to 255 characters.

Assembler variable names are translated to upper case as part of command processing.

Simple variables

Simple variables are defined by name only.

Aggregate variables

Structures are examples of aggregate variables, where a number of variables are associated together in a hierarchical manner.

The major component is the term which is used to define the variable which "owns" the aggregate. The variables within the aggregate are variously known as components, fields, and members depending on the source language.

Dot qualification

A simple variable can have the same name as a variable which is within a structure. In this case, *dot qualification* is used to distinguish the variables.

```
Dcl
  minor     Char(1),   /* Simple variable     */

  1 major   Char(3),   /* Structure           */
   2 minor   Char(1),
   2 minor2  Char(1),
   2 minor3  Char(1);

var minor         <- "MINOR" is an ambiguous name
var .minor        <- displays simple variable "MINOR"
var .major        <- displays structure major component "MAJOR"
var major.minor   <- displays structure component "MINOR"
```

When a structure is more than 2 levels deep, IDF lets you omit any of the leading names, as long as the name is not ambiguous. If multiple variables within the same structure have the same name, and dot qualification levels are skipped, IDF selects the component which is reached in the fewest number of skipped levels.

Based variables

You can specify based variables by name only, in which case the *implicit* variable basing is used if available. This is:
- The variable basing which was defined when the variable was declared. For some variables, such as parameters, this information is determined during the extraction process (by ASMLANGX).
- The variable basing as redefined by an active assembler USING statement.

Alternatively, you can specify an *explicit* locating expression, making use of the **->** locator operator.

There are limitations in the processing of some Assembler USING statements by ASMLANGX, which in turn limits ASMIDF. These USING statements are:
- Dependent USINGs (labeled and unlabeled).
- USINGs coded with an end value.
- USINGs that do not cover the start of an area:
  ```
  USING MYAREA+4096,R8   - There is no USING MYAREA,R7
  ```

Array indexing

Information regarding the upper and lower bounds of each array dimension are defined within the IDF Language extract file.

Array indexes are specified from most-significant to least-significant in a left to right direction. Array elements are mapped within the array in *row major* order.

If too many array indexes are specified, a warning message is issued and the extra indexes are ignored.

If insufficient array indexes are specified, a warning message is issued and the remaining indexes are set to the low bound of the corresponding array dimension.

The indexing expression can be:
- PL/I-like-style, where the index values are enclosed within parentheses
- C-style, where the index values are enclosed within either square brackets ("[" and "]"), or the equivalent trigraphs ("??(" and "??)")

The style used is only significant if a substring specification follows the array index specification.

Variables and built-in function results can specify array index specifications.

Array index values must normally be within the bounds defined by the variable declaration. Suppress this check with the CHECK BOUNDS OFF command.

Substrings

For Character String and Bit String variables, you can specify that only a portion of the string (from here on known as a "substring") is processed by a command.

There are three basic forms of substring specifications:
1. single element
   - varname(element-index)
2. from-to notation
   - varname(from-element-index:to-element-index)
3. length notation
   - varname(from-element-index::element count)

The definition of the first element of a string varies from language to language. IDF lets you specify the format to be used:

PL/I-like-style
- array index or substring expression is enclosed within parentheses
- first element in the string is element **1**

C-style
- array index or substring expression is enclosed within either square brackets ("[" and "]"), or the equivalent trigraphs ("??(" and "??)")
- first element in the string is element **0**

Variables and built-in function results can be used to specify substring specifications.

Substring expressions follow any array index specifications, with a delimiting comma.

Substring ranges must normally be within the limits defined by the variable declaration. Suppress this check with the CHECK SUBSTRING OFF command.

Examples

```
var stuff
str addr(x'20000')->struct
var addr(12(R2))->ptr->stuff(2)
var ptr->ptr2->stuff
var ptr(3)->ptr->stuff
arr stuff(-5)
arr stuff(1:10)
arr stuff(1::25)
var var1;var2
var var1 ;ptr->stuff
var stuff(1:10) ; stuff(30:40)
var chrarray(15,1:10);chrarray(15,1::10)
var chrarray[15,0:9];chrarray[15,0::10]
```

Displaying variable names

The NAMES command provides a list of the variables that are eligible for display. This list is potentially quite large, so you can provide name patterns. Each name pattern acts as a filter; only names that match at least one name pattern are displayed.

For more information about NAMES, see “NAMES” on page 154.

Displaying CALLERS

Calling hierarchy information display is supported for all programs which use the standard OS/VS "Type-1" convention for Save Area area register usage and chaining.

For more information about displaying the calling hierarchy information for one or more generations in the program caller hierarchy, see “CALLERS” on page 103.

Source level support

If you need source level support for any dynamically loaded programs then you need to get IDF to load the required ASMLANGX files. There are two ways to do this:

- Use the STMTSTEP command (see “STMTSTEP” on page 186). For this to work correctly, the CSECT name must match the file name of the extract file. So if you are in a section named CODE then the extract file name also has to be CODE for the automatic load to complete successfully.
- Use the LANGUAGE LOAD command (see “LANGUAGE LOAD” on page 129). This command lets you preload the extract files before execution of these sections. It also lets you load extract files where the name of the extract file does not match the section name. This might be useful when you are debugging a module with multiple code sections, but you only have one extract file.

The QUALIFY command (see “QUALIFY” on page 165) lets you change the default module name. This is useful with dynamically loaded programs to simplify the commands.

For example, assume that the initial program is ARROW, and IDF has loaded another program called TARGET. The default module name starts off as ARROW and a command like `DISASM SECTA` searches ARROW for the section names SECTA.

If there is also a section named SECTA in TARGET, to display this section, you have to enter the command `DISASM TARGET.SECTA`.

If you issue `QUAL TARGET` first to change the default section name, then `DISASM SECTA` searches module TARGET.

The qualified name is used whenever a module name is omitted from a command, and the displays remove the qualified name from the window title. Using the previous example, and with `QUAL TARGET` issued, then the DISASM window from DISASM SECTA shows (SECTA), and if there is another window displaying SECTA from the module ARROW, then the title in that window shows (ARROW.SECTA).

Chapter 10. Commands and operating procedures

This chapter describes each IDF command, except for EXTRACT, which is used only in macros. The EXTRACT command is described in Chapter 15, "The EXTRACT command," on page 223.

IDF commands perform the same functions regardless of whether they are invoked through a PF key, the command line (when a PF key set to COMMAND is pressed), or a macro.

If you have not read "Arguments and cursor positioning" on page 83, please do so before you continue. Many of the following command descriptions assume you understand how IDF obtains an address from the cursor position.

The documentation of each command describes a way in which the command will work. The IDF command processor is powerful and flexible, and it is possible that there are alternative undocumented variations to these commands that will also work.

IDF commands cross-reference

Breakpoint Commands	**Page**
ADSTOP	94
ADSTOPS	95
BREAK	100
DBREAK	109
REGSTOPS	169
SET ADSTOP	174
SET BREAK	175
SET REGSTOP	180
TRIGGER LOAD	192
UNTIL	194
WATCH	197
Program Information Commands	
BASE	99
CHECK	105
EPOFFSET	115
LOAD	138
LOCATION	140
MODULE	144
MODULE (type)	144
MODULE BASE	145
MODULE SIZE	146
PROGCHK	163
PROGCK	163
PSWSTEAL	164

Breakpoint Commands	**Page**
QUALIFY	165
SELFNUCX	173
SET SIZE	180
STOKEY	187
STOREMAP	187
VERSION	196
XEDEXIT	199
Program Running Commands	
GOTO	122
MACRO	141
MRUN	149
MSTEP	153
PAUSE	159
PER	160
PSW	163
RUN	171
RUNEXIT	171
SET PSW	179
SKIPSTEP	184
STEP	185
STMTSTEP	186
SUBSET	189
SVC	189
SYMBOL	190

Breakpoint Commands	Page
TASKS	191
VALUE	194
WHERE	199
Program Exit Commands	
ABEND	94
QQUIT	166
QUIT	167
RCQUIT	167
Register Modification Commands	
FPC	121
FPR	121
GPR	122
GPRG	123
GPRH	123
R0-R15	171
Meta Control Commands	
APROGMSG	96
AUDIT	98
DROP GLOBAL	112
DROP MODULE	112
DROP SYMBOLS	113
EXITEXEC	115
GLOBALS	121
GPACK	122
GSTATUS	123
HISTORY	125
ICOUNT	125
KWDSYN	126
LIBE	137
MODE	144
MPACK	149
MSTATUS	152
PRESERVE	162
RESTORE	169
REGS64	168
SALIMIT	172
SAREGS	172
SAVE	172
SET EXITEXEC	176
SET GLOBAL STEM	176
SET GLOBAL TEXT	177

Breakpoint Commands	Page
SET ICOUNT	177
SET OPTION	178
Terminal Control Commands	
ALARM	95
COLORS	106
COMMAND	107
CURSOR	108
LASTMSG	136
MSG	150
MSGID	151
MSGMODE	151
PFK	160
PFKDISP	160
QUIET	166
QUIETLY	166
REFRESH	168
RETRIEVE	169
SET COMMAND	175
SWAP	190
Window Open and Close Commands	
AFPR	95
CALLERS	103
CLOSE	106
DISASM	111
DUMP	113
EPNAMES	114
LANGUAGE +	126
MAP	142
OPEN	156
OPTIONS	157
OREGS	158
PARMS	159
REGS	168
STATUS	185
Window Manipulation Commands	
BACK	98
DOWN	112
DUMPMODE	114
FIND	116
FOLLOW	119
LEFT	136

Breakpoint Commands	Page
LOCATE	140
MAXIMIZE	143
MINIMIZE	143
MOVE	147
NEXT	155
OFFSET	155
ORDER	158
PLOCATES	161
PREVIOUS	162
RIGHT	169
SEARCH	173
SET OFFSET	177
SIZE	182
TITLE	191
UP	194
Source Level Set and Query Commands	
BINARY	99
BIT	99
BRIEF	102
CHARACTER	104
COMPACT	107
DETAIL	111
EXLIMIT	116
FIXED	118
FLOAT	118
FMT	119
FORMAT	120
LANGUAGE COLOR	127
LANGUAGE COMMENTS	128
LANGUAGE DECLARES	128
LANGUAGE MACROS	132
LANGUAGE OPTIONS	132
LANGUAGE STATUS	133
LANGUAGE STEM	134

Breakpoint Commands	Page
LANGUAGE VERSION	134
LANGUAGE XPATH	134
PACKED	158
SPACE	184
VSEP	197
ZONED	200
Source Level Display Commands	
ARRAY	97
BOTTOM	100
FIRST	117
HIDE	124
LANGUAGE DEBUG	128
LANGUAGE DROP	129
LANGUAGE LOAD	129
LANGUAGE SCROLL	133
LAST	135
MAJOR	142
NAMES	154
SHOW	181
STRUCTURE	188
TOP	192
TYPE	193
UNION	193
VARIABLE	195
Logging Commands	
RLOG	170
VCHANGE	196
VS	196
Access and Control Register Commands	
ALET	96
AREGS	96
CREGS	108
LOCATION ALET	141
SET AREG	174

ABEND (CMS and z/OS)

Performs the usual IDF cleanup, then issues an OS ABEND. Use this command *only* for the intentional triggering of a working ABEND intercept routine.

```
►►──ABEND──┬────────────┬──────────────────────────────────────────────►◄
           └─abend-code─┘
```

abend-code
: A valid IDF expression denoting the ABEND code.

  If no argument is supplied, the command line is checked for an expression and if a valid expression is found its value is used as the ABEND code. If no argument is provided and the command line is empty, the current contents of the target program's R15 are used as the ABEND code.

ADSTOP (CMS only)

Sets one end of a Storage Alteration Stop (ADSTOP) range.

```
►►──ADSTop──┬────────────┬─────────────────────────────────────────────►◄
            └─expression─┘
```

expression
: The storage location. If not supplied, the address is determined from the cursor position, if possible.

If no expression is provided on the command line, and it is not possible to determine an address from the cursor position (for example when the cursor is on the command line and the command line is empty), this command is processed as an ADSTOPS command (see "ADSTOPS (CMS only)" on page 95).

All locations between (and including) the start address of the range and the end address of the range are monitored for storage alteration. Up to four separate storage areas can be monitored simultaneously. You are notified when a monitored location is modified.

Since the ADSTOP command sets one end of a range, you must issue it twice to define the range.

Storage locations can only be monitored for alteration when PER is enabled. Entering the ADSTOP command when PER is disabled produces an error message.

Because of the way the PER hardware works, IDF partially disassembles any instruction that changes PER storage, to determine the storage address modified. From this it determines whether the storage address falls in one of the defined ranges. The partial disassembly is valid for most of the common instructions. Some instructions do not resolve to the correct address. If you miss storage modification stops, you may need to set the 1ADSTOP option (see "SET OPTION" on page 178). This option treats the four ranges you have specified as a single storage area, from the lowest address in all of the ranges to the highest. When the 1ADSTOP option is set, partial disassembly is not needed to determine the address of modified data.

See also "SET ADSTOP (CMS only)" on page 174.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	PER is disabled, or an ADSTOP is already set at that location

ADSTOPS (CMS only)

Toggles the display of the AdStops window.

```
►►──┬─ADSTops──┬──────────────────────────────────────────────────────►◄
    └─REGSTops─┘
```

The AdStops window is opened if it is not already open. If the AdStops window is already open, it is closed.

See also "ADSTOP (CMS only)" on page 94, "SET ADSTOP (CMS only)" on page 174, and "SET REGSTOP (CMS only)" on page 180.

Return codes

0	Operation successful
6	PER is disabled

AFPR

Toggles the display of the Additional Floating-Point Registers window.

```
►►──AFPR──────────────────────────────────────────────────────────────►◄
```

The Additional Floating-Point Registers window is opened if it is not already open. If the Additional Floating-Point Registers window is already open, it is closed.

Return codes

0	Operation successful

ALARM

Controls the terminal alarm.

```
              ┌─ON──┐
►►──ALARM─────┤     ├─────────────────────────────────────────────────►◄
              └─OFF─┘
```

ON Ring the terminal alarm.

OFF
Reset the terminal alarm.

Examples

```
SET ALARM
SET ALARM ON
SET ALARM OFF
```

ALET

Specifies the ALET used to qualify the dataspace to be displayed in a Dump window.

```
►►──ALEt──┬────────┬──expression───────────────────────────────────────────►◄
          └─window─┘
```

window
: The window whose ALET is to be set. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Dump window, sets the ALET for the first Dump window.

expression
: If an access register is used in the expression, the ALET in the referenced access register is used instead of the value of the expression.

You must be in an ESA environment for this command to work.

Examples

```
SET ALET =3 X'00010003'
SET ALET =2 0(AR4)
```

APROGMSG (CMS only)

Enables and disables the trapping of asynchronous program-checks that happen while IDF displays the user interface.

```
►►──APROGMSG──┬─ON──┬──────────────────────────────────────────────────────►◄
              └─OFF─┘
```

ON Enable trapping of asynchronous program-checks.

OFF
: Disable trapping of asynchronous program-checks.

AREGS

Rotates the register display between the General Purpose and the Access Registers. Valid only in an ESA environment.

```
►►──AREGs──────────────────────────────────────────────────────────────────►◄
```

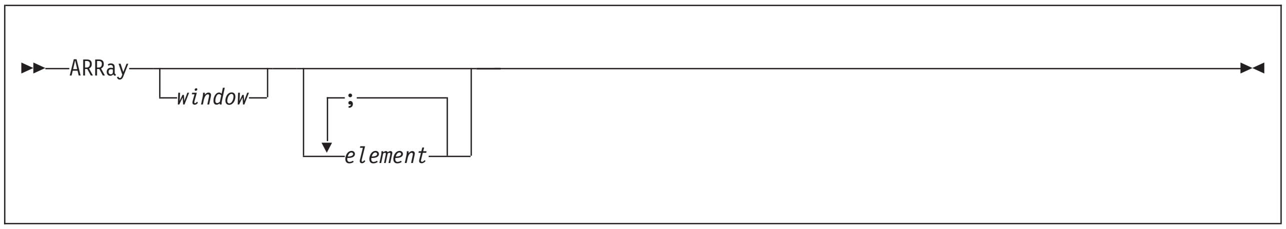
When issued, opens the Current Registers window if it is not already open.

Return codes

0	Register display was toggled
7	Not valid in this environment.

ARRAY

Displays the contents of one or more variables which are array elements.



window
The LSM Information window used to display the array element contents information. Select it by a Window Specification, or by placing the cursor in the window.

If supplied, the window must be an LSM Information window. If omitted, the first LSM Information window is used. If no LSM Information window is open, one is opened.

element
A variable name.

Simple variables are defined by name only. You can define based variables by name only, in which case the declared basing is used by IDF, or you can specify an explicit locating expression.

See "Variable expressions" on page 87 for a complete description of the syntax of the expressions which may be used for ARRAY variable name arguments.

The array element display persists until:
- An ARRAY command without arguments is issued
- The window is closed with a CLOSE command.
- Another IDF Language command such as VARIABLE, STRUCTURE, TYPE, CALLERS, PLOCATES, LANGUAGE STATUS, or MAP is issued, which directs IDF to update the LSM Information window with new information
- The target program completes execution
- Target program execution progresses beyond the variable's defined scope

If the contents of the array element change while the program is running to a breakpoint, the changed data is shown on the screen when the breakpoint is reached.

You can change the displayed data by overtyping it.

In EBCDIC display mode, character data equal to X'FF' or below X'40' is displayed as a period character.

In ASCII display mode, character data which does not correspond to a displayable EBCDIC character is displayed as a period character.

If a based variable was respecified, the current active USINGs will be used to locate the variable.

The display of the contents of the variables may be incorrect if the PSW indicates that execution is in the middle of a statement. This is because the variable may be in a transitional state, not having yet achieved its new value. Variable contents are only certain at the start and end of a statement.

Examples

```
array stuff(15)
array addr(x'20000')->stuff(20)
array addr(12(r2))->ptr->stuff(2)
array ptr->ptr2->stuff
array ptr(3)->ptr->stuff
array array1(10);array2(-5)
array chrarray(15,1:10);chrarray(15,1::10)
array chrarray[15,0:9];chrarray[15,0::10]
```

AUDIT

Enables and disables display of variable basing audit information.

```
►►──AUDit──┬─ON──┬──────────────────────────────────────────────────────────►◄
           └─OFF─┘
```

ON Enables the display of variable basing audit information.

OFF
Disables the display of variable basing audit information. This is the initial setting.

When a based variable is displayed, IDF can optionally provide an audit trail to show you the intermediate results of the variable location calculation.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

BACK

Makes a Dump window display the storage it displayed on the previous DUMP command for that window.

```
►►──BACK──┬────────┬─────────────────────────────────────────────────────────►◄
          └─window─┘
```

window
A Dump window. Select by a Window Specification or by placing the cursor in the window. If a Window Specification is not present and the cursor is not in a Dump window, uses the first Dump window.

IDF maintains the addresses used on the last ten DUMP commands for each open Dump window in a circular buffer.

Return codes

0 Operation successful

6 No Dump windows are open at this time.

BASE

Specifies the base address of the target program specified when IDF was invoked.

```
►►──BASe──expression──────────────────────────────────────────────────────►◄
```

expression
An expression that resolves to an address value.

Use the MODULE BASE command (see "MODULE BASE" on page 145) to set the base address of other programs defined to IDF.

May be used when debugging a self-loading nucleus extension to follow the code to its new location.

Examples

```
SET BASE X'21000'
```

BINARY

The BINARY command is a synonym of the FIXED command. For details see "FIXED" on page 118.

BIT

Sets or queries the format used to display the data for bit variables.

```
          ┌─BIT─┐
►►──BIT───┼─────┼─────────────────────────────────────────────────────────►◄
          ├─*───┤
          └─HEX─┘
```

BIT
Bit variables are displayed as a string of 0 and 1 characters.

***** Bit variables are displayed in the default format (BIT).

HEX
Bit variables are displayed in hexadecimal. This mode is especially useful for displaying long bit strings.

If the length of the bit string is not a multiple of 4 bits, the last hexadecimal character represents less than 4 bits. Since bit strings are left-aligned, the extraneous trailing bits are ignored if you overtype this nibble with a new value with 1 bits in any of these positions. A warning message is shown, and the new digit is displayed with the corrected hexadecimal value.

If the display format is not specified, a message shows the current display format for bit variables.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid bit variable display format

BOTTOM

Displays source code starting at the highest available address within the current code section.

```
►►──BOTtom──┬────────┬──────────────────────────────────────────────────►◄
            └─window─┘
```

window
A Disassembly window. Select by a Window Specification or by placing the cursor in the window.

If omitted, and the cursor is not in a Disassembly window, uses the first Disassembly window.

Return codes

0	Operation successful
6	No code section definition corresponds to the current address.

BREAK

Sets an instruction breakpoint, or toggles the Break window display.

```
►►──BREak──┬─────────┬──┬──────────────────┬────────────────────────────►◄
           └─address─┘  │  ┌────────────┐  │
                        └──▼─|──command─┴──┘
```

address
An address. If an expression, the expression is used to provide the address.

If omitted, the address is determined from the cursor position. If it is not possible to determine an address from the cursor position (for example when the cursor is on the command line and the command line is empty), the Break window is opened if it is not already open, or closed if it is open.

command
A command to be executed immediately after the breakpoint is taken.

To set a deferred or sticky breakpoint, use the DBREAK command. See "DBREAK" on page 109 for details.

If an address is supplied, either as an expression on the command line or by means of cursor position, a breakpoint is set at that address. If a breakpoint or a watchpoint is already set at the specified address, it is cleared. The BREAK command is a toggle that turns a breakpoint on or off. (The OLDBREAK option is available if the toggle style of operation is not desired. If OLDBREAK is set and the BREAK command is used against an address where a breakpoint is already set, an error message is issued.)

The breakpoint is taken (that is, execution is interrupted and the operator notified) just prior to the execution of the indicated instruction. A maximum of 64 breakpoints may be active at one time. Breakpoints remain in effect until they are explicitly cleared (through the Break window).

You can associate a list of commands with the breakpoint. These commands are executed when the breakpoint is taken, before control is returned to you. The commands are specified at the end of the BREAK command, separated from the address and each other by vertical bars (|). If a command receives a non-zero return code, the remaining commands in the list are not executed.

z/VM IDF provides two types of instruction breakpoint, PER and non-PER. If PER is enabled (with the Break window or SET PER command) then all breakpoints are PER breakpoints. Likewise, if PER is disabled, all breakpoints are non-PER breakpoints.

- PER breakpoints are implemented by means of PER Instruction Fetch and Branch events, and are only possible so long as the processor can be kept in Extended Control mode. In some cases, for example when an SVC is issued in a read-only DCSS, Extended Control mode is dropped by CMS.
- Non-PER breakpoints are implemented by temporarily inserting an invalid opcode at the indicated location.

For more information about the distinction between the two available breakpoint types, see “PER versus non-PER mode” on page 56.

z/OS IDF provides two types of instruction breakpoint, SVC 97 and non-SVC 97.

- SVC 97 breakpoints are implemented by temporarily inserting an SVC 97 instruction at the indicated location.
- Non-SVC 97 breakpoints are implemented by temporarily inserting an invalid opcode at the indicated location.

For more information about the distinction between the two available breakpoint types, see “Breakpoint method selection (TSO)” on page 41.

z/VSE IDF provides one type of breakpoint.

- A breakpoint is implemented by temporarily inserting an invalid opcode at the indicated location.

If a WATCH command with a condition is issued for an address at which there is a breakpoint, that breakpoint is converted into a watchpoint. If commands were specified with the original breakpoint, they are associated with the watchpoint unless commands were specified with the WATCH command.

If a breakpoint is placed within the execution path of the subroutine to be skipped, execution stops at that breakpoint.

IDF verifies that there is a valid instruction at the breakpoint address, and rejects the BREAK command (with a message) if not. This test is not infallible. Certain combinations of DATA can appear as valid instructions. For example X'1E14' might be data or it might be an ALR R1,R4. IDF lets you install a break here.

For non-PER and non-SVC 97 breakpoints, IDF uses an invalid instruction of the format x'02xx', to cause a program-check (operation exception), which gives IDF control to handle the breakpoint logic. At the time of handling the breakpoint, IDF reinstalls the original instruction bytes for execution. But if you manage to install a non-PER breakpoint (that is, x'02xx') on top of program DATA (which looks like an instruction), then IDF never gains control to remove its x'02xx' from your program's DATA. So IDF corrupts valid data, potentially leading to bizarre program behavior. Take care to install breakpoints only on valid target program instructions.

IDF establishes breakpoints by modifying target instructions to invalid opcodes (or SVC 97 instructions if SVC97 under TSO is in effect). The DISASM window only displays the original instruction. If a breakpoint is established on an instruction which is the target of an EXecute instruction, then the EX instruction will fail.

z/VM and z/VSE

As part of the test for a valid instruction, IDF backs up 2 bytes from the breakpoint instruction (assuming that the instruction appears valid so far), and checks for:

On CMS

SVC 201 (x'0AC9'), SVC 202 (x'0ACA'), and SVC 203 (x'0ACB').

On z/VSE

SVC 34 (x'0A22')

Each of these SVCs may be followed by DATA bytes. If IDF finds any of these codes preceding the breakpoint address, IDF assumes that the breakpoint instruction is in fact DATA for the SVC, and prevents the breakpoint installation.

This check cannot be completely certain that the x'0Axx' preceding the breakpoint address is in fact an SVC. It is possible that it is also DATA, or perhaps the last 2 bytes of a 4-or 6-byte instruction. However the risk of either of these is small, and since the result of allowing a breakpoint to be installed on top of SVC 203 data (for example) is catastrophic, being over-zealous in its checking is worthwhile.

If IDF rejects (due to the above SVC test) an attempted breakpoint installation on what you know for certain is a valid instruction you should be able to successfully install the break on the *next* instruction.

See also "SET BREAK" on page 175.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	Location does not contain a valid instruction, location is read only and PER is unavailable, or a breakpoint already set at that location and the OLDBREAK option is on.

BRIEF

Controls the display of declaration information when a variable is displayed.

```
►►──BRIef──┬─ON──┬──────────────────────────────────────────────────────►◄
           └─OFF─┘
```

ON Disables the display of declaration information.

OFF
Enables the display of declaration information. This is the initial setting.

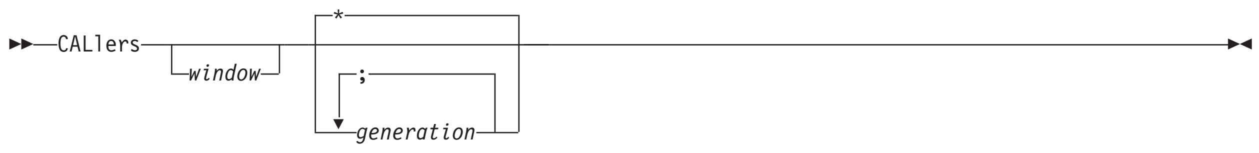
The COMPACT option is initially ON, which also suppresses the declaration information. To display the declaration information, both the COMPACT and the BRIEF options must be OFF.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

CALLERS

Displays information for each generation in the program caller hierarchy.



window
: A LSM Information window. Select by a Window Specification, or by placing the cursor in the window.

  If supplied, the window must be an LSM Information window. If omitted, and the cursor is not in an LSM Information window, uses the first LSM Information window. If no LSM Information window is open, one is opened.

*
: Information is shown for all caller generations, beginning with the current location and followed by all previous generations along the save area chain until either:
  - the save area back link is zero
  - the save area back link contains an invalid address
  - the generation specified by the SALIMIT command is reached

generation
: A program caller generation.

  The program caller generations numbering convention is:

0	Current program
1	Parent (caller)
2	Grandparent (caller of caller), and so on

  When supplied, information is displayed for only the nominated caller generations.

The information displayed for each generation in the program caller hierarchy includes:
- Current execution location, as:
  - Memory location, in IDF symbolic format
    ```
    (module.CSECT)Stmt#nnnnn+offset
    ```
  - Logical location
    ```
    program-block-name+offset
    ```
    This location is only provided when extract data was loaded, for the program. Extra information may also be used from data areas in the program's storage.
- Save Area Header
- Save Area register values, if applicable

Use the SAREGS command to enable and disable the display of the Save Area header and registers in the CALLERS display. The Save Area registers are formatted according to the IDF ROWSTYLE option setting.

Use the SALIMIT command to control the maximum depth of the CALLERS display. This is intended to prevent problems when the program call chain is damaged, or is of unexpected depth (due to runaway recursion).

The calling hierarchy information display persists until:

- A CALLERS command without arguments is issued when the current Call Hierarchy command has no arguments.
- The window is closed with a CLOSE command.
- Another IDF Language command such as VARIABLE, STRUCTURE, ARRAY, TYPE, PLOCATES, LANGUAGE STATUS, or MAP is issued, which directs IDF to update the LSM Information window with new information.
- The target program completes execution.

Examples

```
CALLERS
CALLERS 0;99
```

CHARACTER

Sets or queries the format used to display the data for character variables.

```
                 ┌─EBCdic────┐
►►──CHAracter────┼───────────┼──────────────────────────────────────────────────────►◄
                 ├─*─────────┤
                 ├─ASCii─────┤
                 ├─CHAracter─┤
                 ├─HEX───────┤
                 ├─PACked────┤
                 └─ZONed─────┘
```

EBCDIC
: Character variables are EBCDIC, with unprintable characters (X'00' to X'3F', X'FF') displayed as . (periods).

*****
: Character variables are displayed in the default format (EBCDIC).

ASCII
: Character variables are ASCII, with unprintable characters displayed as . (periods).

CHARACTER
: Character variables are displayed as EBCDIC or ASCII, tracking the value of the ASCII setting selected for the DUMP command. Unprintable characters are displayed as . (periods).

HEX
: Character variables are displayed in hexadecimal. This mode is especially useful for buffers which are declared as CHAR and which contain non-character information.

PACKED
: Character variables of less than 24 bytes are displayed in Packed Decimal format. Variables exceeding this length are displayed in hexadecimal. This mode is especially useful for examining data in a language without native Packed Decimal variables.

ZONED
: Character variables of less than 48 bytes are displayed in Zoned Decimal format. Variables exceeding this length are displayed in EBCDIC. This mode is especially useful for examining data in a language without native Zoned Decimal variables.

If the display format setting is not specified, the current display format for character variables is shown in a message.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid character variable display format

CHECK

Controls the checking of selective input values when variable information is displayed or altered by overtyping.

```
►►──CHEck──┬─BOUnds────┬──┬─ON──┬──────────────────────────────────────────────►◄
           ├─NEGative──┤  └─OFF─┘
           ├─SUBstring─┤
           └─┬─ALL─┬───┘
             └─*───┘
```

BOUNDS
Controls array index specification processing.

ON Array indices must be inside the declared range.

OFF
Array indices outside the declared range may be specified, allowing you to follow errant program results.

This option also affects scrolling behavior of the ARRAY command.

NEGATIVE
Determines whether negative values can be assigned to Unsigned Fixed variables.

ON An error message is issued and the variable is not updated.

OFF
A warning message is issued and the variable is updated. Attempts are made to update both the storage and register portions of shadowed variables, but expressions derived from variables are *not* reevaluated.

SUBSTRING
Controls substring index specification processing for character string and bit string variables.

ON Substring indices must be inside the declared range.

OFF
Substring indices outside the declared range (including negative values) may be specified, allowing you to follow errant program results.

ALL | *
Applies ON or OFF processing to BOUNDS, NEGATIVE, and SUBSTRING.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

CLOSE

Closes a window.

```
►►──CLOse──┬────────┬──────────────────────────────────────────────►◄
           └─window─┘
```

window
A window. Select by a Window Specification or by placing the cursor in the window.

Return codes

0	Operation successful
1	No window selected.

COLORS

Sets the colors used to display IDF display elements.

```
►►──┬─COLors──┬──MHTI──────────────────────────────────────────────►◄
    └─COLours─┘
```

MHTI
A four-letter color specification. Each position indicates the color for a screen element. The positions are:

1	Messages
2	Headings
3	Text
4	Input areas

Valid letters are:

B	Blue
G	Green
P	Pink
R	Red
T	Turquoise
Y	Yellow
W	White

Examples

```
SET COLORS RYGB
```

This example sets the following color scheme:

Messages	Red
Headings	Yellow
Text	Green
Input	Blue

COMMAND

Performs the IDF command specified on the command line.

```
►►──COMmand──text──────────────────────────────────────────────────────────────►◄
```

IDF-command
: The IDF command to be executed.

arguments
: Arguments to be passed to the command. The particular arguments depend on the command that is supplied.

The command line is examined for the name of the IDF command. If the command is not known to IDF, the IMPMACRO option is checked. If this indicates that implied macro processing is desired, the indicated macro is used as the command.

The command name is then temporarily removed from the command line, so that any arguments which are present are available to the specified command. The command is then executed as if you have pressed a PF key.

If the command line is empty and no input fields in the remainder of the screen are changed, the only function performed is to move the cursor to the command line.

Return codes

The return code when COMMAND is issued by a macro is the propagated return code from the IDF command.

COMPACT

Minimizes the number of lines used for variable display.

```
►►──COMPact──┬─ON──┬──────────────────────────────────────────────────────────►◄
             └─OFF─┘
```

ON As with BRIEF ON, disables the display of declaration information. This is the initial setting.

To further reduce the number of display lines, if the length of the variable value permits the variable name and contents are shown on the same display line. If sufficient space exists, the variable location information is also shown in this case.

OFF
: As with BRIEF OFF, enables the display of declaration information.

When you display a variable, IDF normally uses multiple lines of the display to show all the variable information. When variables with short names and short formatted values are displayed, much of these display lines contain blanks. Use the COMPACT display mode to minimize the number of lines used for variable display.

As with the BRIEF display mode, the optional variable declaration information is suppressed.

To maximize the number of variables that can be displayed, you can use the SPACE OFF command to eliminate the blank line optionally generated between variables.

The COMPACT option is initially ON, which also suppresses the declaration information. To display declaration information, the COMPACT and BRIEF options must both be OFF.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

CREGS (CMS only)

Toggles the register displayed between the General Purpose and Control Registers. Valid only while PER exploitation is disabled (option PER is N).

```
►►──CREGs──────────────────────────────────────────────────────────────────►◄
```

The CREGS command opens the Current Registers window if it is not already open.

Return codes

0	Register display was toggled
6	Unable to comply due to PER setting

CURSOR

Provides IDF macros with some control over cursor positioning.

```
►►──CURsor──┬─STAY──────────┬──────────────────────────────────────────────►◄
            ├─COMmand───────┤
            ├─DISasm────────┤
            ├─DUMP──────────┤
            │           (1) │
            └─xx──yy──zz────┘
```

Notes:

1 2-digit hexadecimal values, separated by blanks.

`STAY`
Leaves the cursor at the position it had when the PF key was pressed.

`COMMAND`
Positions the cursor at the start of the command line.

`DISASM`
Places the cursor in the first input field of the Disassembly window.

`DUMP`
Places the cursor in the first input field of the Dump window.

xx yy zz

Positions the cursor at an absolute position on the screen.

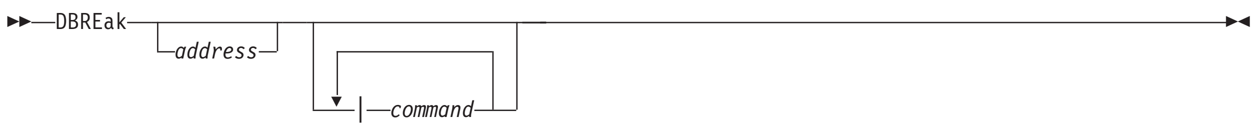
These three values are blank separated, 2-digit *hexadecimal* numbers representing:
- The window containing the cursor
- The row within the window of the cursor
- The column within the window of the cursor

Examples

```
SET CURSOR STAY
SET CURSOR COMMAND
SET CURSOR DISASM
SET CURSOR DUMP
'SET CURSOR' varname
```

DBREAK

Sets or clears a deferred instruction breakpoint.



address

An address. If an expression, the expression is used to provide the address.

If omitted, the address is determined from the cursor position. If it is not possible to determine an address from the cursor position (for example, when the cursor is on the command line and the command line is empty), the Break window is opened if it is not already open, or closed if it is open.

command

A command to be executed.

This function is a variation of the BREAK function (see “BREAK” on page 100). Breakpoints established with this command are placed in a special IDF table.

If an address is supplied, either as an expression on the command line or by means of cursor position, a deferred breakpoint is set at that address. If a breakpoint or a watchpoint is already set at the specified address, it is cleared. The DBREAK command acts as a toggle to turn a breakpoint on or off. (The OLDBREAK option is available to disable the toggle style of operation. If OLDBREAK is set and the DBREAK command is used against an address where a breakpoint is already set, an error message is issued.)

The DBREAK command only works on TSO if the SVC97 option is in force.

If the module containing the deferred breakpoint is *not* in storage (or defined to IDF):
- IDF monitors program loading
- When IDF detects the module is loaded, (or the TRIGGER LOAD command is issued) a standard instruction breakpoint is established at the location specified within the module.
- The deferred breakpoint remains active:
  - if the module is later dropped from IDF's module list (and possibly removed from storage), the standard instruction breakpoint is removed automatically.

– if the module is once again loaded into storage, the deferred breakpoint is again triggered, and a new standard instruction breakpoint established.

Thus IDF deferred breakpoints may be termed “sticky”.

If the module containing the deferred breakpoint *is* in storage, the initial monitoring of program loading is bypassed.

If any deferred breakpoints are active, they are shown in a special section of the Break window. See “BREAK” on page 100 for details regarding the operation of the Break window.

1. The full syntax for all IDF symbols is:

```
►►──┬─────────────────────────────┬─SYMbol──┬─────────────────┬────────────────────────────►◄
    ├─(section-name)──────────────┤         └──┬─ + ─┬─offset─┘
    └─(module-name.section-name)──┘            └─ - ─┘
```

   With most other IDF commands, you are working within the qualified module. This is the default module name if *module-name* is omitted from the symbol specification.

   The QUALIFY command makes the module you are about to load the default module. If this is not done, you must be careful to explicitly specify the correct module name in the DBREAK command parameter.

   For dynamically loaded programs, see also “LANGUAGE LOAD” on page 129.
2. Deferred breakpoints are normally for modules not yet in storage. In this case, the expression must specify a symbolic address, since the base address of a module is not known until after it is loaded by another program or command.
3. Each time a new module is encountered, the module name is checked. If it matches the name of one of the desired modules, IDF then attempts to load the symbol table (from the Load Module on z/OS, from the MAP file on CMS, from the librarian MAP member in z/VSE). If this fails, IDF ignores the module.
4. If no section or symbol is specified, the default is to establish the breakpoint at the entry point of the module.

   For example, to insert a deferred breakpoint at the entry point of the module MOD1, the command is:

   ```
   DBREAK (MOD1.)
   ```
5. DBREAK supports an extension to the standard IDF symbol syntax. If * is specified as the module name, the deferred breakpoint is triggered whenever a new module is loaded.
6. Only use external symbols in a DBREAK address expression.
7. While hexadecimal or decimal offsets may be used, this is not encouraged. The definition of specific external symbols for use as deferred breakpoint locations ensures that operation are not affected as section sizes (and hence offsets) change over the course of program development.

Return codes

0	Operation successful
5	Syntax or other error in expression

DETAIL

Controls the display of data for the structure components of intermediate depth. Initially, this data is not shown.

```
►►──DETail──┬─MINimum─┬──┬─number-of-levels───┬───────────────────────────────────►◄
            └─MAXimum─┘  ├─ + number-of-levels─┤
                         └─ - number-of-levels─┘
```

MAXIMUM
Enables the display of intermediate component data.

MINIMUM
Disables the display of intermediate component data.

number-of-levels
Integer, indicating number of levels of structure component data to be displayed.

+ *number-of-levels*
Integer, indicating number of extra levels of structure component data to be displayed.

- *number-of-levels*
Integer, indicating the number of fewer levels of structure component data to be displayed.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
5	Arguments are invalid

DISASM

Displays a disassembly listing.

```
►►──DISasm──┬────────┬──┬─────────┬───────────────────────────────────────────────►◄
            └─window─┘  └─address─┘
```

window
A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Disassembly window, uses the first Disassembly window.

address
An address specifying the storage location at which the disassembly listing begins.

If an address is supplied on the command line, or an address can be determined from the cursor position, a disassembly of the indicated memory area is shown.

If no address is supplied, and an address cannot be determined from the cursor position, a disassembly listing is displayed, beginning at the first instruction which previously appeared on the disassembly display.

If the DISASM function is invoked when no Disassembly windows are open, one is opened.

If the DISASM function is invoked when a Disassembly window is open, but no address is specified, the selected window is closed.

If the DISASM function is invoked when a Disassembly window is open, and an address is specified, the starting address of the disassembly in the selected window is changed.

By positioning the cursor in the first halfword of an instruction and pressing DISASM, you can make that instruction the first one displayed.

If the cursor is positioned in the first halfword of the first instruction displayed, and that instruction is a branch, the effective address of that branch instruction becomes the first instruction disassembled.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	Specified address exceeds the current virtual memory size

DOWN

The DOWN command is a synonym of the NEXT command. For details, see “NEXT” on page 155.

DROP GLOBAL

Discards the information for stems from Global Storage.

```
                   ┌──────────┐
                   ▼          │
►►──DROp──GLObal────stem-name.┴──────────────────────────────────────────────►◄
```

stem-name
  A Global Storage stem name. Each stem name must have a trailing period.

Return codes

0	Operation successful
1	Missing stem name
2	Stem name truncated
5	Stem name specified is not defined in Global Storage

DROP MODULE

Discards information known about a module and the symbols associated with the module.

```
►►──DROp──MODUle──module-name────────────────────────────────────────────────►◄
```

module-name
  The name of the module for which information is discarded.

The module specified when IDF was invoked cannot be dropped.

Return codes

0	Operation successful
1	Missing module name
2	Module name truncated
5	Module specified not known to IDF

DROP SYMBOLS

Discards symbols known to IDF.

```
►►──DROp──SYMbols──┬─────────────┬──────────────────────────────────────────────►◄
                   └─module-name─┘
```

module-name
: The name of the module for which symbols are dropped. If omitted, the symbols of the qualified module are dropped.

DROP SYMBOLS and LOAD SYMBOLS (see "LOAD" on page 138) lets you use symbol information from any file, providing the file is recognized by IDF.

DUMP

Displays a storage dump.

```
►►──DUMP──┬────────┬──┬─────────┬───────────────────────────────────────────────►◄
          └─window─┘  └─address─┘
```

window
: A Dump window. Selected by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Dump window, uses the first Dump window.

address
: The storage location at which the storage dump begins.

  If an address is supplied on the command line, or an address can be determined from the cursor position, a storage dump of the indicated memory area is shown.

  If no address is supplied, and an address cannot be determined from the cursor position, a storage dump listing is displayed, beginning at the first instruction which previously appeared on the dump display.

If the DUMP function is invoked when no Dump windows are open, one is opened.

If the DUMP function is invoked when a Dump window is open, but no address is specified, the window is closed.

If the DUMP function is invoked when a Dump window is open, and an address is specified, the starting address of the storage displayed in the window is changed.

IDF provides two DUMP modes, symbolic and unformatted, toggled by the DUMPMODE command.

The symbolic dump shows the names of storage areas along with the contents of each named area. The unformatted dump shows the traditional storage dump.

The storage dump shown by the DUMP command is presented in the current dump format. Toggle the dump format with the DUMPMODE command.

By positioning the cursor in a hexadecimal field displayed on the screen before pressing DUMP, you can make that field the first one displayed.

If the cursor is positioned in the first field displayed, and that field is a fullword and in AMODE24, the low-order 24 bits of the contents of that word specify the first location to DUMP. If in AMODE31, the low-order 31 bits are used instead. If in AMODE64, the contents of that word and the next word are used as the first location to dump.

Using this feature, it is easy to follow a linked list. Just place the cursor in the fullword that represents the link word, and repeatedly press the DUMP key.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	Specified address exceeds the current virtual memory size

DUMPMODE

Toggles the DUMP format between symbolic and unformatted.

```
►►──DUMPMode─────────────────────────────────────────────────────────►◄
```

Return codes

0	Operation successful

EPNAMES

Displays the Entry Point Names window.

```
►►──EPNAMES──┬──────────────┬────────────────────────────────────────►◄
             └─section-name─┘
```

section-name
The name of the first section to be displayed.

Return codes

0	Operation successful

If you invoke the EPNAMES function when no Entry Point Names window is opened, one is opened.

If you invoke the EPNAMES function when an Entry Point Names window is open, but you specify no section-name, the window is closed.

If you invoke the EPNAMES function and specify a section-name, and an Entry Point Names window is open, the contents of the window change to display information about the entry point.

If the module that IDF loaded has more than one section-name, and thus more than one section-name is loaded, you can use the PREVIOUS and NEXT commands to scroll the Entry Point Names window to view the information for the extra section-names.

IDF derives the section-name in this manner:
- If the long name is eight characters or less, this is the uppercase representation of the name.
- If the long name is more than eight characters in length, the section-name is the first five characters, followed by a double quote ("), followed by the last two characters of the long name, all in uppercase. For example, the long name `Set_IDF_Rules_message_in_ASMMSAM1-VChar` is transformed to the short name `SET_I"AR`.

You can modify this field by overtyping it.

EPOFFSET

Specifies the offset of the primary entry point within a program.

```
►►──EPOffset──entry-point-offset──────────────────────────────────────────────►◄
```

entry-point-offset
: An expression which is resolved to the entrypoint offset value. This entrypoint offset is the entrypoint address of the program less the start (base) address of the program.

Example

```
SET EPOFFSET X'24'
```

EXITEXEC

Enables or disables (toggles) exit routine processing.

```
►►──EXItexec──────────────────────────────────────────────────────────────────►◄
```

If the current exit routine is present and enabled, it is invoked when execution of the target program is normally interrupted to notify the operator of an unusual event (such as a breakpoint). The exit routine then determines whether to notify the operator of the event.

If exit routine processing is disabled, the exit routine is not invoked and the operator is notified as usual when one of these events occurs.

For more information, see Chapter 13, "The IDF exit routine," on page 213.

Return codes

0	Operation successful
6	Exit routine not found

EXLIMIT

Sets the maximum LSM stemmed array index during EXTRACT LANGUAGE commands execution.

```
►►──EXLimit──max-stemmed-array-index─────────────────────────────────────►◄
```

max-stemmed-array-index
: Maximum stemmed array index. Integer between 1 and 9999999. The initial value is 20000.

When information is written to the stemmed array, user free storage is consumed. This can present a problem if data is extracted for extremely large variables, or for variables such as unbounded bit or character strings which have no upper limit (aside from the end of machine storage). In these cases, no storage remains for application or IDF operation. EXLIMIT helps prevent this from happening.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
5	Arguments are invalid

FIND

Locates a string and displays the section of code where it occurs.

```
►►──Find──┬────────┬──┬─string─┬──┬──────────────────────────────┬──┬───────────┬──────────────►◄
          └─window─┘  └─*──────┘  └─start-col──┬──────────────┬─┘  ├─FIRst─────┤
                                               └─finish-col───┘    ├─LASt──────┤
                                                                   ├─NEXt──────┤
                                                                   └─PREvious──┘
```

window
: A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted, and the cursor is not in a Disassembly window, uses the first Disassembly window.

string
: The group of characters being searched for, the search string.

  Enclose this search text in quotes if it is numeric or contains embedded blanks. Both "..." and '...' are accepted.

*
: Use the current search string.

start-col
: The column at which searching starts. Integer. If omitted, searching starts from column 1.

finish-col
: The column at which searching ends. Integer greater than the start column. If omitted, searching finishes at column 80.

FIRST
: Begin search at lowest address, and look for search string in a forward direction.

`LAST`
Begin search at highest address, and look for search string in a reverse direction.

`NEXT`
Begin search at current address, and look for search string in a forward direction.

`PREVIOUS`
Begin search at current address, and look for search string in a reverse direction.

The function of this command is essentially the same as the ISPF editor FIND command. The search begins at the first source line shown on the screen; the target code, if found, is displayed at the top of the screen.

Unless otherwise qualified, the search is performed from the current address, in the direction last specified.

Examples

```
FIND string

f 'text'
f text
f text

f info first
f init( next
f * prev
f bit( last

find =3 'window 3'
```

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
5	Arguments are invalid
6	Specified string was not located or the search was not conducted

FIRST

Displays the source code corresponding to the lowest memory address.

```
►►──FIRst──┬────────┬──────────────────────────────────────────────►◄
           └─window─┘
```

`window`
A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Disassembly window, uses the first Disassembly window.

Return codes

0	Operation successful

FIXED

Sets or queries the format in which the data for fixed binary variables are displayed.

```
►►──┬─FIXed──┬──┬───────────┬─────────────────────────────────────────────►◄
    └─BINary─┘  ├─DECimal───┤
                ├─*─────────┤
                └─HEX───────┘
```

DECIMAL
Fixed binary variables are displayed in decimal. This is the initial value.

***** Fixed binary variables are displayed in the initial format (DECIMAL).

HEX
Fixed binary variables are displayed in hexadecimal.

If the display format setting is not specified, the current display format for fixed binary variables is shown in a message.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid fixed binary variable display format

FLOAT

Sets or queries the format in which the data for float variables is displayed.

```
►►──FLOat──┬──────────────┬───────────────────────────────────────────────►◄
           ├─┬─STD──────┬─┤
           │ ├─STAndard─┤ │
           │ └─FIXed────┘ │
           ├─*────────────┤
           ├─SCIence──────┤
           └─HEX──────────┘
```

STD | STANDARD | FIXED
Float variables are displayed in fixed-point (or standard) format - for example, as 145.0056. This is the initial value.

***** Float variables are displayed in the initial format (STD).

SCIENCE
Float variables are displayed in scientific format - for example, as 1.450056E+02.

HEX
Float variables are displayed in hexadecimal. Use this option to display binary floating-point variables.

If STD is selected, and the number cannot be displayed in standard format, it is displayed in scientific format.

Numbers may be entered in either fixed-point or scientific format when FIXED or SCIENCE formats are selected.

If the display format setting is not specified, the current display format for float variables is shown in a message.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid float variable display format

FMT

The FMT command is a synonym of the FORMAT command. For details, see “FORMAT” on page 120.

FOLLOW

Makes a Dump window follow the contents of a register or storage location as program execution progresses.

```
►►──FOLlow──┬────────┬──┬─address─┬──────────────────────────────────────────────►◄
            └─window─┘  └─OFF─────┘
```

window
A Dump window. Select the window by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Dump window, uses the first Dump window.

address
The address to be followed. If an expression, the expression is used to provide the address.

If omitted, the address is determined from the cursor position. If it is not possible to determine an address from the cursor position (for example, when the cursor is on the command line and the command line is empty), the current follow state of the selected window is displayed.

If R0 to R15, the indicated register's contents determine the starting DUMP address as target program execution progresses.

The argument “R1” means follow the contents of R1, but the argument “0(R1)” means follow the contents of the storage area now pointed to by R1. If in AMODE64, this is an 8-byte storage area, otherwise it is a 4-byte storage area.

If no register is specified, is the address of a storage area whose contents are to determine the area to be shown in the selected Dump window as program execution progresses. This includes addresses specified by means of cursor position. The storage area need not be aligned.

OFF
Following is turned off for the window. The window contents change to reflect any change in the contents of the current address.

Each open Dump window can have its own follow address or none at all.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	Specified address exceeds the current virtual memory size

FORMAT

Controls the way in which a variable is displayed, or shows the display format for the variable.

```
►►──┬─FORmat─┬───┬─variable-name────────────────────────┬──────────────────────────────►◄
    └─FMT────┘   │ ┌─;─────────────┐                    │
                 │ ▼               │                    │
                 └─variable-name───┴──┬─*─────────────┬─┘
                                      └─format-type───┘
```

variable-name
: The variable for which the format is being set.

  If only one variable is supplied, and there are no following parameters, then a message is displayed. The message indicates the display format for the variable.

`*`
: The variables listed inherit their display format from the current display format for the underlying variable classes.

format-type
: The display format for the listed variables.

  Must be appropriate to the underlying data classes of the variables:

Binary float	HEX
Bit	BIT | HEX
Char	EBCDIC | ASCII | CHAR | HEX | PACKED | ZONED
Fixed	DECIMAL | HEX
Hexadecimal float	STD | STANDARD | FIXED | SCIENCE | HEX
Packed Decimal	DECIMAL | HEX
Zoned Decimal	DECIMAL | HEX

  See the following commands for more details:
  - "BIT" on page 99
  - "CHARACTER" on page 104
  - "FIXED" on page 118
  - "FLOAT" on page 118
  - "PACKED" on page 158
  - "ZONED" on page 200

The format in which a variable is displayed is set in two ways:
1. With the appropriate IDF command to change the default display format for the *all* variables within the fundamental data type (or class) to which the variable belongs
2. With the FORMAT command, to specify an explicit display format for this particular variable. This overrides the class display format.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid display format for specified variables

FPC

Changes the contents of the Floating Point Control Register (FPC).

```
►►──FPC──FPC-value──────────────────────────────────────────────────────────►◄
```

FPC-value
The new FPC value. From one to eight hexadecimal digits, right-aligned.

Example

```
SET FPC 0 FE430000
```

FPR

Changes the contents of a Floating Point Register (FPR).

```
►►──FPR──FPR-number──FPR-value──────────────────────────────────────────────►◄
```

FPR-number
The FPR number, which can be in the range 0 - 15.

FPR-value
The new FPR value. From one to sixteen hexadecimal digits, right-aligned.

Example

```
SET FPR 0 FE43000000000000
```

GLOBALS

Displays information about Global Storage stems.

```
►►──GLObals─────────────────────────────────────────────────────────────────►◄
```

The LSM Information window containing the GLOBALS display is closed if you issue another GLOBALS command.

To define Global Storage areas, see "SET GLOBAL STEM" on page 176.

The GLOBALS command is meant primarily for debugging situations, or for verifying that IDF understood your commands.

Return codes

0	Operation successful
6	There are no Global Storage stems defined.

GOTO

Evaluates an expression and places it in the address portion of the Program Status Word (PSW).

```
►►──┬─GOTo─┬──expression──────────────────────────────────────────────►◄
    └─PSW──┘
```

expression
Specify as an address expression or by the cursor position.

Return codes

0	Operation successful
1	No address specified
5	Syntax or other error in expression
6	Conditions do not permit completion of command

GPACK

Returns the Global Storage data storage areas that no longer contain Global Storage stem data to the operating system free storage pool.

```
►►──GPAck─────────────────────────────────────────────────────────────►◄
```

The GPACK command helps when free storage is at a premium.

Return codes

0	Operation successful
Other	Error occurred while packing Global Storage data storage areas. This is most likely caused by an overlay of the Global Storage data AREA control information by an errant application program under test.

GPR

Changes the low-order 32 bits of a General Purpose Register (GPR).

```
►►──GPR──register-number──expression──────────────────────────────────►◄
```

register-number
The GPR number. An integer from 0 to 15.

expression
The new GPR value. Values are right-aligned with leading zeroes.

Example

```
SET GPR 2 ALLOPEN+X'44'
```

GPRG (z/OS only)

Changes the contents of a 64-bit of a General Purpose Register (GPR).

```
►►──GPRG──register-number──expression──────────────────────────────────────────►◄
```

register-number
: The GPR number. An integer from 0 to 15.

expression
: The new GPR value. Values are right-aligned with leading zeroes.

Example

```
SET GPRG 0 X'1234000123456'
```

GPRH (z/OS only)

Changes the high-order 32 bits of a 64-bit of a General Purpose Register (GPR).

```
►►──GPRH──register-number──expression──────────────────────────────────────────►◄
```

register-number
: The GPR number. An integer from 0 to 15.

expression
: The new value to be placed in the upper 32 bits of the register. Values are right-aligned with leading zeroes.

Example

```
SET GPRH 0 1200
```

GSTATUS

Displays information about the storage used to contain the Global Storage stem data which was loaded with SET GLOBAL STEM commands.

```
►►──GSTAtus────────────────────────────────────────────────────────────────────►◄
```

The LSM Information window containing the GSTATUS display is closed if you issue a GSTATUS command when the window is open. If not opened, it is opened.

The information includes:

- number of Global Storage stems
- Global Storage storage consumption (total, direct, pooled)
- Global Storage storage pool utilization, including the number of areas in the pool that are unused

The GSTATUS command helps when free storage is at a premium.

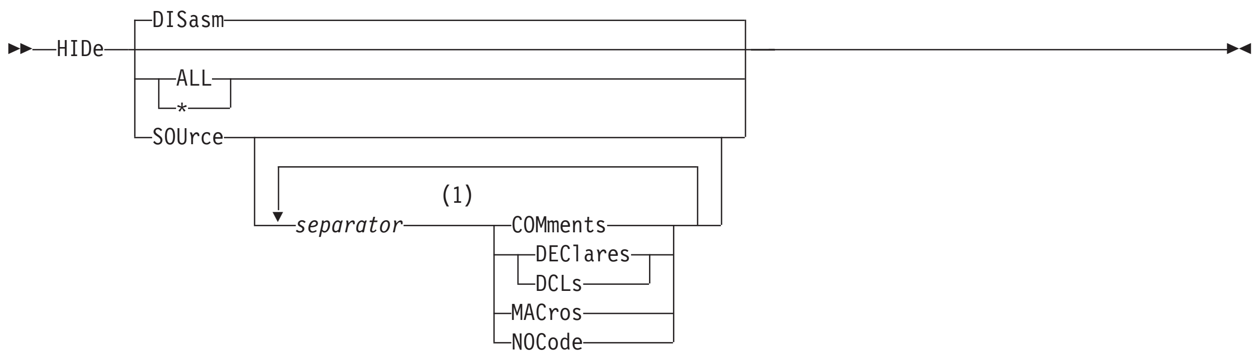
When Global Storage data is removed from storage with DROP GLOBAL, the storage areas which contained the extract data are retained for use by later SET GLOBAL stemname commands. Use the GPACK command to return to the operating system free storage pool those storage areas that no longer contain Global Storage data.

Return codes

0 Operation successful

HIDE

Controls the display of source code and disassembly.



Notes:

1 An option can be chosen no more than once.

DISASM
Show source code only, without interspersed assembler code.

ALL | *
Show disassembled assembler code only, without source code. The display of comments, declarations, macro expansions, and source lines with no corresponding object code is also disabled, so that these lines are excluded when source display is later enabled by a SHOW SOURCE or SHOW BOTH command. These may be enabled by appropriate SHOW commands.

SOURCE
Show disassembled assembler code only, without source code.

COMMENTS
Exclude block comment source when source code is displayed.

SEPARATOR
A comma, blank, or semicolon. Separates SOURCE and its suboptions.

DECLARES | DCLs
Exclude declaration source when source code is displayed.

MACROS
Exclude macro expansion source when source code is displayed.

NOCODE
Exclude source lines with no corresponding object code when source code is displayed.

The HIDE command assumes that everything is showing, then excludes (hides) whatever you specify. Everything else is then shown, regardless of whether it is shown or hidden. “SHOW” on page 181 provides an alternative means of displaying and hiding items.

Return codes

0	Operation successful
2	Keyword truncated
5	Invalid information type keyword

HISTORY

Allows review of the history information maintained by IDF when the PATH or PATHFILE option is set.

```
►►──HISTory──────────────────────────────────────────────────────────────►◄
```

IDF maintains a history of the last 1,023 instructions executed, and the order in which they were executed. HISTORY is only valid when the PATH or PATHFILE option is ON.

When the HISTORY command is executed, an arrow is placed beside the last instruction that was executed. This arrow appears on the right side of the screen, just to the left of the instruction count in the first Disassembly window. The arrow is only displayed against the disassembled assembler code. If the Disassembly window is displaying source code, then you need to issue the SHOW DISASM command to see the arrow.

While the HISTORY arrow is shown in the window, the PREVIOUS and NEXT commands do not scroll the open windows on the screen. Instead, they move the arrow to the previously executed instruction or to the following instruction.

The display remains in this state until a DUMP, DISASM, RUN, or STEP command is executed.

This lets you review the series of instructions that lead to an incorrect result.

Return codes

0	Operation successful
6	PATH option is not ON

ICOUNT

Displays the number of instructions executed since the last ICOUNT command.

```
►►──ICOunt───────────────────────────────────────────────────────────────►◄
```

After the number is displayed, it is reset to zero. If the PATH option was not set ON, it is set on by the ICOUNT command.

Return codes

0	Instruction execution count successfully displayed

KWDSYN

Defines a synonym of an IDF keyword.

```
►►──KWDSYN──oldkwd──newkwd──────────────────────────────────────────────────────►◄
```

oldkwd
The existing IDF keyword. It may be abbreviated to the permitted minimum.

newkwd
The overriding synonym. It can have a length different from the IDF keyword it is a synonym of. Specify its minimum abbreviation by putting that part of the new keyword in upper case.

The synonym does not replace the IDF keyword, but it is recognized as the IDF keyword and given all its properties.

Examples

- The following command equates the keyword DUMP with the keyword DUMP. However, the new keyword has a minimum abbreviation of D, where the old one's minimum abbreviation was DUMP.

```
set kwdsyn dump Dump
```

- The following command replaces the keyword DISASM with SOurce.

```
SET KWDSYN DIS SOurce
```

LANGUAGE +

Scrolls information in an LSM Information window.

```
►►──LANguage──┬────────┬──┬─ + ─┬──┬─────────────────────────┬──────────────────►◄
              └─window─┘  └─ - ─┘  └─scroll-number-of-lines─┘
```

window
A LSM Information window. Select by a Window Specification or by placing the cursor in the window. If a Window Specification is not present and the cursor is not in a LSM Information window, uses the first LSM Information window.

+ Scroll forward.

- Scroll backwards.

scroll-number-of-lines
The number of lines to scroll (forwards or backwards).

The default value is the scroll amount specified by the last LANGUAGE SCROLL command (see "LANGUAGE SCROLL" on page 133).

Many IDF commands provide user feedback in the form of messages which are displayed in a dedicated area above the IDF command line. Some messages provide sufficient information, for example, when an error occurs, but other commands need more space for displays, for example, VARIABLE, STRUCTURE, and LANGUAGE STATUS.

To handle this situation, IDF provides many LSM Information windows. The first LSM Information window opens automatically when needed, and you can open more with the IDF OPEN command.

There may still not be enough screen lines to display all of the information.

In this situation, IDF provides a “More: + –” indicator in the upper right corner of each LSM Information window. You can now scroll the LSM Information window in IDF with the LANGUAGE + and LANGUAGE – commands.

LANGUAGE + scrolls the LSM display forward, and reissues the current LSM command. When the last line is displayed, the message is not issued, and further forward scrolling does not occur.

A LANGUAGE + command is automatically issued by IDF when the cursor is within the LSM Information window, and the NEXT IDF command is issued.

A LANGUAGE – command is automatically issued by IDF when the cursor is within the LSM Information window, and the PREVIOUS IDF command is issued.

Return codes

0	Operation successful
2	Keyword truncated
5	Arguments are invalid

LANGUAGE COLOR

Controls the color used to display the source code.

```
►►──LANguage──┬─COLor──┬──┬─BLUe──────┬──────────────────────────────────────────────►◄
              └─COLour─┘  ├─RED───────┤
                          ├─PINk──────┤
                          ├─GREen─────┤
                          ├─TURquoise─┤
                          ├─YELlow────┤
                          └─WHIte─────┘
```

The available colors are:
- BLUE
- RED
- PINK
- GREEN
- TURQUOISE
- YELLOW
- WHITE

Initially, source code is displayed in the color you have specified for IDF display of text data.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
5	Not valid color keyword

LANGUAGE COMMENTS

Enables or disables the block comment display.

```
►►──LANguage──COMments──┬─ON──┬──────────────────────────────────────────────►◄
                        └─OFF─┘
```

`ON` Block comment display enabled.

`OFF`
Block comment display disabled.

LANGUAGE DEBUG

Displays line-mode debug information for all interactions on the LSM interface.

```
►►──LANguage──DEBug──qualifiers──────────────────────────────────────────────►◄
```

This command is used for debugging the IDF Language Support internal interfaces.

It is not intended as a general user interface.

`qualifiers`
Options that limit this information to specific categories.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

LANGUAGE DECLARES

Enables or disables declare display.

```
►►──LANguage──┬─DEClares─┬──┬─ON──┬──────────────────────────────────────────►◄
              └─DCL──────┘  └─OFF─┘
```

`ON` Declare display enabled.

`OFF`
Declare display disabled.

LANGUAGE DROP

Removes language extract data from storage, freeing the space for reuse by new extract files loaded by subsequent LANGUAGE LOAD commands.

```
►►──LANguage──DROp──┬─*─────────────────┬──────────────────────────────────────►◄
                    └─extract-file-name─┘
```

`*` All extract files in memory are removed from storage.

extract-file-name
: The extract file name specifying the loaded compiles to be removed from storage. All compiles with the extract file name matching this value are removed.

  If no extract data matches the argument, an error message is displayed.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
5	Arguments are invalid
28	The specified extract file was not loaded with LANGUAGE LOAD

LANGUAGE LOAD

Loads an extract file, optionally associating it with an executable module.

```
                                              ┌─ASMLANGX──┐  ┌─*─────────┐
►►──LANguage──LOAd──extract-file-name──┴──────────┴──┴───────────┴──┬──────────────────────────┬──►◄
                                              └─file-type─┘  └─file-mode─┘  └─(──MODUle──module-name─┘
```

extract-file-name
: The extract file that is loaded.
  - On CMS, the file name (FN) of the extract file created by ASMLANGX.
  - On z/OS, the PDS member name of the extract file created by ASMLANGX. For z/OS sequential files (DSORG(PS)), a dummy name must be specified, but is ignored.
  - On z/VSE, the librarian member name of the extract file created by ASMLANGX.

  If not specified (the file type default accepted) IDF uses the entries in the XPATH to locate the extract file.

ASMLANGX
: The default file type for extract file (normally not specified).

file-type
: The file type of the extract file.
  - On CMS, the file type (FT) of the extract file created by ASMLANGX.
  - On z/OS, the DD name allocated to the extract file created by ASMLANGX.
  - On z/VSE, not used.

file-mode
: The file mode of the extract file.

- On CMS, the FM of the extract file created by ASMLANGX.
- On z/OS, not used, ignored if specified.
- On z/VSE, not used, ignored if specified.

*****
: Indicates that all disks are to be searched in order.

MODULE
Indicates that a module name is being specified.

module-name
The module with which to associate the extract file.

Multiple instances of the same extract file may be loaded as long as each specifies a different *module-name*.

If a module is *not* specified:

- If the extract file contains information that needs load-time resolution the module defaults to the qualified target module.
- Otherwise, the extract file is generic, where it is freely associated with any relevant code sections in all MODULES.

The LANGUAGE STATUS command (see "LANGUAGE STATUS" on page 133) shows the module name specified when an extract file is loaded.

The MAP command (see "MAP" on page 142) shows the association between extract files and program code sections.

Return codes

0	Operation successful.
1	Missing keyword.
2	Keyword truncated.
3	Keyword unknown.
28	File not found.
29	File was already loaded with the same module specification.
201	The input file is from a version of ASMLANGX that is not supported by the LSM.
202	Duplicate or invalid scope record in input file.
203	More source statements in file than indicated by the scope record.
204	Duplicate or invalid number-of-symbol record in the input file.
205	Symbol id number larger than maximum symbol number was encountered in a symbol record.
206	Symbol id number larger than maximum symbol number was encountered in an array record.
207	File contains no records of a known format.
208	Duplicate or invalid check record in the input file.
209	Duplicate or invalid address expression record in the input file.
210	Duplicate or invalid structure layout record in the input file.
211	Symbol id number larger than maximum symbol number was encountered in a respecify record.
212	Address expression id number larger than maximum address expression number or symbol id number larger than the maximum symbol number was encountered in an address expression record.
213	Address expression id number larger than maximum address expression number or symbol id number larger than the maximum symbol number was encountered in an address expression respecify record.
214	Symbol id number larger than maximum symbol number was encountered in a symbol optimization record.
215	Symbol id number larger than maximum symbol number was encountered in a structure layout record.
216	Errors were encountered unpacking a source record.
217	Base debugger (IDF) version is too low to support the required function.
218	Module private-code CSECT was not found. The functions in the extract file cannot be located in the IDF symbol table.

219	Scope ID record present, but it was not preceded by a Scope record.
220	Symbol id number larger than maximum symbol number was encountered in a scope ID record.
221	Duplicate or invalid CSECT record in the input file.
222	CSECT record expected but not found in the input file.
223	CSECT id number larger than maximum CSECT number was encountered in a PC (Private Code) lookup record.
224	CSECT id number larger than maximum CSECT number was encountered in a source record.
225	CSECT id number larger than maximum CSECT number was encountered in a symbol record.
226	Symbol id number larger than maximum symbol number was encountered in a constant record.
227	Error occurred building the statement offset table.
237	Duplicate or invalid On Event Map record in the input file.
238	Duplicate or invalid Secondary Entry Point record in the input file.
239	Symbol id number larger than maximum symbol number was encountered in a Secondary Entry Point record.
240	Duplicate or invalid External Symbol List record in the input file.
241	Symbol id number larger than maximum symbol number was encountered in an External Symbol List record.
242	Duplicate or invalid Symbol Display Format record in the input file.
243	Symbol id number larger than maximum symbol number was encountered in a Symbol Display Format record.
244	Duplicate or invalid Class Layout record in the input file.
245	Symbol id number larger than maximum symbol number was encountered in a Class Layout record.
246	Duplicate or invalid Class Hierarchy record in the input file.
247	Symbol id number larger than maximum symbol number was encountered in a Class Hierarchy record.
248	Symbol id number larger than maximum symbol number was encountered in an OverLoaded Symbol record.
249	Duplicate or invalid Linkage Convention record in the input file.
250	Symbol id number larger than maximum symbol number was encountered in a Linkage Convention record.
251	Symbol id number larger than maximum symbol number was encountered in a Parameter List record.
252	Duplicate or invalid Compile Information record in the input file.
253	Duplicate or invalid OverLoaded Entry Stmt record in the input file.
254	Symbol id number larger than maximum symbol number was encountered in an Overloaded Entry Stmt record.

1xxx

Error occurred during scan of the extract file.

The values for xxx are:

1yy	yy is the RC from File_Open
2yy	yy is the RC from File_Read
4yy	yy is the RC from File_Point
5yy	yy is the RC from Mem_Allocate
6yy	yy is the RC from Mem_Free
7yy	yy is the RC from File_Close
8yy	yy is the RC from File_Note

LANGUAGE MACROS

Enables and disables the display of assembler source generated by macros.

```
►►──LANguage──MACros──┬─ON──┬──────────────────────────────────────────────►◄
                      └─OFF─┘
```

ON The display of assembler source generated by macros is enabled.

OFF
The display of assembler source generated by macros is disabled. This is the initial setting.

For more details see the command description of `SHOW` and `HIDE`.

LANGUAGE OPTIONS

Displays the current value for each IDF Language Support setting.

```
►►──LANguage──OPTions──────────────────────────────────────────────────────►◄
```

Displays the current value of the following options or settings:
- Show
- Debug
- Scroll
- Detail
- Major
- Settings save stack nesting level
- Checks
- LSM stem
- EXLIMIT
- XPATH
- Bit variable format
- Char variable format
- Fixed variable format
- Float variable format
- Packed variable format
- Zoned variable format

The OPTIONS command is meant primarily for debugging situations, or for checking that IDF understood your commands.

The LSM Information window containing the LANGUAGE OPTIONS display closes if you issue a second LANGUAGE OPTIONS command.

Return codes

0	Operation successful

LANGUAGE SCROLL

Sets the default for the number of screen lines by which an LSM Information window is scrolled when the LANGUAGE command is issued.

```
►►──LANguage──SCRoll──┬─0───────────────┬──────────────────────────────────────────►◄
                      ├─number-of-lines─┤
                      └─┬─MAX─┬─────────┘
                        └─*───┘
```

0 Scrolling is disabled.

number-of-lines
Scrolling is by this number of lines

MAX | *
Scrolling is by the maximum number of lines (the current size of the LSM Information window)

The current scroll amount value is displayed as part of the LANGUAGE OPTIONS command information.

Return codes

0	Operation successful
2	Keyword truncated
5	Arguments are invalid

The current scroll amount value is displayed as part of the LANGUAGE OPTIONS command information.

LANGUAGE STATUS

Displays information about the extract files that were loaded with LANGUAGE LOAD commands.

```
                      ┌─*──────────────────────┐
►►──LANguage──STAtus──┼────────────────────────┼──────────────────────────────────────►◄
                      │  ┌─;────────────────┐  │
                      │  ▼                  │  │
                      └───extract-file-name─┴──┘
```

***** Information about all the currently loaded (with LANGUAGE LOAD) language files is displayed.

extract-file-name
Information about the compiles contained in this extract file is displayed.

The LSM Information window containing the LANGUAGE STATUS display is closed if you issue a LANGUAGE STATUS command without operands and the current display is for the same command with a null argument.

The LANGUAGE STATUS command is meant primarily for debugging situations, or for verifying that IDF understood your commands.

To display the current value of the IDF Language Support settings, use the LANGUAGE OPTIONS command.

Return codes

0	Operation successful
28	The specified extract file was not loaded with LANGUAGE LOAD.

LANGUAGE STEM

Alters the name of the REXX stemmed variable array which is used to return information to an IDF macro as a result of all subsequent EXTRACT LANGUAGE commands, as well as a number of other EXTRACT commands.

```
                    ┌─LSM───────┐
►►──LANguage──STEM──┴─stem-name─┴───────────────────────────────────────────►◄
```

LSM
The default value of the REXX stem name.

stem-name
The REXX stem name. It must be between 1 and 8 characters, with the first character alphabetic.

The current LANGUAGE STEM setting can be displayed with the LANGUAGE OPTIONS command, and retrieved with the EXTRACT LANGUAGE STEM command.

For more information, see Chapter 15, "The EXTRACT command," on page 223.

Return codes

0	Operation successful
2	Keyword truncated
5	Arguments are invalid

LANGUAGE VERSION

Displays the IDF Language Support version information.

```
►►──LANguage──VERsion───────────────────────────────────────────────────────►◄
```

The information is in the form:

```
ASMLANG Vn.Rn.nn (generated ccyy.ddd hh:mm)
```

Return codes

0	Operation successful

LANGUAGE XPATH (CMS and z/OS)

Defines the extract file search path file type (TSO DD name) information.

```
                    ┌─ASMLANGX──────────────────┐
►►──LANguage──XPATH─┼───────────────────────────┼──────────────────────────────►◄
                    │  ┌─separator──────────┐   │
                    │  ↓               (1)  │   │
                    └───extract-file-type───┴───┘
```

Notes:

1 Up to 10 entries can be specified.

ASMLANGX
: The default extract file search path.

extract-file-type
: The extract file search path.

separator
: A comma, blank, or semicolon. Separates extract file types.

The following characters are allowed in XPATH entries:
- A–Z (or a–z)
- @, #, _, $, –, +
- 0–9 (*not* allowed as first character of an entry)

This information is used to locate extract files for which the extract file type (z/OS DD name) was not explicitly specified. The XPATH entries are searched in the supplied order.

The LANGUAGE OPTIONS command displays the current XPATH information.

The ability to define the extract file search path gives you full control of the extract file loading process. It is easy to support the coexistence of many versions of a target program.

Return codes

0	Operation successful
2	Keyword truncated
5	Arguments are invalid

LAST

Displays source code starting at the highest available address.

```
►►──LASt─┬────────┬────────────────────────────────────────────────────────────►◄
         └─window─┘
```

window
: A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Disassembly window, uses the first Disassembly window.

Return codes

0	Operation successful

LASTMSG

Displays the last two messages issued in the message areas.

```
►►──LASTMsg──────────────────────────────────────────────────────────────────────►◄
```

Return codes

0	Operation successful

LEFT

Scrolls an open window left.

```
►►──LEFt──┬────────┬──┬───────────────────┬──────────────────────────────────────►◄
          └─window─┘  └─number-of-columns─┘
```

window
: A window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a scrollable window, then all scrollable windows are scrolled.

number-of-columns
: The number of columns by which the window is to be scrolled. This may be specified as:

  - An integer
  - An integer prefixed by a +
  - An integer prefixed by a -, specifying the number of screen columns by which the window is to be scrolled to the *right*.

  If omitted, each window to be scrolled is scrolled by the current number of data columns for that window.

This command is normally only meaningful when a Disassembly window is open. However, you can also scroll windows made smaller with the SIZE command.

Return codes

0	Operation successful
5	Arguments are invalid
6	Command issued when it has no meaning

LIBE (CMS and z/OS)

Controls the source of the target program which IDF is to load.

```
►►──LIBE──file-name──────────────────────────────────────────────────────────►◄
```

file-name
The file to be loaded.

z/VM

- LIBE indicates that the target program should be loaded from an OS-style LOADLIB rather than from the usual CMS-style MODULE file.

  file-name
  The file name of the LOADLIB to be used.

  If this file name is not present in the list of files declared with the CMS GLOBAL LOADLIB command, it is added as the first library.

  $ Indicates that the LOADLIB was declared through the CMS GLOBAL LOADLIB command.

z/OS

- LIBE is used to explicitly control the loading of the target Load Module.
- A single parameter is needed. It may be either:
  - The DD name of the PDS containing the Load Module.
  - $ to indicate that standard OS Load Module search order should be used.

The LIBE command may only be issued from the PROFILE macro, before loading the target program into storage.

Examples

```
SET LIBE $
SET LIBE mylib
```

LOAD

Loads the target program and associated symbol information into memory.

```
►►──LOAd──┬───────────────────────┬──────────────────────────────────────────────►◄
          ├─MODule────────────────┤
          └─SYMbols──┤ File Info ├┘

File Info:

              ┌─MAP──────────┐
├──┬─cms-fn─┴─ft─┬──────┬─┴──┬──┬─────────────────────────┬──────────────────────────┤
   │             └─fm───┘     │  └─(──MODUle──module-name─┘
   ├─pds-member──┬─────────┬─┤
   │             └─dd-name─┘ │
   └─phasename───────────────┘
```

MODULE
Keyword indicating module-only load.

Only the target module is loaded into storage.

This option may only be used in the PROFILE macro. For more details see "Command restrictions related to PROFILE execution" on page 204.

SYMBOLS
Keyword indicating symbols-only load. Does not apply to z/OS.

Only the module symbols are loaded into storage. The following parameters provide more information.

cms-fn
CMS file name

MAP
Default CMS file type

ft CMS file type

fm CMS file mode

pds-member
z/OS PDS member name.

dd-name
z/OS DD name. If omitted, IDF looks in the ISPLLIB or STEPLIB if present. If the LIBE option was used, then that DD name is used as the default. IDF does not look in LPALIB or the z/OS link list.

phasename
The z/VSE phase map, as generated by ASMLKEDT.

MODULE
Keyword to indicate that a module name is supplied. If not supplied, then symbols are loaded for the qualified module.

module-name
Module for which symbols are loaded.

If LOAD MODULE or LOAD SYMBOLS is issued by the PROFILE macro, but not both, IDF makes sure that both functions are performed after the profile completes.

For example, if the PROFILE issues LOAD SYMBOLS but does not issue LOAD MODULE, IDF loads the module anyway after the profile is completed.

You can stop IDF from loading the module with the MODULE command.

If no parameters are supplied with this command (the command is just “LOAD”) then both the target module and its associated symbols are loaded into storage. This may only be used in PROFILE macro. For more details see “Command restrictions related to PROFILE execution” on page 204.

Return codes

0 Target program loaded successfully

Other An error occurred, see “Message numbers and severity levels” on page 263 for return codes.

Notes for LOAD SYMBOLS for CMS

The underlying assumption for the IDF processing which loads symbols is that the symbol information is found as INVALID CARD images in the LOAD MAP file, interspersed with entrypoint addresses. You can use the LOAD SYMBOLS command to load symbol information from a TEXT file rather than a renamed LOAD MAP if you wish, but you have to perform some more setup for this case.

When loading symbol information directly from a TEXT file, you must first load all of the applicable TEXT files by means of the LOAD SYMBOLS command, then you must issue a SYMBOL command to define the offset-within-module for each CSECT.

An example profile to perform this kind of operation follows, but you probably want something more elaborate to prevent retyping the addresses within the profile each time you rebuild the program:

```
/* load symbols directly from TEXT files */
'LOAD SYMBOLS ASMPARM TEXT'
'LOAD SYMBOLS ASMSCAF TEXT'
'SET SYMBOL (ASMPARM) ASMPARM  00000000 00000000 00000060 E F 01'
'SET SYMBOL (ASMSCAF) ASMSCAF  00000000 00000060 00000400 E F 01'
'SET SYMBOL (ASMSCAF) ASMSUBCM 0000028E 000002EE 00000000 E F 24'
```

Figure 23. Loading symbols directly from TEXT files

The records in the LOAD MAP file which IDF uses to determine the start addresses of various CSECTs have the following format (in some cases you may want or need to build the map manually):

Col 1 Blank

Col 2-9
Symbol name (upper case)

Col 10 Blank

Col 11-12
If the symbol marks the start of a CSECT, these columns should contain the characters SD, otherwise they should be blank.

Col 13 Blank

Col 14-19
The hexadecimal address associated with this symbol.

Col 20-*
Blank

LOCATE

Locates a string and displays the section of code where it occurs.

```
►►──┬────────┬──┬────────┬──┬─────┬──/──string──┬───┬──────────────────────────────►◄
    └─Locate─┘  └─window─┘  └─ - ─┘             └─/─┘
```

window
: A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted, and the cursor is not in a Disassembly window, uses the first Disassembly window.

-
: Search direction is up, rather than down.

string
: The string to be searched for. Include the trailing delimiter "/" when the string has trailing blanks.

Format is essentially the same as the XEDIT LOCATE command. The search begins at the first source line shown on the screen; the target code, if found, is displayed at the top of the screen.

Examples

```
loc /procedure

loc -/end;

l /declare /

/init(

/first use/

-/gobackward

loc =3 /window 3/
```

Return codes

0	Operation successful
1	Missing search string
6	No extract data files containing source code were loaded with LANGUAGE LOAD.

LOCATION

Sets main storage to the contents of REXX variable MEMAREA.

```
►►──LOCation──storage-start-address──────────────────────────────────────────────►◄
```

location-start-address
: An expression specifying the storage start address.

  If the expression contains an Access Register then the storage that is modified is in the dataspace identified by the ALET in the referenced Access Register.

The SET LOCATION command lets you modify storage within your program's defined limits. If the TRACEALL or RISK option is ON, you may be able to modify storage beyond the defined limits.

For more details on your program's defined limits and how to change them, see (for CMS) “Your program's defined limits” on page 56, (for z/OS) “Your program's defined limits” on page 49 and (for z/VSE) “Your program's defined limits” on page 62.

Examples

```
SET LOCATION plist
SET LOCATION 0(R1)
SET LOCATION 0(AR2)
```

REXX variables read

MEMAREA
: New contents of the specified memory area

LOCATION ALET

Sets storage in a dataspace to the contents of REXX variable MEMAREA.

```
►►──LOCation──ALEt──access-link-entry-token──storage-start-address──────────────────────►◄
```

access-link-entry-token
: A token identifying the ALET. Must be a one to eight character hexadecimal value.

storage-start-address
: An expression resolving to a storage area start address within the dataspace.

You must be in an ESA environment for this command to work.

Examples

```
SET LOCATION ALET 10003 X'1000'
SET LOCATION ALET 10004 0(R1)
```

REXX variables

MEMAREA
: New contents of the specified memory area

MACRO

Issues an IDF macro.

```
►►──MACro──macro-name──┬──────────────────┬──────────────────────────────────────────────►◄
                       └─macro-parameters─┘
```

macro-name
: The name of the macro.

macro-parameters
: Arguments for the macro.

Execute the MACRO command by pressing a PF key, by typing on the command line when the COMMAND command is invoked, or by issuing as a command from another macro.

Return codes

If the macro is found, its return code is propagated to the caller.

The following return codes are generated by IDF:

-3	Macro not found
1	No macro name specified
2	Macro name exceeds eight characters
5	Arguments to macro exceed 79 characters in length

MAJOR

Enables and disables the display of data for the structure major component.

```
►►──MAJor──┬─ON──┬──────────────────────────────────────────────────────►◄
           └─OFF─┘
```

ON Enables the display of structure major component data.

OFF
Disables the display of structure major component data. This is the initial setting.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

MAP

Displays the location of modules and code sections known to IDF, and the name of any associated extract file.

```
                          ┌─*───────────────┐
►►──MAP──┬────────┬───────┼─────────────────┼───────────────────────────►◄
         └─window─┘       │ ┌─;───────────┐ │
                          └─▼─module-name─┴─┘
```

window
An LSM Information window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in an LSM Information window, uses the first LSM Information window.

<u>*</u> Displays information for all known modules.

module-name
The module for which information is displayed.

The LSM Information window containing the MAP display is closed if you issue a MAP without operands and the current display is for the same command with a null argument.

The module and code section entries are built (and possibly associated with an extract file) as part of processing the LANGUAGE LOAD command, as well as the DISASM command.

The MAP command is meant primarily for debugging situations, or for verifying that IDF understood your commands.

Return codes

0	Operation successful
28	The specified module was not defined to IDF

MAXIMIZE

Removes a window from the Minimized Windows Viewer and restores it to its previous position on the display screen.

```
►►──MAXimize──┬────────┬───────────────────────────────────────────────────────►◄
              └─window─┘
```

window
The window to maximize. Select by a Window Specification or by placing the cursor in the window name token in the Minimized Windows Viewer.

Return codes

0	Operation successful
1	No window selected

MINIMIZE

Removes a window from the IDF display, and places an entry representing the window in the Minimized Windows Viewer.

```
►►──MINimize──┬────────┬───────────────────────────────────────────────────────►◄
              └─window─┘
```

window
The window to be minimized. Select it by a Window Specification or by placing the cursor in the window.

Return codes

0	Operation successful
1	No window selected

MODE (CMS only)

Sets the file mode IDF uses for CMDLOG or MACROLOG and associated playback operations.

```
►►──MODE──file-mode──────────────────────────────────────────────────────►◄
```

file-mode
A CMS file mode specification.

Examples

```
SET MODE b
SET MODE A2
```

MODULE

Stops IDF loading an executable file into memory.

```
►►──MODUle───────────────────────────────────────────────────────────────►◄
```

MODULE intended for use by user profiles that need to perform unusual program loading. It can only be issued from an IDF macro or profile.

Examples

```
MODULE
```

MODULE

Defines the origin and length of a module that was loaded while the original target program was executing.

```
►►──MODUle──module-name──┬────────┬──────────────────────────────────────►◄
                         ├─CDE────┤
                         ├─NUCext─┤
                         └─TRANs──┘
```

module-name
The name of the module that is to have its origin and length defined.

Note: Unless the module is qualified by the QUALIFY command, address expressions that refer to a dynamically loaded module must prefix any symbols with the module and CSECT wrapped in parentheses (see "Address expressions" on page 80).

CDE
z/OS only.

Scan the Contents Dictionary Entries (CDEs) in the Job Pack Queue (list of modules loaded for the current job, the z/OS session) for an entry corresponding to *module-name*. If no suitable CDE was found, the search continues with a scan of the Link Pack Dictionary Entries (LPDEs) in the Link Pack Area (the list of pre-defined and resident modules).

If a matching CDE or LPDE is found, set up a module definition for *module-name* with appropriate module start address and length values.

NUCEXT
CMS Only.

Scan the CMS Nucleus Extension Subcommand control block chain and locate an entry for *module-name*.

If an entry is found, set up a module definition for *module-name* with appropriate module start address and length values.

TRANS
CMS Only.

Check the *module-name* MODULE file to make sure that it is a transient program (has only 1 record).

If *module-name* MODULE is a transient program, set up a module definition for *module-name* with a module start address of X'0E000' and a module length of X'FFF'.

The second argument is the module search type. It is not needed for z/VSE.

The origin and length are taken from system control blocks.

This command lets you debug many modules at once.

The new module definition is needed before symbols are loaded for modules loaded by the original target.

Examples

```
MODULE IDF CDE
MODULE IDFMAIN NUCEXT
```

MODULE BASE

Sets the origin of a module that was loaded while the original target program was executing.

```
►►──MODUle──module-name──BASe──module-start-address──────────────────────────────►◄
```

module-name
The name of the module that is to have its origin set.

Note: Unless the module is qualified by the QUALIFY command, address expressions that refer to a dynamically loaded module must prefix any symbols with the module and CSECT wrapped in parentheses (see "Address expressions" on page 80).

module-start-address
An expression which resolves to the start address value for module *module-name*.

This command lets you debug many modules at once.

The new module definition must be completed with a MODULE SIZE command before symbols are loaded.

See “BASE” on page 99 for information about how to set the base address of the target program specified when IDF was invoked.

Examples

```
MODULE ASMLANGX BASE 0(R0)
```

MODULE SIZE

Sets the length of a module that was loaded while the original target program was executing.

```
►►──MODUle──module-name──SIZe──module-length─────────────────────────────────────────────►◄
```

module-name
: The name of the module that is to have its length set.

  Note: Unless the module is qualified by the QUALIFY command, address expressions that refer to a dynamically loaded module must prefix any symbols with the module and CSECT wrapped in parentheses (see “Address expressions” on page 80).

module-length
: An expression which resolves to the length value for module *module-name*.

This command lets you debug many modules at once.

The new module definition must be completed with a MODULE BASE command before symbols are loaded.

Examples

```
MODULE ASMLANGX SIZE X'123000'
MODULE FDISK SIZE F'1024'
```

MOVE

Moves a window to a new location on the screen.

```
►►──MOVe──┬─window───┬─┤ Location ├──────────────────────────────────────────►◄
          ├─ADSTops──┤
          ├─AFPR─────┤
          ├─BREak────┤
          ├─DISasm───┤
          ├─DUMP─────┤
          ├─OPTions──┤
          ├─OREGs────┤
          ├─REGs─────┤
          ├─SKIPstep─┤
          ├─STAtus───┤
          └─LSMinfo──┘
```

Location:

```
├──┬──────────────────────────────────────┬────────────────────────────────────┤
   └─┬─┬─────┬─row─┬─┬─┬─────┬─column─┬─┘
     │ ├─ + ─┤     │ │ ├─ + ─┤        │
     │ └─ - ─┘     │ │ └─ - ─┘        │
     ├─*───────────┤ ├─*──────────────┤
     └─┬─STANdard─┬┘ └─┬─STANdard─┬───┘
       └─STD──────┘    └─STD──────┘
```

WINDOW
The window to be moved. Select by a Window Specification, or by placing the cursor in the window.

ADSTOPS
The AdStops window.

AFPR
The Additional Floating-Point Registers window.

BREAK
The Break window.

DISASM
The Disassembly window with the lowest window id.

DUMP
The Dump window with the lowest window id.

OPTIONS
The Options window.

OREGS
The Old Registers window.

REGS
The Current Registers window.

SKIPSTEP
The Skipped Subroutines window.

STATUS
The Target Status window.

LSMINFO
The LSM Information window with the lowest window id.

row
The screen row to which the window is to be moved.

This may be specified by:

- an integer. This is the screen row for the upper left corner of the window. The screen rows are numbered from the top. The top row is row 1.
- an integer prefixed by a +. This is the number of screen rows by which the window is moved down on the screen.
- an integer prefixed by a -. This is the number of screen rows by which the window is moved up on the screen.
- * The window is kept at the current row. Use this when you want to change the column, but not the row.
- STANdard | STD. This indicates that the standard window row and column position for this window type should be used.

If this parameter is omitted, the position of the cursor on the screen is used as the new location of the upper left corner of the window.

column
The screen column to which the window is moved.

This may be specified as:

- an integer. This is the screen column for the left side of the window. The screen columns are numbered from the left. The leftmost column is 1.
- an integer prefixed by a +. This is the number of screen columns by which the window is to be moved to the right on the screen.
- an integer prefixed by a -. This is the number of screen columns by which the window is to be moved to the left on the screen.
- * The window is kept at the current column. Use this when you want to change the row, but not the column.
- STANdard | STD. This indicates that the standard window row and column position for this window type should be used.

If this parameter is omitted, and the row parameter is specified, the current screen column for this window is maintained.

The MOVE command lets you place open windows in explicit places on the screen. Any window can overlap any other window. However, none of the windows can overlap the Command window. Also, a window cannot be placed such that some of the window is off the screen. If you try this, IDF repositions the window so that all of it stays on the screen.

Examples

- To move the window 03 origin to row 5 and column 20:
```
MOVE =03 5 20
```
- To move the window in which the cursor is placed to row 8 and column 1:
```
MOVE 8 1
```
- To move the window 01 origin to the current cursor position:
```
MOVE =01
```

Return codes

0	Operation successful
1	Window name or Row location missing
2	Operand longer than eight characters
5	Window name invalid or syntax or other error in expression

MPACK

Returns extract data storage AREAs that do not hold extract data to the operating system free storage pool.

```
►►──MPAck──────────────────────────────────────────────────────────►◄
```

The MPACK command is meant primarily to help when free storage is at a premium.

Return codes

0	Operation successful
Other	Error occurred while packing extract data storage AREAs. This is probably caused by an overlay of the extract data AREA control information by an errant application program under test.

MRUN

Executes target program instructions until the next event occurs.

```
►►──MRUn───────────────────────────────────────────────────────────►◄
```

The MRUN command is different to RUN, because the target program is executed *before* control returns to the issuing macro. When RUN is issued from a macro, it has no immediate effect, being processed *after* the macro exits.

After issuing MRUN, the macro can use EXTRACT EVENT to determine what kind of event occurred in the target program.

z/VM

You should ensure that the MRUN command is issued through the LPSW Fastpath addressing environment, which is the default environment established when the IDF macro is entered. Using this interface eliminates any SVC linkages between REXX and IDF, so that IDF can provide optimum flexibility in what the target program can itself execute under MRUN. "REXX linkage considerations" on page 208 provides information concerning the available methods for invoking IDF commands from REXX.

If instead the address ASM environment is used to invoke the MRUN command, REXX uses a CMSCALL (SVC 204) to invoke the MRUN command. This SVC linkage introduces some potential problems in the execution of the target program. These are described in "MRUN invoked through address ASM on CMS" on page 150.

Return codes

0	Operation successful
6	The target program is not yet loaded

MRUN invoked through address ASM on CMS

If the address ASM environment is used to invoke the MRUN command (or MSTEP, or any other command which causes immediate execution of the target program) on CMS, then REXX uses a CMSCALL (SVC 204) to invoke the MRUN command. This SVC linkage introduces some potential problems in the execution of the target program:

1. If the target program obtains GETMAIN storage during execution under the MRUN command, that storage may be implicitly FREEMAIN'ed again, without warning. This occurs because of the default CMS setting of STORECLR ENDSVC, and the fact that a CMSCALL/CMSRET pair was used in the REXX linkage to the MRUN. CMS is working as designed by implicitly freeing any GETMAIN storage that was obtained between the CMSCALL and CMSRET. This can of course lead to unpredictable behavior of the program's further execution.

   The implicit FREEMAIN can be avoided by a CMS SET STORECLR ENDCMD setting, but this in turn introduces another consideration of its own: CMS then honors any STRINIT request. Any MODULE file that is generated with the STR option causes a STRINIT request when that MODULE is executed.

   So switching to STORECLR ENDCMD is not a good solution. It is best to leave the default STORECLR ENDSVC. (IDF does not itself alter the current STORECLR setting.) The only complete solution is to make sure that the LPSW Fastpath linkage is used to invoke the MRUN command.
2. If the target program does a CMSCALL to another program, and an event occurs to signal completion of the MRUN, then CMSCALL synchronization is destroyed by IDF's attempt to CMSRET back to REXX to complete the MRUN. CMS treats each CMSRET as being paired to the most recent CMSCALL. The most recent CMSCALL in the above scenario is the one from the target program execution. So instead of returning to the MRUN within REXX as intended, this CMSRET actually returns to the next instruction after the target program's CMSCALL.

   IDF cannot compensate for this CMSCALL/CMSRET linkage problem. All it can do is to *detect* when it is about to happen. This is done by recording the current SVC nesting depth at entry to its address ASM command handling, and checking that the nesting is the same just before it returns from that command handling. If not, another SVC depth was created (or destroyed) while executing the target program, and CMS is hopelessly confused if IDF returns. So instead IDF issues a series of messages about the situation, telling you to consult *this* section of the documentation for details. A re-IPL of CMS is needed afterwards.

   Exactly the same situation occurs when using LINK or SVC 202 linkage to another program instead of CMSCALL. Again, IDF detects a change in the CMS SVC nesting depth, issues its messages, and needs a re-IPL.
3. If the target program does a DMSKEY NUCLEUS, DMSKEY USER, or DMSKEY LASTUSER which does not specify the NOSTACK option, and an event occurs to signal completion of the MRUN before the corresponding DMSKEY RESET in the target program, then a CMS ABEND occurs, since the change of key appears to have occurred from within IDF and is not restored before returning to the calling function, in this case REXX.

These considerations apply only to an address ASM invocation on CMS. They are of no concern if you use the default LPSW Fastpath interface to invoke the command. With no SVC linkage used in that path, each of these potential problems is bypassed.

MSG

Places text in the next available Command window message display line.

```
►►──MSG──message-text──────────────────────────────────────────────────►◄
```

message-text
The message text.

If all message display lines are occupied, the current messages scroll up and the new message appears in the bottom display line.

Examples

```
SET MSG This is a message
```

MSGID (CMS and z/OS)

Controls the display of the message identifier.

```
►►──MSGId───┬─ON──┬──────────────────────────────────────────────────────────►◄
            └─OFF─┘
```

ON The message identifier is displayed.

OFF
The message identifier is not displayed.

The initial setting is based on the execution environment and z/OS or OS runtime settings:

- Under CMS, the messages respect the CMS EMSG setting:
  ON The MSGID option is initially ON
  TEXT The MSGID option is initially OFF
  OFF The MSGID option is initially ON
- Under TSO, the messages respect the TSO MSGID setting:
  MSGID
  The MSGID option is initially ON
  NOMSGID
  The MSGID option is initially OFF
- Under z/VSE, the MSGID option is always ON.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

MSGMODE

Controls the display of messages during macro execution.

```
►►──MSGMode───┬─ON──┬────────────────────────────────────────────────────────►◄
              └─OFF─┘
```

ON Enable the display of all messages.

This is the initial setting.

OFF
Disable the display of specific messages.

The message filtering is dependent on the message severity level. Only I (Informational), W (Warning) and E (Error) messages are suppressed by MSGMODE OFF. All other types of messages are not affected.

When some IDF commands are issued through PF keys, or a command typed on the Command window command line, appropriate status and informational messages are displayed.

During the execution of a macro, these messages may become rather lengthy, hence this command to give you control.

When macro execution is complete, the MSGMODE setting is returned to the default ON value.

The most recent 10 messages, including those suppressed by MSGMODE OFF, may be retrieved with the EXTRACT LASTMSG command. See “LASTMSG” on page 236 for details.

The QUIETLY prefix command may be used to override the current MSGMODE setting, and suppress messages (as if MSGMODE was OFF) for the invocation of a single command. See “QUIETLY” on page 166 for details.

Examples

Since this is one of the settings which is saved in the status stack, a useful sequence within a macro is:

```
'PRESERVE'      /* Save current settings in stack       */
'MSGMODE OFF'   /* No messages displayed                */
 :
                    /* Quietly perform macro functions       */
'RESTORE'       /* Restore settings, including MSGMODE   */
```

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

MSTATUS

Opens or closes (toggles) the MStatus window.

```
►►──MSTAtus──────────────────────────────────────────────────────────────►◄
```

This window displays information about the storage used to contain the extract data information that was loaded with LANGUAGE LOAD commands. This includes:

- number of compile areas
- extract data storage consumption (total, direct, pooled)
- extract data storage pool utilization, including the number of AREAs in the pool which are unused

The MSTATUS command helps when free storage is at a premium.

When extract data file information is removed from storage with LANGUAGE DROP, the storage AREAs that held the extract data are kept for use by later LANGUAGE LOAD commands. The MPACK command returns storage AREAs that no longer hold extract data back to the operating system free storage pool.

Return codes

0	Operation successful

MSTEP

Executes the next target program instruction.

```
►►──MSTep──────────────────────────────────────────────────────────────►◄
```

The MSTEP command is different to STEP because the target program is executed *before* control returns to the issuing macro. When STEP is issued from a macro, it has no immediate effect, being processed *after* the macro exits.

After issuing MSTEP, the macro can use EXTRACT EVENT to determine what kind of event occurred in the target program. In general this indicates that a STEP has occurred, but it may also indicate some other condition (for example, a program check).

z/VM You should ensure that the MSTEP command is issued through the LPSW Fastpath addressing environment, which is the default environment established when the IDF macro is entered. Using this interface eliminates any SVC linkages between REXX and IDF, so that IDF can provide optimum flexibility in what the target program can itself execute under MSTEP. See the usage notes under “MRUN” on page 149 for further details.

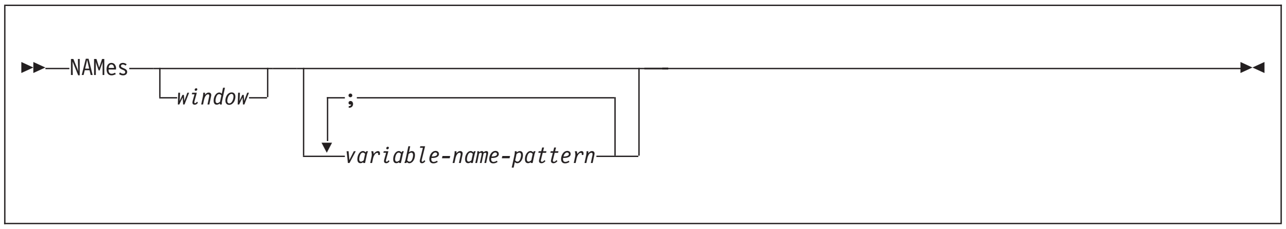
1. If you have subroutines within the program which you do not want to single-step through, use the SKIPSTEP command. The SKIPSTEP command causes IDF to skip single stepping when it comes to a subroutine call to a subroutine that was added to the list of subroutines being skipped. For the purposes of single-stepping, the skipped subroutine is treated as one instruction, the subroutine call instruction itself. If a breakpoint or a watchpoint whose condition is true is placed within the execution path of the subroutine being skipped, execution stops at that breakpoint or watchpoint.
2. If the STOPNOP option is OFF or the NOSTOPNP option is ON, then IDF does not stop on NOP and NOPR instructions that follow BAL, BALR, BAS, and BASR instructions.

Return codes

0	Operation successful
6	The target program is not yet loaded

NAMES

Display the names of all variables that match a particular pattern and are eligible for display.



window
: An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in an LSM Information window, uses or opens the first LSM Information window.

variable-name-pattern
: A variable name matching pattern.

  If this parameter is not supplied, then the names of all variables which are eligible for display are shown. Supplied variable name matching patterns define a subset of the eligible names. A variable name is displayed if it matches one variable pattern name.

  The variable name matching pattern rules are:
  - ? matches a single variable name character
  - % matches zero or more variable name characters
  - other characters represent themselves
  - the search is *not* case sensitive

The variable name information display persists until:
- A NAMES command without arguments is issued
- The window is closed with a CLOSE command.
- Another IDF Language command such as VARIABLE, STRUCTURE, ARRAY, TYPE, CALLERS, PLOCATES, LANGUAGE STATUS, or MAP is issued. These commands update the LSM Information window with new information
- The target program completes execution
- Target program execution progresses beyond the variable's defined scope

Examples

If `name`
: lists:

```
301char
ab30xcc
aptr
bptr001
ctra
ctrd
i
ii
mstidx
slvidx
title
```

Then `NAME`
: lists:

```
aptr
bptr001
```

and NAME
lists:

```
301char
ab30xcc
```

and NAME
lists:

```
ctra
ctrd
i
mstidx
slvidx
```

NEXT

Scrolls windows forward.

```
►►──┬─NEXt─┬──┬────────┬──────────────────────────────────────────────►◄
    └─Down─┘  └─window─┘
```

window
The window to be scrolled. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a scrollable window, then all open scrollable windows are scrolled.

If the HISTORY command was executed, this command has an alternate meaning, see “HISTORY” on page 125 for explanation.

This command works only on a scrollable window, which is any window made smaller with the SIZE command, or an open:
- Break window
- Dump window
- Disassembly window
- LSM Information window
- Options window
- Skipped Subroutines window
- Target Status window

Return codes

0	Operation successful
6	Command issued when it has no meaning

OFFSET

Sets or queries the current offset value.

```
►►──OFFSet──┬─────────┬───────────────────────────────────────────────►◄
            └─address─┘
```

address
The new value of the offset.

If not provided, displays the current value of the offset.

The OFFSET option must be ON to obtain a window display that shows dump or disassembly addresses in terms of the offset.

For information about specifying arguments in terms of the offset, see “Address expressions” on page 80. Arguments may be specified in terms of the offset regardless of the setting of the OFFSET option.

The initial value of the offset is the same as the program's base address (origin in memory).

If an address is provided as part of a command issued by a macro, that address is used in preference to any data typed on the screen.

See also “SET OFFSET” on page 177.

Return codes

0	Operation successful
5	Syntax or other error in expression
6	Command issued when it has no meaning

OPEN

Opens a Disassembly window, a Dump window, or an LSM Information window.

```
►►──OPEn──┬─ARRay─────┬──command-parameters──────────────────────────────────►◄
          ├─CALlers───┤
          ├─DISasm────┤
          ├─DUMP──────┤
          ├─LANguage──┤
          ├─MAP───────┤
          ├─STRucture─┤
          ├─TYPe──────┤
          ├─UNIon─────┤
          └─VARiable──┘
```

ARRAY
Open an LSM Information window with an array element display.

CALLERS
Open an LSM Information window with a program call hierarchy display.

DISASM
Open a Disassembly window.

DUMP
Open a Dump window.

LANGUAGE
Open an LSM Information window with a display from a LANGUAGE command.

MAP
Open an LSM Information window with a program map display.

`STRUCTURE`
Open an LSM Information window with a structure component display.

`TYPE`
Open an LSM Information window with a variable type attribute display.

`UNION`
Is a synonym for STRUCTURE.

`VARIABLE`
Open an LSM Information window with a variable display.

`command-parameters`
Parameters for the window. The format of this argument depends on the type of the window being opened:

- If a Disassembly window is being opened, the second argument is the address at which the disassembly listing is to begin. For more information about how to specify this address see "DISASM" on page 111.
- If a Dump window is being opened, the second argument is the address at which the storage dump is to begin. For more information about how to specify this address see "DUMP" on page 113.
- If an LSM Information window is being opened for ARRAY, CALLERS, MAP, STRUCTURE, TYPE, or VARIABLE display, the remaining arguments are processed by the command as appropriate to generate the new display.
- If an LSM Information window is being opened for a LANGUAGE command display, the second argument is that needed by the LANGUAGE command. This window is closed if the command does not generate anything to display.

The OPEN command lets you have more than one of these types of window open, letting you view many non-contiguous areas of storage at once.

Return codes

0	Operation successful
1	Window type missing
2	Window type too long
5	Window type invalid
6	Too many windows open

OPTIONS

Toggles the display of the options window.

```
►►──OPTions──────────────────────────────────────────────────────────────────►◄
```

ORDER

Moves a window to the top of the screen.

```
►►──ORDer──┬────────┬──────────────────────────────────────────────────────────►◄
           └─window─┘
```

window
: The window that is moved. Select by a Window Specification or by placing the cursor in the window.

Return codes

0 Operation successful

1 No window selected.

OREGS

Opens or closes (toggles) the Old Registers window which contains a display of the registers and PSW as they were the last time IDF had control.

```
►►──OREGs────────────────────────────────────────────────────────────────────────►◄
```

If the CREGS command or the AREGS command was issued, the control registers or the access registers as of the last time IDF was in control are displayed. If the target has not executed any instructions the window is blank.

Return codes

0 Operation successful

PACKED

Sets or queries the format in which the data for packed decimal variables are displayed.

```
►►──PACked──┬─────────┬──────────────────────────────────────────────────────────►◄
            ├─DECimal─┤
            ├─*───────┤
            └─HEX─────┘
```

DECIMAL
: Packed decimal variables are displayed in decimal. This is the initial value.

***** The same as DECIMAL.

HEX
: Packed decimal variables are displayed in hexadecimal.

If the display format setting is not specified, the current display format for packed decimal variables is shown in a message.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid packed decimal variable display format

PARMS

Displays the Parameter List for Callable Blocks.

```
                       ┌─;──────────┐
                       ▼            │
►►──PARms──┬────────┬──module-name──┴──────────────────────────────────────►◄
           └─window─┘
```

window
: An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in an LSM Information window, uses or opens the first LSM Information window.

module-name
: A Callable Block name.

The Parameter List is shown as a list of parameter variable names.

Return codes

0	Operation successful
5	Arguments are invalid
6	No extract data files containing source code were loaded with LANGUAGE LOAD.

PAUSE

Delays the execution of IDF (and thus of the target program).

```
►►──PAUSe──┬───────┬───────────────────────────────────────────────────────►◄
           └─delay─┘
```

delay
: The number of hundredths of seconds to pause. An integer from 0 to 99999999. If omitted, a pause of 0.25 seconds is assumed.

Return codes

0	Operation successful

PER (CMS only)

Enables and disables the exploitation by IDF of Program Event Recorder (PER) functions.

```
►►──PER──┬─Y─┬──────────────────────────────────────────────────────────►◄
         └─N─┘
```

Y Enable PER exploitation. Break window displays PER=Y.

N Disable PER exploitation. Break window displays PER=N.

Examples

```
SET PER Y
SET PER N
```

PFK

Assigns a command or macro invocation to a PF key.

```
►►──PFK──pfkey-number──command-string──────────────────────────────────►◄
```

pfkey-number
An integer identifying the PF key. Has a value from 0 to 24. 0 is the ENTER key.

command-string
The PF key command text. This has a maximum of 40 characters.

Examples

```
PFK 1  SUBSET
PFK 10 ADSTOP
PFK 0  MACRO MYENTER
```

When you press the PF key, the command that you entered against the key is executed. Parameters typed on the command line may be used as parameters for the command.

If you want a PF key to invoke a macro, you must use the MACRO command in the text assigned to the PF key.

PFKDISP

Adjusts the display of PF key settings at the bottom of the screen in the Command window.

```
►►──PFKDISP──┬─ALL─┬───────────────────────────────────────────────────►◄
             ├─ON──┤
             └─OFF─┘
```

ALL
: Display settings of PF keys 1 to 24.

OFF
: Display no PF key settings.

ON Display settings of PF keys 1 to 12.

The size of the Command window varies depending on the number of PF key settings displayed.

PLOCATES

Displays the variables that may be located with a given locator (pointer) variable.

```
►►──PLOcates──┬────────┬──┬────────────────────┬──────────────────────────────────►◄
              └─window─┘  │ ┌─;──────────────┐ │
                          │ ▼                │ │
                          └───variable-name──┴─┘
```

The PLOCATES command determines the variables which may be located with a given locator (pointer) variable, and displays an entry for each in the LSM Information window.

window
: An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in a LSM Information window, uses or opens the first LSM Information window.

variable-name
: A variable name.

  Only the variable name is relevant in determining the located variables. Extra information such as:
  - Locating expressions for based variables
  - array index values
  - substring ranges

  is not needed, and should *not* be specified.

The Pointer Locates information display persists until:

- A PLOCATES command without arguments is issued.
- The window is closed with a CLOSE command.
- Another IDF Language command such as VARIABLE, STRUCTURE, ARRAY, TYPE, CALLERS, LANGUAGE STATUS, or MAP is issued. These commands update the LSM Information window with new information.
- The target program completes execution.
- Target program execution progresses beyond the variable's defined scope.

Examples

```
ploc base@
ploc DSA_ptr
plocates ptr1;ptr2
```

PRESERVE

Saves the current value of the IDF settings in a 32 element stack.

```
►►──┬─PREServe─┬──────────────────────────────────────────────────────────────►◄
    └─SAVe─────┘
```

If the IDF settings stack is full, the last stack element is replaced. Use the RESTORE command (see “RESTORE” on page 169) to restore the settings.

This command is intended for use within IDF macros, as it lets you save the current values of IDF settings, then change them for the duration of the macro, then restore them to their values before the macro was run.

Return codes

0	Operation successful

PREVIOUS

Scrolls a window backwards.

```
►►──┬─PREvious─┬──┬────────┬──────────────────────────────────────────────────►◄
    └─Up───────┘  └─window─┘
```

window
The window to be scrolled. Select it by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a scrollable window, then all scrollable windows are scrolled.

This command has an alternative meaning after the HISTORY command is executed. See “HISTORY” on page 125 for further information.

This command works only on a scrollable window, which is any window made smaller with the SIZE command, or one of the following:
- Break window
- Dump window
- Disassembly window
- LSM Information window
- Options window
- Skipped Subroutines window
- Target Status window

Whenever you use the NEXT command, IDF records information about the windows that are displayed on the screen in a page buffer. This page buffer holds information for about forty windows. When you use the PREVIOUS command, IDF attempts to determine the proper starting address for the previous display page by examining the page buffer. If you page forward, then page back, the starting location on the first page should be the same as on its previous display. If you page forward beyond the limits of the page buffer, information about the oldest pages is lost. If you open or close windows, the information in the page buffer may no longer be correct and scrolling may be erratic.

If IDF is unable to obtain information from the page buffer, the number of bytes that are displayed in the window is subtracted from the address of the first byte displayed. Unless the display shown is an unformatted dump, this is an imprecise amount. Paging backward and then paging forward does not necessarily display the original location.

Return codes

0	Operation successful
6	Command issued when it has no meaning

PROGCHK (CMS only)

Simulates a program check to the target program, typically to test the target program's recovery from program checks (for example, through an established ESPIE or ESTAE or ABNEXIT or PSW steal).

```
►►──┬─PROGck──┬──check-code──────────────────────────────────────────►◄
    └─PROGchk─┘
```

check-code
The program check code to be simulated. An integer, with the following possible values:

1	Operation exception
2	Privileged operation exception
3	Execute exception
4	Protection exception
5	Addressing exception
6	Specification Exception
7	Data exception
8	Fixed-point overflow exception
9	Fixed-point divide exception
10	Decimal overflow exception
11	Decimal divide exception
12	Exponent overflow exception
13	Exponent underflow exception
14	Significance exception
15	Floating-point divide exception

The program check is simulated with the current GPRs, FPRs, ARs, CREGS, and PSW.

As this command is intended to simulate an error and drive an error handler, that handler may be expecting some other conditions that must be set up before this command is issued. If necessary, you must first change these values before issuing the PROGCK command.

Return codes

0	Operation successful

PROGCK (CMS only)

The PROGCK command is a synonym of the PROGCHK command. For details, see “PROGCHK (CMS only).”

PSW

The PSW command is a synonym of the GOTO command. For details, see “GOTO” on page 122.

PSWSTEAL (CMS only)

Declares a target program instruction that steals the SVC or PGM new PSW.

```
►►──PSWSTEAL──address──────────────────────────────────────────────────────────►◄
```

address
: The storage location. If an expression is present on the command line, that expression is used as the argument. If the command line is empty, an attempt is made to determine the address from the cursor position.

The PSWSTEAL command effectively sets a permanent breakpoint at the specified location. Whenever that instruction is executed, IDF performs an interpretive execution of the instruction. It appears to the target program that it has stolen the new PSW. However, IDF retains control of the PSW locations.

To guarantee correct results, you must declare all occurrences of the following:
- All references (both store and fetch) to either SVC or PGM New.
- All store references to SVC or PGM Old.
- All store references to locations 136-159 (interrupt information)

The following instructions are supported for interpretive execution through the PSWSTEAL command, and are therefore its only valid targets:

MVC	Move characters
STM	Store multiple
LM	Load multiple
ST	Store
L	Load

The maximum number of PSWSTEAL breakpoints that may be active at any time is 75. These are not shown on the Break window, but may be obtained by means of the EXTRACT BREAK command.

If the PSWSTEAL command is issued against an instruction that is already declared, the PSWSTEAL breakpoint is cleared. The PSWSTEAL command toggles these special breakpoints.

A normal breakpoint may be set at the same instruction location as a PSWSTEAL breakpoint. When execution reaches the normal breakpoint, you are notified as usual. When execution passes through the PSWSTEAL breakpoint, no notification is performed, just as no notification is performed if the instruction is executed normally. Once set, the PSWSTEAL breakpoints may be forgotten.

Instructions which are the target of PSWSTEAL commands must reside in read/write storage.

When you issue the PSWSTEAL command, IDF exploitation of PER is locked out for the remainder of the debugging session. This does not affect the program in terms of its ability to use PER.

When the new PSW locations in low storage are displayed on a Dump window, the values the program placed in them are visible. However if you use EXTRACT LOCATION to obtain this information, IDF's PSWs are returned.

This facility enables you to use IDF to single-step through every instruction of the program's first level interrupt handler.

As a matter of convenience, the following is suggested:

- Define all instructions in the program that must be declared with PSWSTEAL as external symbols with a common prefix, for example, PSWnnnn
- Insert code in the PROFILE macro to read the program's MAP file, locate the external symbols, and set the PSWSTEAL breakpoints. For example:

```
/*-----------------------------------------------------*/
/* Read pgm MAP file into MAPREC.n array               */
/*-----------------------------------------------------*/
 fid = pgm 'MAP *'
 Address Command 'EXECIO * DISKR' fid '(FINIS STEM MAPREC.'
 If rc ¬= 0 Then
   Do
     'SET MSG2 Return code' rc 'trying to read 'fid''
     'SET ALARM'
     Exit
   End

/*-----------------------------------------------------*/
/* Establish PSWSTEALs using information from MAP file   */
/*-----------------------------------------------------*/
 PSWsteals = 0
 Do i = 1 To maprec.0
   Parse Upper Var maprec.i label .
   If Substr(label,1,3) ¬= 'PSW' Then Iterate
   'PSWSTEAL' label
   If RC ¬= 0 Then
     Do
       'SET MSG2 Error on PSWSTEAL' label' after' ,
               PSWsteals 'successful, RC='RC
       'SET ALARM'
       Exit
     End
   PSWsteals = PSWsteals + 1
 End
```

This makes the fact that you are stealing PSWs transparent as far as your operational procedures are concerned.

Return codes

0	Operation successful
1	No address specified
5	Syntax or other error in expression, or unsupported instruction
6	Module not yet loaded, or maximum number of PSWSTEALs already set

QUALIFY

Sets the name of the module that is used with some commands and addresses when no explicit module name is supplied.

```
►►──QUAlify──default-module-name──────────────────────────────────────────►◄
```

default-module-name
The name of the new default module.

Examples

```
SET QUALIFY ASMLANGX
```

QQUIT

The QQUIT command is a synonym of the QUIT command. For details, see "QUIT" on page 167.

QUIET

Controls the display of informational messages.

```
►►──QUIET──┬─ON──┬──────────────────────────────────────────────────────────────►◄
           └─OFF─┘
```

ON Informational messages are displayed.

OFF
Informational messages are not displayed.

QUIETLY

Temporarily suppresses the display of I (Informational), W (Warning) and E (Error) messages during the execution of the specified command.

```
►►──QUIETLY──command-name──command-parameters──────────────────────────────────►◄
```

command-name
The command that is being executed.

command-parameters
The parameters needed by this command.

The QUIETLY prefix command overrides the current MSGMODE setting, and suppresses messages (as if MSGMODE was OFF). It does not affect the actual value of the MSGMODE setting.

See "MSGMODE" on page 151 for more details about message suppression.

If an IDF message that normally rings the terminal alarm is issued under QUIETLY, the alarm is also suppressed.

Return codes

0	Operation successful
1	No command specified
2	Command name truncated
3	Command unknown
Other	Return code from the command

QUIT

Quits IDF.

```
►►──┬─QUIT──┬─────────────────────────────────────────────────────────►◄
    └─QQUIT─┘
```

If the QUIT command is issued from a PF key, it must be issued twice (press the key twice) to quit.

If the QUIT command is issued from the command line through a PF key that is set to COMMAND, IDF exits immediately.

If the QUIT command is issued by a macro, it must be issued twice to make IDF exit. In this case, IDF does not exit until the macro stops.

Return codes

0	Operation successful

RCQUIT

Quits IDF with a specific return code.

```
►►──RCQuit──┬─────────────┬───────────────────────────────────────────►◄
            └─return-code─┘
```

return-code
: The return code. This must be a valid IDF expression. If omitted, the command line is checked for an expression and if a valid expression is found its value is used as the return code. If omitted and the command line is empty, the current content of the target program's R15 is used as the return code.

Intended for use by macros, but can also be issued from the command line if desired.

Return codes

If an invalid expression is provided, either as part of the command or on the command line, RC=5 is returned to the invoking macro and processing is ended.

If the RCQUIT command completes successfully, a return code of 0 is passed to the issuing macro. When the macro exits, IDF returns to CMS, z/OS, or z/VSE and passes the specified return code.

REFRESH

Forces IDF to rebuild the IDF user interface screen image, and optionally rewrite it to the terminal.

```
              ┌─DISP───┐
►►──REFresh───┤        ├──────────────────────────────────────────────────────►◄
              └─NODISP─┘
```

DISP
: Rebuilds the screen image, and writes it to the terminal.

  This is useful if a macro wishes to update the IDF user interface display, but continue running. Normally the rebuild and screen update does not occur until the macro completes execution.

NODISP
: Rebuilds the screen image, but does not write the updated image to the terminal.

  This is useful if a macro has issued a command which causes an update of the IDF user interface display, and needs to issue EXTRACT commands for the new display image. Eliminating the extra terminal update speeds up the execution of the macro.

  This facility can even be used to hide the update by immediately issuing more commands to restore the user interface display to the previous configuration.

The REFRESH command can also be used to restore the IDF user interface display. This is normally only needed in a macro that sends line mode output to the screen, and should only be needed on TSO.

Return codes

0	Operation successful

REGS

Opens or closes (toggles) the Current Registers window which contains a display of the registers and PSW or the current contents of the control registers or the access registers.

```
►►──REGs──────────────────────────────────────────────────────────────────────►◄
```

Return codes

0	Operation successful

REGS64 (z/OS only)

Toggles the display of the Current Registers and Old Registers windows, from displaying the 32-bit General Purpose Register and PSW to 64-bit Registers and PSW.

```
►►──REGS64────────────────────────────────────────────────────────────────────►◄
```

If the Current Registers window is not open the window is opened.

Return codes
0 Operation successful

REGSTOPS (CMS only)

The REGSTOPS command is a synonym of the ADSTOPS command. For details, see "ADSTOPS (CMS only)" on page 95.

RESTORE

Restores the IDF settings from the 32 element stack.

```
►►──REStore──────────────────────────────────────────────────────────────►◄
```

If the IDF settings stack is empty, the current setting values are maintained.

This command is intended for use within IDF macros, as it lets you restore the IDF settings to the state that they held when the last PRESERVE or SAVE command was issued.

Return codes
0 Operation successful

RETRIEVE

Places the previous command on the command line.

```
►►──┬─RETRieve─┬─────────────────────────────────────────────────────────►◄
    └─?────────┘
```

IDF maintains the last 30 inputs in a circular buffer. The RETRIEVE function shifts the most recent of these into the command line.

Return codes
0 Operation successful

RIGHT

Scrolls a window to the right.

```
►►──RIGHt──┬────────┬──┬───────────────────┬─────────────────────────────►◄
           └─window─┘  └─number-of-columns─┘
```

window
A scrollable window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a scrollable window, then all open scrollable windows are scrolled.

number-of-columns
The number of columns by which the window is scrolled.

This can be specified as:

- an integer
- an integer prefixed by a +
- an integer prefixed by a -, specifying the number of screen columns by which the window is to be scrolled to the *left*.

If this parameter is omitted, each window being scrolled is scrolled by the current number of data columns for that window.

This command is normally only meaningful when a Disassembly window is open. However, any window made smaller with the SIZE command can also be scrolled.

Return codes

0 Operation successful
5 Arguments are invalid
6 Command issued when it has no meaning

RLOG

The RLOG command (Repeat from Log) controls the execution of IDF commands stored in the command log.

```
►►──RLog──┬───────────────┬──────────────────────────────────────────────────►◄
          ├─search-string─┤
          └─$─────────────┘
```

search-string
The text of a search string. All commands in the log file, up to and including the first command which begins with the specified search argument, are executed. This search is case insensitive.

By providing an argument not in the log file, for example, eof, you can execute all commands in the log.

$ RLOG first executes anything on the command line, then retrieves the next command from the log and places it on the command line. Thus by repeatedly pressing the PF key, you can step through as many commands as you like, previewing (and optionally modifying) them before they are executed.

When issued with no arguments, the next command is retrieved from the log file and placed on the command line. You can then press ENTER to execute it, or clear the command line as you choose. By clearing the command line, you skip executing the command.

You can get a similar result by specifying the RLOG option at invocation. When specified at invocation, RLOG means:

1. Repeat all commands in the log, and
2. Start logging any new commands issued.

If you are executing many commands from the log file, any command which has a nonzero return code or sets the alarm, stops the RLOG operation.

For more details on command logging support, see "Command record and playback features" on page 80.

RUN

Begins execution of the target program and runs to the next unusual event (such as a breakpoint, program check, or completion).

```
►►──RUN──────────────────────────────────────────────────────────────────────►◄
```

If the RUN command is issued by a macro, it does not take effect until the macro exits.

The MRUN command performs a similar function. It executes target instructions *before* returning control to the invoking macro.

Return codes
0 Operation successful

RUNEXIT

Executes the exit routine, passing it the single argument PFKEY.

```
►►──RUNExit──────────────────────────────────────────────────────────────────►◄
```

For more information, see Chapter 13, "The IDF exit routine," on page 213.

Return codes
0 Exit routine executed
6 Exit routine not found

R0-R15

Evaluates the expression provided and places it in the indicated General Purpose Register.

```
►►──Rn──expression───────────────────────────────────────────────────────────►◄
```

expression
The expression to be evaluated. Can also be set from the cursor position.

There are 16 separate commands, named R0, R1, ... R15. They all have the same arguments and function, and differ only in which GPR is changed.

Return codes

0	Operation successful
1	No address specified
5	Syntax or other error in expression
6	Conditions do not permit completion of command

SALIMIT

Sets the maximum depth of the CALLERS display.

```
►►──SALimit──max-caller-display-depth──────────────────────────────────────────►◄
```

max-caller-display-depth
The maximum depth of the CALLERS display. Integer from 1 to 999999.

The initial value is 100.

This is intended to prevent problems when the program call chain is damaged, or is of unexpected depth (due to runaway recursion).

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
5	Arguments are invalid

SAREGS

Toggles the display of Save Area header and register information when the CALLERS command is displaying information about each generation of the program caller hierarchy.

```
►►──SARegs──┬─ON──┬────────────────────────────────────────────────────────────►◄
            └─OFF─┘
```

ON Enable the display of the Save Area header and registers.

OFF
Disable the display of the Save Area header and registers.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

SAVE

The SAVE command is a synonym of the PRESERVE command. For details see “PRESERVE” on page 162.

SEARCH

Searches for a string in storage.

```
►►──SEArch──┬────────┬──string──────────────────────────────────────────────►◄
            └─window─┘
```

window
: A Dump window or Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted, the search begins at the first location that is displayed. If both a Dump window and a Disassembly window are open, but show different storage areas, it is not clear where the search should begin, and an error message is issued.

string
: The string for which IDF is to search.

  Search arguments may be specified in either hexadecimal or character form. Hexadecimal search arguments must contain an even number of hex digits. An apostrophe in a character argument is represented by two successive apostrophes.

  Examples of valid search arguments are:

```
X'0741fe3022'
C'The quick brown fox'
C'The dog''s feet are'
```

This function is only valid when a Dump window or a Disassembly window is open.

When the search argument is located, it becomes the first location displayed in the selected window or in the first Disassembly window or Dump window.

Repeat the command to continue searching the remainder of virtual storage.

If the search argument is not found within the target program, a message is issued.

Return codes

0	Operation successful
5	Search argument missing or invalid
6	Not valid in current display mode, unable to determine start address for search, or target not found

SELFNUCX (CMS only)

```
►►──SELFNucx──┬─SYMbol──symbol-name──────────┬──────────────────────────────►◄
              └─VALue──start-offset-value─┘
```

Lets the PROFILE macro control the start offset of the code that is relocated by a self-loading CMS Nucleus extension.

symbol-name
: A symbol that defines the start of the relocated code.

start-offset-value
An expression that resolves to the start offset value.

This only applies to the target program specified when IDF was invoked.

SELFNUCX SYMBOL may only be issued within the PROFILE macro before the target program is loaded into memory. If it is issued after that point, RC=6 results.

SELFNUCX VALUE may be issued within the PROFILE macro, and may also be used later to let you (or IDF macros) redefine this offset after the target program is loaded.

When debugging a self-loading nucleus extension, it should be possible to invoke IDF to debug the program as a standard user-area or transient module, and trace through the self-loading code. After the code is relocated, use the BASE and SELFNUCX VALUE commands to follow the relocated code to its new location. Symbol resolution is automatic, determined by the program base address, symbol value, and SELFNUCX offset.

Examples

```
SELFNUCX SYMBOL FREEGO
SELFNUCX VALUE X'44'
```

SET ADSTOP (CMS only)

Sets or clears one end of an ADSTOP range.

```
►►──SET──ADSTop──┬─ON──┬──┬────────────┬──────────────────────────────────►◄
                 └─OFF─┘  └─expression─┘
```

ON Set one end of an ADSTOP range.

OFF
Clear one end of an ADSTOP range.

expression
An expression resolving to the address for the end of the ADSTOP range.

See also “ADSTOPS (CMS only)” on page 95.

Examples

```
SET ADSTOP ON ALLOPEN+4
SET ADSTOP OFF X'200E4'
```

SET AREG

Changes the contents of an Access Register (AR).

```
►►──SET──AREG──access-register-number──expression──────────────────────────►◄
```

access-register-number
The Access Register number; an integer from 0 to 15.

expression
The new value for the Access Register.

You must be in an ESA environment for this command to work.

Examples

```
SET AREG 2 X'00010004'
```

SET BREAK

Sets or clears a breakpoint at an address.

```
►►──SET──BREak──┬─ON──┬──┬─────────┬──┬──────────────────┬──────────────────────────►◄
                └─OFF─┘  └─address─┘  │ ┌──────────────┐ │
                                      │ ▼              │ │
                                      └───|──command───┴─┘
```

ON Set the breakpoint.

OFF
Clear the breakpoint.

address
The address at which the breakpoint is set. If an expression, the expression is used to provide the address.

If omitted, the address is determined from the cursor position. If it is not possible to determine an address from the cursor position (for example, when the cursor is on the command line and the command line is empty), then an error.

command
A command that is executed when the breakpoint is taken, before control is returned to the user. Many commands can be specified at the end of the SET BREAK command, separated from the address and each other by a vertical bar (|). If a command receives a non-zero return code, the remaining commands in the list are not executed.

See also “BREAK” on page 100.

Examples

```
SET BREAK ON ALLOPEN+8
SET BREAK OFF 4(R2)
SET BREAK ON MOON | SET MSG Houston... the Eagle has landed.
```

SET COMMAND

Places text on the command line.

```
►►──SET──COMmand──text────────────────────────────────────────────────────────────►◄
```

text
The text to be placed on the command line.

Examples

```
SET COMMAND Press VALUE to see it yell about this expression
```

SET EXITEXEC

Names the current IDF exit routine (exec).

```
              ┌─EXIT──────────┐
►►──SET──EXItexec──┤               ├──────────────────────────────────────────────►◄
              └─IDF-exit-exec─┘
```

EXIT
  Default name of IDF exit routine.

IDF-exit_exec
  The name of the current IDF exit routine.

For more information, see Chapter 13, "The IDF exit routine," on page 213.

Examples

```
SET EXITEXEC STUFF
```

SET GLOBAL STEM

Writes the data in the specified REXX stemmed arrays to the same named Global Storage stems.

```
                   ┌──────────────┐
                   ▼              │
►►──SET──GLObal────stem-name.─────┴─────────────────────────────────────────────►◄
```

stem-name.
  A REXX stemmed array name. Must have a terminating period.

Each REXX stemmed array must be in standard EXECIO READ/WRITE stem format.

Any existing Global Storage stem by the same name is dropped before the REXX stemmed array is inspected.

If the *stemname*.0 item is not a positive integer, an error message is issued. The new Global Storage stem is *not* created.

If a REXX stemmed array element is omitted, an error message is issued. The new Global Storage stem is *not* created.

REXX stemmed array elements with null values are supported.

If a REXX stemmed array element is longer than 4,096 bytes, an error message is issued. The new Global Storage stem is *not* created.

Examples

```
SET GLOBAL ptr-array.
SET GLOBAL data.buf1.stem.
```

REXX variables read

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Information for the stemmed array element

SET GLOBAL TEXT

Saves the value of the IDF global area. Other macros can then access it, by using the EXTRACT GLOBAL command.

```
►►──SET──GLOBAL──global-text──────────────────────────────────────────────►◄
```

global-text
The new global text value, up eighty characters in length.

Do not end the first blank-delimited word of the new global text value with a period (.), as this is a SET GLOBAL STEM command.

Examples

```
SET GLOBAL state3
```

SET ICOUNT

Sets the count of instructions executed to a specified value.

```
►►──SET──ICOunt──instruction-count──────────────────────────────────────────►◄
```

instruction-count
An expression which is resolved to the new instruction count value.

Valid only when the PATH option is set.

Examples

```
SET ICOUNT 0
```

SET OFFSET

Toggles the display of addresses using offsets.

```
►►──SET──OFFSet──┬─ON──┬──────────────────────────────────────────────────►◄
                 └─OFF─┘
```

`ON` Enables the display of addresses using offsets.

`OFF`
Disables the display of addresses using offsets.

SET OPTION

Sets an IDF option ON or OFF.

```
►►──┬─SET──OPTION──┬─ON──┬──option─┬──────────────────────────────────────────►◄
    │              └─OFF─┘         │
    │ (1)                          │
    └──────┬─────┬──option──┬─ON──┬─┘
           └─SET─┘          └─OFF─┘
```

Notes:

1 This alternative form may be used only when it does not conflict with other command names.

option
The IDF option name.

The IDF options control the environment of IDF, which in turn controls the way IDF works, and the information you see on the screen.

Most options can be controlled through the SET OPTION command. Many of these options can also be specified at the invocation of IDF, either as parameters to the ASMIDF call, or as items in the PROFILE macro. The restrictions that apply to options specified in a PROFILE macro are set out in "Command restrictions related to PROFILE execution" on page 204.

Some options are invocation options only, and cannot be controlled after invocation with the SET OPTION command. The information provided by these options must be available at invocation. It makes no sense for it to be provided later.

Some options are controlled by a command of the same name. Some of these can also be specified at invocation.

These options can be specified at invocation and set by the SET OPTION command:

1ADSTOP	DMS0	NOAUTOSZ	OFFSET	SCDACTIV
AMODE24	EXITEXEC	NOBCX	OLDBREAK	SELFNUCX
AMODE31	FULLQUAL	NODSECTS	PASSPGM	STOPNOP
AMODE64	HEXDISP	NOIMPMAC	PATH	STOPSTMT
ASCII	HEXINPUT	NOINVPSW	PATHFILE	SVC97
AUTOLOAD	IMPMACRO	NOMODMAP	QWDUMP	SWAP
AUTOSIZE	INVPSW	NOSTOPNP	RISK	SYSTEM
BCX	LSMDEBUG	NOSTOPST	RLOG	TRACEALL
CKSUBCM	MACROLOG	NOSVC97	ROWSTYLE	TRANS
CMDLOG	MODMAP	NUCEXT	SBORDER	UNFTDUMP
CMPEXIT	NOAUTOLD			

These options are set and changed by a command with the same name. They cannot be specified as an invocation parameter, but each option has an initial value:

APROGMSG	BRIEF	MAJOR	MSGMODE	SPACE
AUDIT	COMPACT	MSGID	SAREGS	

These options are set by a command with the same name, and can be specified as an invocation parameter:

COLORS	COLOURS	MODE

These options are specified only as an invocation option. They cannot be changed by a command, or by the SET OPTION command:

FASTPATH	LIBE	LUNAME	PROFILE
ISA	LINE	NOPROFIL	RLOG

For explanations of options specified at invocation, see Chapter 4, “Invoking IDF to debug your program,” on page 21.

For explanations of options specified by a command, see the command description.

The current state of many options is displayed in the Options window, see “Options window” on page 75.

You can use the alternate syntax of *option* ON or SET *option* ON when the option name does not conflict with any other command or SET command name. If the option name conflicts with a command name, the alternate syntax SET option ON may still be used.

Examples

```
hexinput ON
SET offset ON
SET OPTION ON impmacro
```

SET PSW

Sets the current PSW.

```
►►──SET──PSW──hex─────────────────────────────────────────────────────►◄
```

hex
The new PSW value, expressed as 1 to 16 hexadecimal digits.

Normally the SET PSW command does not let you set the PSW to an invalid PSW. If the INVPSW option is ON, then the PSW may be set to any combination of sixteen hexadecimal digits. This is useful in writing a macro that gets control at a breakpoint in an error recovery routine and you want the PSW to show the original PSW in error.

Normally it does not let you set the address part of the PSW to an address that is outside the programs limits. If the TRACEALL option is on, the address part of the PSW may be set to any even address from zero to VMSIZE. If the RISK option is on, the address part of the PSW may be set to any even address you want.

Examples

```
SET PSW FFE0000000020000
```

SET REGSTOP (CMS only)

Controls PER monitoring of General Purpose Register (GPR) contents.

```
►►──SET──REGSTop──┬─ON──┬──general-purpose-register-number─────────────────────────────►◄
                  └─OFF─┘
```

ON Enable PER monitoring of the contents of the specified GPR.

OFF
Disable PER monitoring of the contents of the specified GPR.

general-purpose-register-number
The GPR number; an integer from 0 to 15.

See also "ADSTOPS (CMS only)" on page 95.

Examples

```
SET REGSTOP ON 4
SET REGSTOP ON 12
SET REGSTOP OFF 9
```

SET SIZE

Specifies the size in bytes of the target program specified when IDF was invoked.

```
►►──SET──SIZE──module-length──────────────────────────────────────────────────────────►◄
```

module-length
An expression which resolves to the module length value.

IDF automatically sets the size based on information found in the module file. Use this command to override the size determined by IDF.

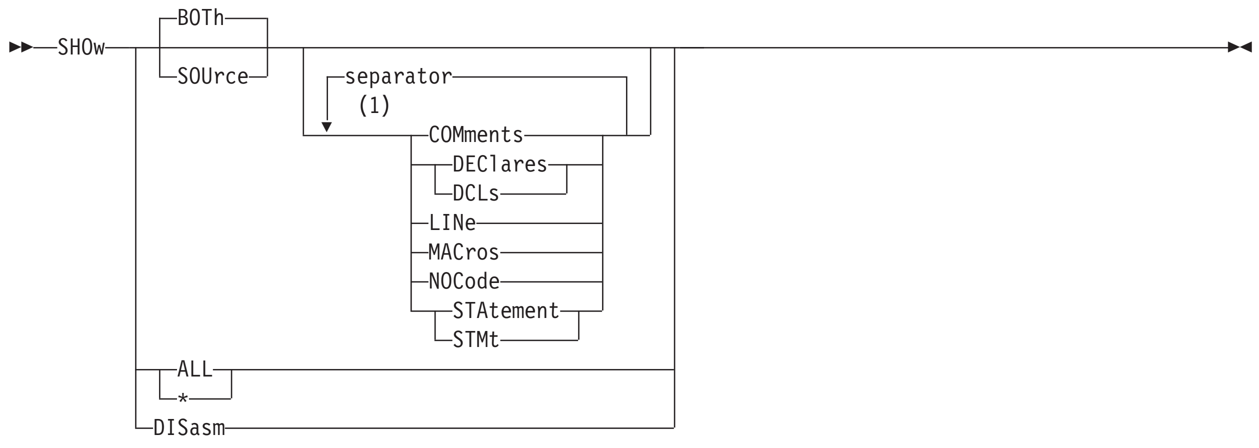
Use the MODULE SIZE command to specify the size in bytes of other programs defined to IDF.

Examples

```
SET SIZE X'1A08'
```

SHOW

Controls the display of source code and disassembly. It is used to enable the display of specific information, and is the opposite of the HIDE command.



Notes:

1 An option can be chosen no more than once.

BOTH
Show source code interspersed with the generated assembler.

SOURCE
Show source code only, without interspersed assembler code.

SEPARATOR
A comma, blank, or semicolon. Separates the suboptions of SOURCE and BOTH.

COMMENTS
Show block comment source when source code is displayed.

DECLARES | DCLS
Show declaration source when source code is displayed.

LINE
Show source line numbers when source code is displayed.

MACROS
Show macro expansion source when source code is displayed.

NOCODE
Show source lines with no corresponding object code when source code is displayed.

STATEMENT | STMT
Show source statement numbers when source code is displayed.

ALL | *
Show all source code, interspersed with the generated assembler.

DISASM
Show disassembled assembler code only, without source code.

The initial settings are:
- BOTH, COMMENTS, DECLARES, MACROS, NOCODE, STATEMENT

Return codes

0	Operation successful
2	Keyword truncated
5	Invalid information type keyword

SIZE

Changes the size of a window on the screen.

```
►►──SIZe──┬──────────┬──┤ Location ├──────────────────────────────────────►◄
          ├─window───┤
          ├─ADSTops──┤
          ├─AFPR─────┤
          ├─BREak────┤
          ├─DISasm───┤
          ├─DUMP─────┤
          ├─OREGs────┤
          ├─REGs─────┤
          ├─SKIPstep─┤
          ├─STAtus───┤
          └─LSMinfo──┘
```

Location:

```
├──┬──────────────────────────────────────────┬──────────────────────────────┤
   └─┬─┬─────┬─row───┬─┬─┬─────┬─column──┬─┘
     │ ├─ + ─┤       │ │ ├─ + ─┤         │
     │ └─ - ─┘       │ │ └─ - ─┘         │
     ├─*─────────────┤ ├─*───────────────┤
     └─┬─STANdard─┬──┘ └─┬─STANdard─┬────┘
       └─STD──────┘      └─STD──────┘
```

WINDOW
The window to be sized. Select by a Window Specification, or by placing the cursor in the window.

ADSTOPS
The AdStops window.

AFPR
The Additional Floating-Point Registers window.

BREAK
The Break window.

DISASM
The Disassembly window with the lowest window id.

DUMP
The Dump window with the lowest window id.

OREGS
The Old Registers window.

REGS
The Current Registers window.

`SKIPSTEP`
The Skipped Subroutines window.

`STATUS`
The Target Status window.

`LSMINFO`
The LSM Information window with the lowest window id.

`rows`
The number of window data rows to be displayed.

This may be specified as:

- an integer. Specifies an absolute number of window data rows.
- An integer prefixed by a +. Specifies the number of extra window data rows to be used.
- An integer prefixed by a -. Specifies the number of fewer window data rows to be used.
- * Indicates that the current number of data rows for this window should be used.
- STANdard | STD. Indicates that the standard number of data rows for this window type should be used.

If this parameter is omitted, the position of the cursor on the screen is used as the new location of the lower right corner of the window. In this case, the columns parameter must also be omitted.

`columns`
The number of window data columns to be displayed.

This may be specified as:

- An integer. Specifies an absolute number of window data columns.
- An integer prefixed by a +. Specifies the number of extra window data columns.
- An integer prefixed by a -. Specifies the number of fewer window data columns.
- * Indicates that the current number of data columns for this window should be used.
- STANdard | STD. Indicates that the standard number of data columns for this window type should be used.

If this parameter is omitted, the number of data columns in the window remains the same.

If the window is not named, it is assumed that this command is specifying the number of rows in a newly sized window.

Windows cannot be made larger than their maximum size. You can size them so that they overlap any other window. However, no window can overlap the Command window. You cannot size a window so that some of the window falls outside the screen. If you try, IDF repositions the window so that it remains completely on the screen.

If a window is closed and then opened, its size is the system default for that type of window and not that specified on the last SIZE command for that window type.

Examples

To change the window 01 to have 5 data rows and 60 data columns:

```
SIZE =01 5 60
```

To change the window in which the cursor is placed to have 8 data rows and 20 data columns:

```
SIZE 8 20
```

To add 2 data rows to the window in which the cursor is placed:

```
SIZE +2
```

To resize window 03 so that the lower right corner is at the current cursor position:

```
SIZE =03
```

Return codes

0	Operation successful
1	Window name or Row location missing
2	Operand longer than eight characters
5	Window name invalid or syntax or other error in expression

SKIPSTEP

Sets or clears a subroutine to be skipped, or displays the Skipped Subroutines window.

```
►►──SKIPstep──┬─────────┬──────────────────────────────────────────────►◄
              └─address─┘
```

address
: A storage location.

  If an expression is present on the command line, that expression is used as the argument. If the command line does not contain an expression, an attempt is made to determine the address from the cursor position.

If no expression is provided on the command line, and it is not possible to determine an address from the cursor position (for example, when the cursor is on the command line and the command line is empty), the Skipped Subroutines window is opened if it is not already open. If the Skipped Subroutines window is already open, it is closed.

If an address is supplied, either as an expression on the command line or by means of cursor position, the subroutine at that address is skipped when the instruction that calls it is single-stepped, statement stepped, or executed when the PATH or FASTPATH options are on. If the subroutine at the specified address is already being skipped, it is removed from the list of skipped subroutines. The SKIPSTEP command acts as a toggle to set or clear a skipped subroutine.

If a breakpoint or a watchpoint whose condition is true is placed within the execution path of the subroutine to be skipped, execution stops at that breakpoint or watchpoint.

Return codes

0	Operation successful
1	No address specified
5	Syntax or other error in expression
6	Location, or location does not contain a valid instruction

SPACE

Separates the display of variables or sets of components with a blank line.

```
►►──SPAce──┬─ON──┬─────────────────────────────────────────────────────►◄
           └─OFF─┘
```

ON Enable the display of the blank separator line.

OFF
Disable the display of the blank separator line.

When many variables or structure components are displayed, IDF normally begins the display of each variable or set of components immediately. This command can separate each variable or set of components with a blank line.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown

STATUS

Opens and closes (toggles) the Target Status window that contains information about the target programs.

```
►►──STAtus──────────────────────────────────────────────────────────────────────────►◄
```

For each target program, this includes:
- the target program name
- the target program start location
- the target program length (bytes)
- the target program primary entrypoint location if the first segment of any program object
- the number of internal assembler level symbols known to IDF

Information about the initial target program is shown when the Target Status window is opened. If many target programs are loaded, the PREVIOUS or NEXT commands may be used to scroll the Target Status window to view the information for the extra programs.

The target program name is highlighted when the information for the qualified program is shown.

Return codes

0	Operation successful

STEP

Executes the next instruction in the target program.

```
►►──STEp────────────────────────────────────────────────────────────────────────────►◄
```

If the STOPNOP option is OFF or the NOSTOPNP option is ON, then IDF does not stop on NOP and NOPR instructions that follow BAL, BALR, BAS, and BASR instructions.

Step works by establishing a breakpoint at the next instruction which is immediately reset once reached. If an Execute instruction has the immediately following instruction as its target, then single-stepping from the EX will fail (see “BREAK” on page 100 for information about how breakpoints are established).

For more details see "Controlling single-stepping your program" on page 86

Return codes

0	Next instruction is executed when macro exits
6	Single-step mode locked out pending execution of startup command

STMTSTEP

Performs a repeated single-step operation until the target program is executing a source statement other than the one it was executing when the command was issued.

```
►►──STMTstep──────────────────────────────────────────────────────────────────►◄
```

If a significant event occurs during the execution of the STMTSTEP command, execution stops at that point. Examples of significant events are cases where the target has branched outside its boundaries, completed, and returned control to IDF, breakpoints, watchpoints, and so on.

By default, if the repeated single-step operation reaches code that is within a code section for which IDF Language extract data was not loaded, IDF stops single-stepping. Turning off the STOPSTMT option or using the NOSTOPST option prevents IDF from stopping at these points.

No exit routine processing is performed during, or at the completion of, the STMTSTEP command. As with MRUN or MSTEP, if the STMTSTEP command is issued within a macro, use EXTRACT EVENT to determine what type of event completed the command.

If you issue the STMTSTEP command when the PSW is pointing to code that is within a code section for which IDF Language extract data was not loaded, IDF attempts to issue LANGUAGE LOAD for you. This only works if the name of the code section (CSECT) containing the code matches the file name of the extract file. If you do not want IDF to automatically issue these LANGUAGE LOAD commands you should set the AUTOLOAD option to OFF or the NOAUTOLD option to ON.

If the STOPNOP option is OFF or the NOSTOPNP option was used, then IDF does not stop on NOP and NOPR instructions that follow BAL, BALR, BAS, and BASR instructions.

z/VM If you issue STMTSTEP from within an IDF macro, you should ensure that it is issued through the LPSW Fastpath addressing environment, which is the default environment established when the IDF macro is entered. Using this interface eliminates any SVC linkages between REXX and IDF, so that IDF can provide optimum flexibility in what the target program can itself execute under STMTSTEP. See the usage notes under "MRUN" on page 149 for further details.

For more details see "Controlling single-stepping your program" on page 86.

Return codes

0	Operation successful
6	Command issued before IDF initialization is complete

STOKEY

Displays the storage key associated with an address.

```
►►──STOKey──┬─────────┬──────────────────────────────────────────────►◄
            └─address─┘
```

address
: A storage location.

  If an expression is present on the command line, that expression is used as the argument. If the command line is empty, an attempt is made to determine the address from the cursor position. Failing this, the current execution address is obtained from the PSW.

For information about expressions, see "Address expressions" on page 80.

Return codes

0	Operation successful

STOREMAP

Displays information about storage allocation.

```
►►──STORemap──┬──────────────┬───────────────────────────────────────►◄
              │  ┌─;───────┐ │
              │  ▼         │ │
              └───address──┴─┘
```

address-expression
: A storage address expressions.

  The portion of the Storage Allocation Map relevant to the address is displayed.

  If no storage address expression is supplied, a complete Storage Allocation Map is displayed.

For information about expressions, see "Address expressions" on page 80.

The format of the Storage Allocation Map is system dependent.
- On CMS, it is like that displayed by the CMS STORMAP (ALL command.
- On z/OS, it is like that displayed by the TSO/E TEST LISTMAP command.
- On z/VSE, STOREMAP displays a message stating that it is unavailable in this environment.

Return codes

0	Operation successful
5	Arguments are invalid

STRUCTURE

Displays the contents of the components of one or more structures.

```
►►──STRucture──┬────────┬──┬─────────────────────┬──────────────────────────────────►◄
               └─window─┘  │  ┌─;─────────────┐  │
                           │  ▼               │  │
                           └──┴─variable-name─┴──┘
```

window
: An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in a LSM Information window, uses or opens the first LSM Information window.

variable-name
: The variable name. Each variable is:

  - The major component of a structure, in which case variables within the entire structure are visible.
  - A component (member) at an intermediate level within a structure, in which case only variables within that portion of the of the structure are visible.

  Use dot-qualification to uniquely identify structure components with ambiguous names.

  Simple structure components are defined by name only. You can define components within a based structure by name only, in which case the declared basing is used by IDF, or you can specify an explicit locating expression.

  See “Variable expressions” on page 87 for a complete description of the syntax of the expressions used as STRUCTURE variable name arguments.

Display is *not* supported for:
- embedded arrays within structures
- arrays of structures

The STRUCTURE display is in the form of a list of the component variables of the structure, in order of definition.

The STRUCTURE display persists until:
- A STRUCTURE command without arguments is issued
- A UNION command without arguments is issued
- The window is closed using a CLOSE command.
- Another IDF Language command such as VARIABLE, ARRAY, CALLERS, PLOCATES, LANGUAGE STATUS, or MAP is issued. These commands update the LSM Information window with new information
- The target program completes execution
- Target program execution progresses beyond the structure's defined scope

If the contents of a variable within the structure change while the program is running to a breakpoint, the changed data is shown on the screen when the breakpoint is reached.

You can change the displayed data by overtyping it.

In EBCDIC display mode, character data equal to X'FF' or below X'40' is displayed as a period character.

In ASCII display mode, character data which does not correspond to a displayable EBCDIC character is displayed as a period character.

If a based structure was respecified, the current basing specification is used.

Note: The display of the contents of the variables within a structure may be incorrect if the PSW indicates that execution is in the middle of a statement. This is because the variable may be in a transitional state, not having yet achieved its new value. Variable contents are only certain at the start and end of a statement.

Examples

```
Str struct1
Str addr(x'20000')->struct1
Str addr(12(R2))->ptr->struct1
Str ptr->ptr2->struct3
Str ptr(3)->ptr->struct1
Str struct1;struct2
Str struct1 ;ptr->struct1
Str struct1 ; struct4
```

SUBSET (CMS only)

Enters CMS SUBSET.

```
►►──SUBset──────────────────────────────────────────────────────────►◄
```

Valid unless the target program is a transient.

Return codes

0	Operation successful
6	The target program is a transient and SUBSET destroys the debugging environment

SVC (CMS only)

Controls the trapping of SVC instructions within the defined limits of the target programs.

```
►►──SVC───┬─Y─┬─────────────────────────────────────────────────────►◄
          └─N─┘
```

Y Enable SVC trapping. Break window displays SVC=Y.

N Disable SVC trapping. Break window displays SVC=N.

Examples

```
SET SVC Y
SET SVC N
```

SWAP

Displays the screen image associated with the target program.

```
►►──SWAp──────────────────────────────────────────────────────────────────►◄
```

This function is only valid if the SWAP option is specified.

After displaying the target program's screen image, IDF waits until any PF key is pressed before refreshing its display.

Return codes

0 Operation successful

6 Either the SWAP option was not specified, or no screen image was captured

SYMBOL

Defines an IDF internal (TESTRAN format) symbol.

```
►►──SYMBOL──(──┬────────────────┬──code-section-name──)──symbol-name──code-section-offset────────►
               └─module-name──.─┘

►─module-offset──symbol-length──┬─I─┬──┬─F─┬──symbol-type──────────────────────────────────────►◄
                                └─E─┘  └─U─┘
```

module-name
The name of the module within which the symbol occurs. If present, the name must be followed by a period. If omitted, the symbol is assumed to be within the qualified module.

code-section-name
The name of the code section within which the symbol occurs.

symbol-name
The name of the actual symbol.

code-section-offset
The offset of the symbol within the specified CSECT (hex).

module-offset
The offset of the symbol within the target module (hex).

symbol-length
The total length associated with the symbol (hex).

I The symbol is an internal symbol

E The symbol is externally known.

F The symbol is fully defined.

U The symbol is not fully defined. This may occur if no information is available to define the start of the CSECT within which the symbol occurs.

symbol-type

The type of the symbol. This is a 2-digit hex value, and is one of the following:

00	Space
01	CSECT
02	DSECT
03	COMMON
04	Machine Instruction
05	CCW
06	EQU, LTORG, CNOP, ORG
10	C-con
14	X-con
18	B-con
20	F-con
24	H-con
28	E-con
2C	D-con
30	A/Q-con
34	Y-con
38	S-con
3C	V-con
40	P-con
44	Z-con
48	L-con
FE	Self-defining, addr is actual value
FF	Unknown, no symbol type available

Examples

```
SET SYMBOL (ASMIDF) ASMIDF     00000000 00000000 00000060 E F 01
SET SYMBOL (VARMVSXA.VARASM) BTHING 000009C0 00000050 00000028 I F 10
```

TASKS (TSO only)

Displays information about the executing tasks.

```
►►──TASKs──────────────────────────────────────────────────────────►◄
```

- It is like that displayed by the TSO/E TEST LISTMAP command.

  The SVC97 option is needed to allow task information to be obtained.

Return codes

0	Operation successful
6	Unable to obtain task information

TITLE

Sets the title text displayed on the top border line of the selected window.

```
►►──TITle──┬────────┬──┬────────────┬──────────────────────────────►◄
           └─window─┘  └─title-text─┘
```

window
A window. Select by a Window Specification, or by placing the cursor in the window.

title
The new title text for the window. If omitted, the window text reverts to the default for the window.

The new window title text is retained while the window is open, or if the window is minimized and later maximized. If the window is closed, and later reopened, the window text for the new window is the default for the window.

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
5	Arguments are invalid

TOP

Displays source code, starting at lowest available address within the current code section.

```
►►──TOP──┬────────┬──────────────────────────────────────────────────────►◄
         └─window─┘
```

window
A Disassembly window. Select by a Window Specification or by placing the cursor in the window. If omitted and the cursor is not in a Disassembly window, uses the first Disassembly window.

Return codes

0	Operation successful
6	No code section definition corresponds to the current address.

TRIGGER LOAD

Initiates deferred breakpoint processing for a module.

```
►►──TRIgger──LOAd──module-name──────────────────────────────────────────►◄
```

module-name
The name of the module.

Examples

Given an initial target program *FPROG*, and a program *BPROG* which is loaded through a method that bypasses IDF's DBREAK trapping, you can establish the deferred breakpoints as usual with DBREAK commands. Once *BPROG* is in storage, the command TRIGGER LOAD BPROG causes IDF's deferred breakpoint processing to begin.

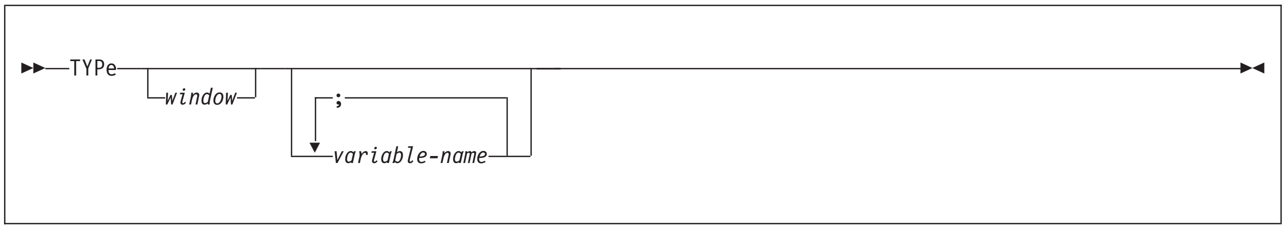
```
dbreak (bprog.bedrock)
⋮
trigger load bprog
```

Return codes

0	Operation successful
1	Missing keyword
2	Keyword truncated
5	Arguments are invalid

TYPE

Displays type information for a variable.



`window`
: An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in a LSM Information window, uses or opens the first LSM Information window.

`variable-name`
: A variable name.

  Only the variable name is relevant in determining the variable type attributes. Extra information such as:
  - Locating expressions for based variables
  - array index values
  - substring ranges

  is not needed, and should *not* be specified.

The type attribute information for the variable includes:
- fundamental data type
- user-defined data type
- type hierarchy

The type attribute information display persists until:
- A TYPE command without arguments is issued
- The window is closed with a CLOSE command.
- Another IDF Language command such as VARIABLE, STRUCTURE, ARRAY, CALLERS, PLOCATES, LANGUAGE STATUS, or MAP is issued. These commands update the LSM Information window with new information
- The target program completes execution
- Target program execution progresses beyond the variable's defined scope

Examples

```
type stuff
type var1;var2
```

UNION

The UNION command is a synonym of the STRUCTURE command. For details, see "STRUCTURE" on page 188.

UNTIL

Executes the target program up to, but not including, an address.

```
►►──UNTil──┬─────────┬─────────────────────────────────────────────►◄
           └─address─┘
```

address
  The address at which IDF should stop. Supply the *address* as part of the command, or by the cursor position.

No exit routine processing is performed during, or at the completion of, the UNTIL command. As with MRUN or MSTEP, if the UNTIL command is issued within a macro, you must use EXTRACT EVENT to determine what type of event completed the command.

z/VM If you issue UNTIL from within an IDF macro, you should make sure that it is issued through the LPSW Fastpath addressing environment. This is the default environment established when the IDF macro is entered. Using this interface eliminates any SVC linkages between REXX and IDF, so that IDF can provide optimum flexibility in what the target program can itself execute under UNTIL. See the usage notes under "MRUN" on page 149 for further details.

Return codes

0	Operation successful
1	No address specified
5	Syntax or other error in expression
6	Command issued before IDF initialization is complete

UP

The UP command is a synonym of the PREVIOUS command. For details, see "PREVIOUS" on page 162.

VALUE

Evaluates an expression and displays its value.

```
►►──VALue──┬─────────┬─────────────────────────────────────────────►◄
           └─address─┘
```

address
  An IDF address expression.

The value is displayed in hexadecimal format, symbolic format, if applicable, and decimal format.

For information about expressions, see "Address expressions" on page 80.

Return codes

0	Operation successful
5	The expression contained an error

VARIABLE

Displays the contents of one or more variables.

```
►►──VARiable──┬────────┬──┬──────────────────┬────────────────────────────────►◄
              └─window─┘  │ ┌─;────────────┐ │
                          │ ▼              │ │
                          └───variable-name─┴─┘
```

window
An LSM Information window. Select by a Window Specification, or by placing the cursor in the window. If omitted and the cursor is not in a LSM Information window, uses or opens the first LSM Information window.

variable-name
A variable name.

Define simple variables by name only. You can define based variables by name only, in which case the declared basing is used by IDF, or you can specify an explicit locating expression.

See "Variable expressions" on page 87 for a complete description of the syntax of the expressions used for VARIABLE variable name arguments.

If you supply no variable name, the nominated window is closed.

The variable display persists until:
- A VARIABLE command without arguments is issued
- The window is closed with a CLOSE command.
- Another IDF Language command such as STRUCTURE, ARRAY, TYPE, CALLERS, PLOCATES, LANGUAGE STATUS, or MAP is issued. These commands update the LSM Information window with new information
- The target program completes execution
- Target program execution progresses beyond the variable's defined scope

If the contents of the variable change while the program is running to a breakpoint, the changed data is shown on the screen when the breakpoint is reached.

You can change the displayed data by overtyping it.

In EBCDIC display mode, character data equal to X'FF' or below X'40' are displayed as a period character.

In ASCII display mode, character data which does not correspond to a displayable EBCDIC character are displayed as a period character.

If a based variable was respecified, the current basing specification is used.

Note: The display of the contents of the variables may be incorrect if the PSW indicates that execution is in the middle of a statement. This is because the variable may be in a transitional state, not having yet achieved its new value. Variable contents are only certain at the start and end of a statement.

Examples

```
var stuff
var addr(x'20000')->stuff
var addr(12(R2))->ptr->stuff(2)
var ptr->ptr2->stuff
var ptr(3)->ptr->stuff
var stuff(-5)
var stuff(1:10)
var stuff(1::25)
var var1;var2
var var1 ;ptr->stuff
var stuff(1:10) ; stuff(30:40)
var chrarray(15,1:10);chrarray(15,1::10)
var chrarray[15,0:9];chrarray[15,0::10]
```

VCHANGE

A special-purpose command used for command logging support.

```
►►──VChange──────────────────────────────────────────────────────────────────►◄
```

When command logging is active, and a field within a window which is *not* an LSM Information window is changed by operator overtyping, a VC entry is generated in the command log. This reproduces the change when the command log is played back with the RLOG command.

For more details on command logging support, see “Command record and playback features” on page 80.

VERSION

Displays the IDF version information.

```
►►──VERsion──────────────────────────────────────────────────────────────────►◄
```

The version number is in the form:

```
ASMIDF Vn.Rn.nn (generated ccyy.mm.dd hh:mm)
```

Return codes

0	Operation successful

VS

Special-purpose command used for command logging support.

```
►►──VS───────────────────────────────────────────────────────────────────────►◄
```

When command logging is active, and a field within an LSM Information window is changed by operator overtyping, a VS entry is generated in the CMDLOG file which reproduces the change when the command log is played back with the RLOG command.

For more details on command logging support, see “Command record and playback features” on page 80.

VSEP

Enables and disables blank line separating variables.

```
►►──VSEP──┬─ON──┬──────────────────────────────────────────────────────►◄
          └─OFF─┘
```

ON Enable the blank line separating variables.

OFF
Disable the blank line separating variables.

WATCH

Supplies a condition that must be true for a particular breakpoint to take effect.

```
►►──WATch──┬──────────────────────────────────────────────────┬───────►◄
           └─address──┬────────────────────────────────────┬──┘
                      └─;──comparator──instruction──┬────────────┬┘
                                                    │ ┌────────┐ │
                                                    └─▼─|─command─┴─┘
```

address
The address of the breakpoint that is to be conditional.

Supply as part of the command, or from the cursor position. If supplied by means of cursor position and a watchpoint condition is specified, the condition must be preceded by a semicolon. If the *address* is supplied by means of cursor position and the command is a query (no condition supplied) the semicolon need not be supplied.

comparator
The condition being checked. Must be:

```
= EQ ¬= NE > GT < LT >= GE <= LE
```

If the comparator and following parameters are not supplied the WATCH command is a query. In this case, the existing watchpoint condition for the address is displayed, or a message indicates that no watchpoint is active at the address.

instruction
A System/370 comparison instruction. This must be one of the following:

CR	Compare register
CLR	Compare logical register
C	Compare
CL	Compare logical
CH	Compare halfword
CLM	Compare logical characters under mask
CLC	Compare logical characters

This instruction is coded using standard assembler syntax, with a few exceptions that are detailed below.

The operation of the watchpoint is that when the watchpoint is encountered, the instruction you specified on the WATCH command is executed. If the result of the comparison instruction matches the comparator you specified, the watchpoint condition is considered true and execution of the target program stops, otherwise the target program continues to execute as if there was no breakpoint at the specified address. When the breakpoint is taken, if there is a list of commands associated with the watchpoint, they are executed before control is returned to you. These commands are specified at the end of the WATCH command, separated from the comparator instruction and each other by vertical bars (|). If a command receives a non-zero return code, the remaining commands in the list are not executed.

For example, if you want to stop execution of the target program at the instruction labeled LOOP only when R15 is nonzero, you can issue the following WATCH command and then start the target program through the RUN command:

```
watch loop; ne c r15,=f'0'
```

Departures from standard assembler syntax are:

- When specifying a register, you must use the Rn or rn notation. If you only supply the number of the register it is not recognized.
- Literal data is supported for the F, H, X, and C data types. The maximum length of literal data is 50 bytes. Only one literal value may be specified when using a CLC instruction. Comparing two literals is meaningless as a watchpoint condition.
- When specifying a symbolic address, do *not* also specify a base register, since the result is to access the specified address in storage as indexed by that register.

command
A command that is issued when the breakpoint is taken.

While the condition is false, control passes through the watchpoint and the target program is not stopped. But when the condition is true and the target program is at the breakpoint address, the target program is stopped for inspection.

Watchpoints are cleared by issuing a BREAK command or a SET BREAK OFF command for the address in question.

When you issue a WATCH command that specifies a CLC instruction, the length actually used in the comparison is shown to the right of the watchpoint condition if you query the watchpoint.

If a WATCH command with a condition is issued for an address at which there is a breakpoint, that breakpoint is *converted* into a watchpoint. If commands were specified with the original breakpoint, they are associated with the watchpoint unless commands were specified with the WATCH command.

If a watchpoint whose condition is true is placed within the execution path of the subroutine to be skipped, execution stops at that watchpoint.

Examples

To set a conditional breakpoint at location X'20040', to be taken when the contents of R3 are equal to the word in ARRAY within CSECT TEST as indexed by R1:

```
watch x'20040' ; = c r3,(test)array(r1)
```

To set a conditional breakpoint at TEST, to be taken if R3 and R4 are not equal:

```
wat test;ne cr r3,r4
```

To set a watchpoint at LOOP, to be taken when the 5 characters pointed to by R1 are thing, any of the following could be used:

```
watc  loop; = clc 0(5,r1),=c'thingxxx'
watch loop ; eq clc =c'thing',0(r1)
wat   loop; = clc 0(r1),=c'thing'
```

Return codes

0	Operation successful
3	Keyword not recognized
1	No address specified
5	Syntax or other error in expression
6	Specified location does not contain a valid instruction, or breakpoint table full

WHERE

Displays the symbolic name for the given address.

```
►►──WHEre──┬─────────┬──────────────────────────────────────────────────►◄
           └─address─┘
```

address
A storage location.

If an expression is present on the command line, that expression is used as the argument. If the command line is empty, an attempt is made to determine the address from the cursor position. Failing this, the current execution address is obtained from the PSW.

The WHERE command is like the VALUE command:

- If the address corresponds to a location within a module defined to IDF, both commands display the same symbolic name.
- If the address corresponds to an address within a module which is defined in a system control block but not yet defined to IDF:
  - the WHERE command shows this module in the symbolic name
  - the VALUE command displays the hexadecimal address value
- If the address corresponds to a location which is not within any known module, both commands display the hexadecimal address value.

For information about expressions, see "Address expressions" on page 80.

Return codes

0	Operation successful
5	The expression contained an error

XEDEXIT (CMS only)

XEDIT the current exit routine.

```
►►──XEDexit─────────────────────────────────────────────────────────────►◄
```

The name of the exit routine that is considered current is by default EXIT. It may be reset at IDF invocation by means of the EXITEXEC option. It may also be reset by a macro with the SET EXITEXEC command.

Be careful when executing CMS commands from within XEDIT. If you are debugging a user-area program, and you invoke another program that runs in the user area while in XEDIT, it destroys the debugging environment. The same care should be taken with routines that run in the transient area.

Return codes

0 Operation successful

ZONED

Sets or queries the format in which the data for zoned decimal variables are displayed.

```
►►──ZONed──┬─────────┬──────────────────────────────────────────────────►◄
           ├─DECimal─┤
           ├─*───────┤
           └─HEX─────┘
```

DECIMAL
Zoned decimal variables are displayed in decimal. This is the initial value.

***** Zoned decimal variables are displayed in the initial format (DECIMAL).

HEX
Zoned decimal variables are displayed in hexadecimal.

If the display format setting is not specified, the current display format for zoned decimal variables are shown in a message.

Return codes

0	Operation successful
2	Keyword truncated
5	Not valid zoned decimal variable display format

Part 3. Advanced topics, macros, profiles, exit routines

Chapter 11. Writing an IDF profile

You can write a profile macro for IDF to change the display colors, PF key assignments, default open windows, set predefined breakpoints for a given program, or perform other functions.

You do not need to have a profile macro, and if you do have one it can be as simple or exotic as you like. One example is to get the name of the program that is being debugged, use it coupled with the CMS NAMEFIND command to get information about that program, and automatically set a number of breakpoints that apply to it.

Regardless of what uses you have planned for it, the profile must be written in the REXX language.

z/VM The profile must have a file type of ASM and can reside on any accessed disk.

z/OS The profile must be a member of the partitioned data sets (PDSs) allocated to the ASM DD name.

z/VSE The profile must reside in a member of a sublibrary in the active PROC chain.

The default file name is PROFILE, but you can specify another file name with the PROFILE option. It is invoked by IDF just before the first display is presented (for more specific information about when the profile is executed, see "When the PROFILE is executed.") The default ADDRESS environment when your profile begins execution is ASM. IDF does not examine the return code from your profile.

To change IDF parameters, use the SET command. You can examine the current settings with the EXTRACT command. You can issue any IDF command from within your profile (though some may not make sense at that point) and you can invoke IDF macros from within your profile.

Here is an example of a simple profile:
This profile begins by changing the display color assignments. It then opens the Current Registers

```
/* PROFILE ASM */
'set color wryg'
'regs'
'disasm'
'dump'
'dumpmode'
exit
```

Figure 24. Example of a simple PROFILE

window, the Disassembly window and the Dump window, inverts the default dump mode (symbolic or unformatted), and exits.

When you issue an IDF command in your profile, it behaves exactly as if you had pressed that PF key.

Warning: Be careful when executing CMS commands from within your profile. If you are debugging a user-area program, and your profile invokes another program that runs in the user area, it will destroy your debugging environment. The same care should be taken with routines that run in the transient area.

When the PROFILE is executed

The following sequence of events occurs when IDF is invoked:

1. Perform the first scan of the argument string. The objective during the first scan is to determine which profile should be executed. These options are processed during the first scan:
   AMODE24
   AMODE31
   AMODE64

   MODE (CMS only)
   NOPROFIL
   PROFILE
2. Execute the specified PROFILE.
3. Perform the second scan of the argument string. The objective during this scan is to set the specified override options. All options not listed above are processed during this scan.
4. If the user's profile did not issue a LOAD command, load symbols and module.
5. Initialize the screen manager.

Command restrictions related to PROFILE execution

During its initialization phase, IDF loads the target program, loads any symbol definitions from the MAP file or load module, and initializes its screen driver. Whether or not these events have occurred determines whether some IDF commands may be issued. The target program and its symbols can be loaded from within the PROFILE by means of the LOAD command; if this is not done explicitly, it is done automatically by IDF after the PROFILE has completed. The screen driver is not initialized until after the PROFILE has completed.

The following commands may only be issued *before* the target program and associated symbols are loaded:

- **Program-oriented:**
  - SET LIBE fn/$
  - On CMS Only
    - SET MODMAP ON
    - SET NOMODMAP ON
    - SET NUCEXT ON
    - SET SYSTEM ON
    - SET TRANS ON
  - On z/OS Only
    - SET SVC97 ON
    - SET NOSVC97 ON
- **Symbol-oriented:**
  - On CMS Only
    - SET SELFNUCX SYMBOL

The following commands may only be issued *after* the target program and associated symbols are loaded:

- **Program-oriented:**
  - EXTRACT ADSTOPS (CMS only)
  - EXTRACT BREAK
  - EXTRACT ALET
  - EXTRACT AREGS
  - EXTRACT LOAD
  - EXTRACT REGS
  - EXTRACT REGSTOPS (CMS only)
  - SET ADSTOPS (CMS only)
  - SET ALET
  - SET AREG
  - SET BASE
  - SET BREAK
  - SET EPOFFSET
  - SET FPR
  - SET GPR
  - SET LOC
  - SET PSW

  - SET PSWSTEAL (CMS only)
  - SET REGSTOP (CMS only)
  - SET SIZE
  - SET FASTPATH ON
  - ADSTOPS (CMS only)
  - BREAK
  - RUN
  - STEP
- **Symbol-oriented:**
  - EXTRACT SELFNUCX (CMS only)

The following command may only be issued *before* the PROFILE has completed:
- MODE (CMS only)

The following commands may only be issued *after* the PROFILE has completed:
- EXTRACT ARGUMENT
- EXTRACT CURSOR
- SET CURSOR
- SWAP

Chapter 12. Writing IDF macros

You can tailor IDF through the use of REXX programs (macros) that IDF invokes. These macros have a default ADDRESS of "ASM".

z/VM

The default ADDRESS environment upon entry to an IDF macro is in the form of a PSW. This instructs REXX to use a LPSW to invoke IDF commands directly, whereas the "address IDF" environment requires a CMSCALL linkage to IDF commands.

The "ASM" subcommand environment is still provided to your macros, and it is still perfectly fine to explicitly direct most IDF commands through this named environment. However, the MRUN and MSTEP commands "behave better" if they are directed through the default "LPSW fastpath" interface.

See the usage notes under “MRUN” on page 149 for MRUN considerations, and see “REXX linkage considerations” on page 208 for details about the available methods for invoking IDF commands from REXX.

Most IDF commands can be issued within a macro. In particular, you can use the SET and EXTRACT commands to set or extract the settings of various parameters, and the target program's memory and registers.

IDF commands may be abbreviated as shown in Appendix C, “Abbreviations,” on page 285. There are many potential uses for IDF macros. A few examples are:
- Provide a "swap" PF key to swap to another set of PF key definitions
- Set a number of predefined breakpoints for a program you are testing repeatedly
- XEDIT the listing file for the program you are debugging
- Redefine the **ENTER** key so that you can use your own syntax for expressions

An IDF macro:
- On CMS, can have any file name that you like, but the file type must be ASM.
- On z/OS, it must be a member of the PDSs allocated to the ASM DD name, with any member name that you like.
- On z/VSE, it must be a member in a sublibrary in the active PROC chain.
- Must be written in the REXX language. It may optionally be compiled by the REXX compiler, in which case the file type (DD name on z/OS) of the compiled EXEC must be ASM.

You can invoke a macro in several ways:
- Assuming that your **ENTER** key retains the default assignment of COMMAND, you can enter MACRO, followed by the name of your macro and any arguments it requires, on the command line and press **ENTER**.
- If you have specified the IMPMACRO option, you can type the macro's name on the command line (without prefixing it with MACRO), followed by any necessary arguments, and press ENTER.
- You can set a PF key to `MACRO myname args` and press the PF key.
- You can issue the macro from within your PROFILE macro.

Your macro can issue any IDF commands you like. These are treated as if they had been invoked by means of a PF key.

Here is an example of a special purpose macro that sets up some breakpoints, opens a Disassembly window, and runs the target program to the first breakpoint:

```
/* ALLOPEN ASM */
'set break on loop'
'set break on exit'
'disasm'
'run'
exit 0
```

Figure 25. Example of a special purpose macro

IDF propagates the return code from your macro to the caller if it is invoked by another IDF macro.

If the top level macro exits with a nonzero return code, IDF issues a message, so if you want to perform your own message handling you need to exit with a return code of zero.

There are times when one of several macros is setting an IDF option, and it is difficult to determine just where the option is being set. The MACROLOG option helps debug this sort of macro problem. Whenever this option is in effect, all IDF commands that are issued by either an IDF macro or a IDF exit routine are written to the macro log. See "MACROLOG" on page 31 for the name of this log.

Warning: Be careful when executing CMS commands from within your macro. If you are debugging a user-area program, and your macro invokes another program that runs in the user area, it destroys your debugging environment. The same care should be taken with routines that run in the transient area.

Numbers in expressions can be specified in explicit (X'123', F'123') or implicit (123) notation. Numbers that do not explicitly specify the base are evaluated according to the current setting of the HEXINPUT option. When writing macros it is recommended that you use explicit base notation.

REXX linkage considerations

This section describes the linkages that are used by IDF to invoke REXX (for execution of an IDF macro), and for REXX to in turn invoke IDF commands (as specified within that macro). You should consult the formal REXX documentation and its online HELP files for more complete details of these topics. Here, only a few of the important basics are discussed.

The REXX ADDRESS statement

The REXX language provides you with explicit and detailed control over how REXX should direct your commands to the appropriate operating system or application component for execution. This "command routing" is controlled by the REXX ADDRESS statement.

Rather than attempt to repeat all of the available REXX literature on this subject, here are a few examples to show how ADDRESS can be used in a REXX exec.

Initial or default ADDRESS environment

Upon the initial entry to your exec, REXX provides a default ADDRESS. This is used for any commands until the addressing is changed. For a normal EXEC file, the default addressing supplied by REXX is to the "operating system" so that commands issued within the exec are by default routed to the underlying operating system for execution.

For IDF macros, the default addressing environment is supplied by IDF when it invokes REXX to execute the macro.

z/VM The default ADDRESS is in the form of a PSW, telling REXX that it should use its "LPSW Fastpath" mechanism to directly enter IDF command handling.

z/OS The default ADDRESS is ASM"

z/VSE The default ADDRESS is "ASM"

Overriding the default ADDRESS Environment

When writing an IDF macro, you may find that you need to issue a command to the underlying operating system, not a command to IDF. You do this by using the REXX ADDRESS statement.

For example:

```
address CMS CMS-command
```

This directs only this command to CMS, with subsequent commands directed to the default environment (ASM).

Compare this with:

```
address CMS
CMS-command
```

This directs *all* commands to CMS until a subsequent `address` command overrides it.

"Which addressing environment is best for the default?" is generally a decision based upon how many IDF commands are required versus how many host commands are required. And of course you are not restricted to setting one default environment and leaving it that way until the end of your macro; you can change it again at any time.

Of course in an IDF macro, it is expected that most of the commands are IDF commands. And in this case, it is probably best to just leave the default addressing environment (as supplied at entry to your macro) alone, and make "temporary overrides" to other names as you may require for the few (if any) host commands issued.

Saving and restoring an ADDRESS environment

When making a “permanent change” to the default addressing environment, it is often required that later in your exec, you restore the original environment again. This can be done in one of three basic ways:

1. By hard-coding a switch from “address CMS” to “address ASM” again, assuming you know that the original default was “ASM”.
2. By exploiting a little-known feature of REXX that an “address” statement with no operands “pops” the previous addressing environment. REXX saves just one recent environment this way, but that is often all that is needed.
3. By explicitly saving and restoring the environment. The current environment can be saved by the REXX ADDRESS() function call, as for example `saveenv = address()`. It can later be restored by another form of the ADDRESS statement, using ADDRESS VALUE, as for example `address value saveenv`. In this case, the `saveenv` is taken to be a REXX variable which holds the environment that is to be set.

There is also some interesting save and restore logic within REXX itself, when calling internal procedures. That topic is beyond the scope of this short introduction to ADDRESS.

z/VM

Since the default ADDRESS environment on entry to IDF macros on CMS is in the form of a PSW, certain special precautions should be made when switching from one ADDRESS mode to another. In particular, be aware that the PSW format is a non-printable hexadecimal eight bytes, which can contain a X'40' byte that might be interpreted as a blank (causing a REXX syntax error perhaps) if misused.

Procedures numbered 2 and 3 in the “Saving and Restoring” examples above are entirely safe for manipulating a PSW form of environment. Procedure 1 depended on the initial environment being “ASM”, which is not the case with IDF on CMS, so it should be avoided.

In most cases, there is no need to make a permanent change to the default addressing. You might have just a couple of host commands that warrant a temporary “address CMS” override. So you do not have to worry at all about any of this save and restore complexity.

IDF on CMS still supplies a usual “ASM” subcom environment, so that “address ASM 'WHATEVER'” is supported on TSO, z/VSE, and CMS. And in almost all cases, it is perfectly acceptable to address IDF commands in this way. Make sure that the “LPSW Fastpath” environment is used for an MRUN command. See “MRUN” on page 149 for further information and cautions.

Example macros

Here are some sample macros that illustrate what may be done with macros.

EX

This macro determines and display the target of an EXECUTE instruction. It is meant to be invoked with a PF key, but it can also be invoked from the command line with an address.

```
/*REXX --------------------------------------------------------------*/
/*                                                                   */
/* EX      - DISASM the instruction which is the target of an        */
/*           EXecute instruction and display the results in a        */
/*           message.                                                */
/*                                                                   */
/*   If the cursor is positioned on an EXecute instruction, or       */
/*   the cursor is on the command line but the address of a valid    */
/*   EXecute instruction has been entered, or if the PSW points      */
/*   to a valid EXecute instruction, dis-assemble the target of      */
/*   EXecute instruction and display the results as a message.       */
/*                                                                   */
/*-------------------------------------------------------------------*/

/*-------------------------------------------------------------------*/
/* Determine which instruction is involved.                          */
/*                                                                   */
/* If EXTRACT ARGUMENT gives RC¬=0, IDF will already have issued     */
/* an error message.  In this case, we exit with RC=-3 to prevent    */
/* IDF from overlaying that message with another that says "MACRO    */
/* RC=xx" if the argument came from the command line.                */
/*-------------------------------------------------------------------*/

 'EXTRACT ARGUMENT'
 If RC ¬= 0 Then
   Exit -3                         /* RC=-3 to prevent msg overlay  */

 If source = '' Then               /* no address provided           */
   Do
     'EXTRACT VALUE 0(PSW)'        /* use next instruction address  */
     field = Word(EXPR, 1)
   End

 address = field                   /* field level resolution        */
 'EXTRACT DISASM X'''address''''
 If RC ¬= 0 | instr = '' Then      /* not a valid instruction       */
  Do
    'SET MSG Location' address 'does not contain a valid EXecute',
                         'instruction'
    'SET ALARM'
    Exit
  End
 Parse Var instr 13 index 14 . 15 base 16 offset 19 . 35 opcode .
/*------------------------------------------------------------------*/
/* The opcode must be EX (execute) for this macro.                  */
/*------------------------------------------------------------------*/
 If opcode ¬= 'EX' Then
   Do
     'SET MSG' STRIP(opcode) 'is not a valid target for the EX macro'
     Exit
   End

/*------------------------------------------------------------------*/
/* Now convert the hex index, base, and offset to an expression      */
/*------------------------------------------------------------------*/
 reghex = '123456789ABCDEF'
 oper   = "X'"offset"'"

 If index ¬= '0' Then
   oper = oper'+0(R'index(reghex,index)')'

 If base ¬= '0' Then
   oper = oper'+0(R'index(reghex,base)')'

/*------------------------------------------------------------------*/
/* Dump the indicated memory area and exit                          */
/*------------------------------------------------------------------*/
 'Extract Disasm' oper
 'SET MSG' instr

 Exit
```

REGS

This macro toggles the Current Registers window, and if it is opened, places it at the top of the screen.

```
/*REXX -------------------------------------------------------------*/
/*                                                                  */
/* REGS    - Toggle the current registers window.                   */
/*                                                                  */
/*   When the REGS window is opened, it will be moved on the IDF     */
/*   display so that it is the first window.                        */
/*                                                                  */
/*------------------------------------------------------------------*/

 'Regs'                             /* Toggle REGS window            */

 'Extract Cursor'                   /* Obtain window information     */
 n = Find(display,'REGS')           /* Is REGS window present?       */
 If n ¬= 0 Then                     /* Yes?  Force to be 1st window  */
   'Order ='n

 Exit
```

SYSCMD

This macro issues a command to the operating system from the IDF command line. On CMS, transient modules, nucleus extensions and EXECs run. On TSO, any unauthorized command or REXX EXEC runs. On z/VSE, REXX/VSE commands or REXX programs run.

```
/*REXX -------------------------------------------------------------*/
/*                                                                  */
/*   Issue the command to the system.                               */
/*                                                                  */
/*------------------------------------------------------------------*/

 Parse Arg cmd                      /* Obtain any operands           */
Parse Upper Source OpSys .         /* Determine current OS          */

Select

  When (OpSys = 'TSO') Then        /* Handle TSO command            */
    Do
      Address TSO cmd              /* Issue command                 */
      trc = rc
      'Refresh'                    /* Refresh WDB screen            */
    End

  When (OpSys = 'CMS') Then        /* Handle CMS command            */
    Do
      Address CMS cmd              /* Issue command                 */
      trc = rc
    End

  When (OpSys = 'VSE') Then        /* Handle VSE command            */
    Do
      Address VSE cmd              /* Issue command                 */
      trc = rc
    End
  Otherwise
    Do
      'SET MSG Not on TSO or CMS or VSE - Unable to issue command'
      trc = -3
    End

End

Exit trc
```

Chapter 13. The IDF exit routine

An exit routine (also called “exit exec”) is a special purpose IDF macro, which runs when certain events occur.

For general instructions on writing a REXX macro, see “REXX linkage considerations” on page 208. For instructions on writing a compiled-language exit routine, see “Writing a compiled-language IDF exit routine” on page 215.

Naming the exit routine

The default name of the macro that serves as the exit routine is "EXIT". Change it at IDF invocation with the EXITEXEC option, or during IDF execution with the SET EXITEXEC command.

Controlling exit routine processing

Exit routine processing is enabled and disabled by the EXITEXEC command.

If exit routine processing is enabled, and the currently defined exit routine exists, then when one of the following events occurs, the exit routine is invoked to determine whether or not to notify the operator:

- IDF is preparing to present its first screen display
- A breakpoint is reached
- The target program branches outside of its memory limits
- A condition exists which puts IDF at risk of losing control
- A program check occurs
- The target program experiences a CMS, z/OS, or z/VSE ABEND
- The target program has completed and returned control to IDF
- A single-step operation has just completed
- On CMS only:
  - SVC tracing is in effect and an SVC is reached
  - A monitored storage location is modified
  - A monitored register is modified

The RUNEXIT command also invokes the exit routine

Passing the reason for invocation

When the exit routine is invoked, the first token passed to it indicates the reason it was invoked. This could be:

INIT IDF is preparing to present its first screen display.

BREAK
A breakpoint has been reached.

LIMITS
A target program has branched outside of its memory limits.

WARN
IDF is at risk because of code the target program has, or is about to, execute.

When the event code is "WARN", the last Command window message display line in the Command window is set to one of these messages:

- ASMMAI025W
- ASMMAI026W
- ASMMAI027W
- ASMMAI091W

- ASMMAI221W

See “Message numbers and severity levels” on page 263 for the text of each message, and the suggested user response.

The contents of the last message should be extracted using the EXTRACT LASTMSG command, and an appropriate action taken by the exit routine.

PROGCHK
A program check has occurred.

ABEND
The target program experienced a CMS, z/OS, or z/VSE ABEND.

PFKEY
The RUNEXIT command was issued.

DONE
The target program has completed and returned control to IDF

STEP A single-step operation was just completed.

QUIT The operator has issued the QUIT or RCQUIT command.

SVC CMS Only. SVC tracing is in effect and an SVC instruction has been reached.

ADSTOP
CMS Only. A monitored storage location has been modified.

REGSTOP
CMS Only. A monitored register has been modified.

Looking at the address

Unless the first token of the argument string passed to the exit routine is "PFKEY", it will also be passed the address at which the event occurred. The argument string passed to the exit routine can be parsed with the following REXX instruction:

```
Parse Upper Arg reason hexvalue . '(' csect ')' symbolic
```

Assume that a breakpoint occurs at label LOOP, which is at location X'2000E' in CSECT ALLOPEN, which begins at X'20000'. After this Parse instruction, the following variables are set:

REASON
BREAK

HEXVALUE
0002000E

CSECT
ALLOPEN

SYMBOLIC
ALLOPEN+14

Note: If the address is outside the target programs defined to IDF, the code section (CSECT) name is omitted and the symbolic name is a second hexadecimal value.

Ignoring the event

The exit routine indicates to IDF that the event should be ignored and execution of the target program should continue by exiting with a return code of 1. *Any* other return code indicates that the operator should be notified as usual.

Other techniques

You can use the EXTRACT command to examine the contents of the registers and main storage, and if necessary you can use host environment commands to examine other parts of the programming environment that lie outside of IDF, for example to see if a given disk file has been created.

If you need to maintain a count of the number of times the exit routine has been entered, or other status information that must carry across its invocations, you can use the SET GLOBAL and EXTRACT GLOBAL commands to access an 80-byte area within IDF that is provided for this purpose.

Warning: Be careful when executing CMS commands from within your exit routine. If you are debugging a user-area program, and your exit routine invokes another program that runs in the user-area, it will destroy your debugging environment. The same care should be taken with routines that run in the transient area.

Writing a compiled-language IDF exit routine

For some applications, like code coverage or path analysis data collection, the performance of an exit routine written in REXX is not adequate. To allow for higher-performance exit routines, IDF provides the ability to use exit routines written in compiled languages.

Specifying that an exit routine is compiled code

To tell IDF that your exit routine is written in a compiled language, set the CMPEXIT option. Here is an example profile which specifies a compiled-language exit routine:

```
/*-------------------------------------------------------*/
Trace ?O
'MACRO PROFILE PROFILE' /* execute usual profile  */
'SET EXIT ASMIDFEX'     /* define name of exit module */
'SET CMPEXIT ON'        /* indicate compiled code exit */
'SET EXITEXEC ON'       /* activate exit routine       */
Exit
```

Figure 26. Specifying a compiled-code exit routine

The compiled-code exit routine is invoked to examine the same events as an exit routine that is written in REXX.

On CMS, It is not *required* that you run IDF with PER exploitation disabled, but it *is* highly recommended for applications that collect execution data because:
1. Performance with PER=Y is much slower than with PER=N
2. IDF operation with PER=Y is not as robust as with PER=N

Requirements for compiled-language exit routines

You must issue a SET EXITEXEC command before setting the CMPEXIT option so that IDF knows the name of the exit routine.

z/VM Your compiled-language exit routine must be a NUCXLOADable CMS MODULE.

When the SET CMPEXIT ON command is issued, IDF will NUCXLOAD the exit routine.

IDF never invokes the NUCXLOADed exit routine by means of SVC-202. It obtains the entrypoint address from the SCBLOCK associated with the exit routine after it is NUCXLOADed, and invokes it with BALR.

z/OS When the SET CMPEXIT ON command is issued, IDF will load the exit with a LOAD SVC. IDF calls the exit with a BALR after having obtained its address from the LOAD of the exit.

z/VSE When the SET CMPEXIT ON command is issued, IDF will load the exit with CDLOAD. IDF calls the exit with a BALR after having obtained its address from the CDLOAD of the exit.

The compiled-language exit routine can be written in the language of your choice, so long as it is able to conform to the calling sequence used by IDF. The interface used is designed for PL/I-like exit routines, but an assembler routine can also be used.

At entry to the exit routine, R1 points to a fixed length parameter list:

```
* parameter list passed to compiled-code exit routine
         DC      A(calltype)       address of call-type (fullword)
         DC      V(subcom)         address of subcommand interface
         DC      A(psw)            address of current PSW
         DC      A(gpr)            address of current GPRs
         DC      A(varname)        address of variable name
         DC      A(varnamel)       address of variable name length
         DS      A(varval)         address of variable value
         DC      A(varvaln)        address of variable value length
         DS      A(pstring)        address of pstring
         DC      A(pstringl)       address of pstring length
```

Figure 27. Parameter list passed to compiled-code exit routine

The first parameter is the call-type. The exit routine is invoked on event occurrence, and if it issues any EXTRACT commands during its processing, it is re-entered (recursively) to receive the values normally set into REXX variables. A call-type of 0 (zero) indicates an EVENT invocation. A call-type of 4 indicates a VARIABLE invocation.

The exit routine, when entered, must inspect the call-type as its first order of business. It will not be re-entered unless is issues EXTRACT commands to the IDF command processor, but if it is re-entered the call-type value will have changed to 4, indicating a VARIABLE invocation.

The remaining parameters are either invariant, or apply to only one type of call.

All of the parameters passed and their meanings are:

1. **Address of call-type:** The call-type is contained in a fullword. A value of 0 (zero) indicates an EVENT call, and a value of 4 indicates a VARIABLE call.
2. **Address of subcommand interface:** This address is the start of the IDF subcommand processor. The exit routine may call the subcommand processor to execute IDF commands. It must enter the subcommand processor with R1 containing the address of a parameter list as follows:

   ```
            DC    A(CSTRING)
            DC    A(CSTRINGL)
            ...
   CSTRING  DC    C'whatever command string'
   CSTRINGL DC    A(L'CSTRING)
   ```

   If an EXTRACT command is issued, the exit routine must be prepared to receive the variables extracted. It will therefore be recursively re-entered and the call-type will indicate a VARIABLE call.
3. **Address of current PSW:** This word points to the target program's current PSW.
4. **Address of current GPRs:** This word points to the target program's current GPRs.
5. **Address of variable name:** (Applicable only when call-type is 4.) This word points to the start of a character string which is the name of the extracted variable.
6. **Address of variable name length:** (Applicable only when call-type is 4.) This word points to a fullword containing the length of the variable name in bytes.
7. **Address of variable value:** (Applicable only when call-type is 4.) This word points to a character string which represents the current value of the extracted variable.
8. **Address of variable value length:** (Applicable only when call-type is 4.) This word points to a fullword containing the length of the current value of the extracted variable in bytes.
9. **Address of pstring:** (Applicable only when call-type is 0.) This word points to the start of a character string describing the event causing the exit routine's invocation. The contents of the character string are described in Chapter 13, “The IDF exit routine,” on page 213.

10. **Address of pstring length:** (Applicable only when call-type is 0.) This word points to a fullword which contains the number of bytes in the parameter string which describes the event causing the exit routine's invocation.

Chapter 14. REXX variables available to macros

IDF provides the EXTRACT command so that an IDF macro can obtain information about IDF, IDF Language Support, or the target program and the environment in which it is executing.

The EXTRACT command returns data in:

- one or more fixed variables, or fixed stemmed arrays
- a stemmed array controlled by the STEM command

A REXX stemmed array will consist of "*stem.n*", where *stem.0* contains the number of items in the stemmed variable array, and the remaining items contain the requested information.

REXX variables with fixed names

The following table lists all REXX variables whose names are fixed, and the EXTRACT command that sets each variable:

Table 2. REXX variables with fixed names

Variable	Command Which Sets Variable
ADSTOP.n	EXTRACT ADSTOP
ALET	EXTRACT ALET
AR.n	EXTRACT AREGS
AREA	EXTRACT LOAD
BREAK.n	EXTRACT BREAK
COLORS	EXTRACT COLORS, EXTRACT COLOURS
COMMAND	EXTRACT CMDMSG
CPDISASM	EXTRACT CURSOR
CPDUMP	EXTRACT CURSOR
CSECT	EXTRACT DISASM
DISPLAY	EXTRACT CURSOR
EPOFFSET	EXTRACT LOAD
EVENT	EXTRACT EVENT
EXACT	EXTRACT ARGUMENT, EXTRACT CURSOR
EXITEXEC	EXTRACT EXITEXEC
EXPR	EXTRACT VALUE
FIELD	EXTRACT ARGUMENT, EXTRACT CURSOR
FPR.n	EXTRACT REGS
GLOBAL	EXTRACT GLOBAL
GPR.n	EXTRACT REGS, EXTRACT REGSTOPS
HEXCURSR	EXTRACT CURSOR
ICOUNT	EXTRACT ICOUNT
INDIRECT	EXTRACT ARGUMENT, EXTRACT CURSOR
INSTR	EXTRACT DISASM
LOADLIB	EXTRACT LOAD

Table 2. REXX variables with fixed names (continued)

Variable	Command Which Sets Variable
LSM	EXTRACT LOAD
MEMAREA	EXTRACT LOCATION, EXTRACT LOCATION ALET
MODE	EXTRACT MODE
MODULES.n	EXTRACT MODULES
MSG1	EXTRACT CMDMSG
MSG2	EXTRACT CMDMSG
NAME	EXTRACT LOAD
NINSTR	EXTRACT DISASM
NPDISASM	EXTRACT CURSOR
NPDUMP	EXTRACT CURSOR
OAR.n	EXTRACT AREGS
OFFSET	EXTRACT LOAD
OFPR.n	EXTRACT REGS
OGPR.n	EXTRACT REGS
OPSW	EXTRACT REGS
OPTION	EXTRACT OPTIONS
ORIGIN	EXTRACT LOAD
PBREAK.n	EXTRACT BREAK
PER	EXTRACT PER
PFK.n	EXTRACT PFK
PLIST	EXTRACT PLIST
PSW	EXTRACT REGS
QUALIFY	EXTRACT QUALIFY
SELFNUCX	EXTRACT SELFNUCX
SIZE	EXTRACT LOAD
SKIP.n	EXTRACT SKIPSTEP
SOURCE	EXTRACT ARGUMENT, EXTRACT CURSOR
STEM	EXTRACT LANGUAGE STEM
SVC	EXTRACT SVC
SYMBOL	EXTRACT LOAD
SYMBOL.n	EXTRACT SYMBOLS
VERSION	EXTRACT VERSION
WINDOW.n	EXTRACT WINDOWS

REXX variables with variable names

The following table lists the commands that return data in the REXX stemmed array whose name is controlled by the STEM command. See Chapter 15, "The EXTRACT command," on page 223 for more details.

The name of the stemmed variable array defaults to **LSM.** and should normally not be altered.
- EXTRACT ARRAY

- EXTRACT CALLERS
- EXTRACT LANGUAGE ARGS
- EXTRACT LANGUAGE CMDS
- EXTRACT LANGUAGE COMMANDS
- EXTRACT LANGUAGE OPTIONS
- EXTRACT LANGUAGE STATUS
- EXTRACT LANGUAGE VERSION
- EXTRACT MAP
- EXTRACT MSTATUS
- EXTRACT NAMES
- EXTRACT PARMS
- EXTRACT PLOCATES
- EXTRACT QUERY
- EXTRACT SCOPE
- EXTRACT SCRVAR
- EXTRACT SOURCE
- EXTRACT STRUCTURE
- EXTRACT TYPE
- EXTRACT VARIABLE
- EXTRACT VDCL
- EXTRACT VDECLARE
- EXTRACT VLOC
- EXTRACT VVALUE

Chapter 15. The EXTRACT command

The EXTRACT command is used to return various types of information to a macro through REXX variables or stemmed arrays. This includes:

- IDF options
- IDF settings
- IDF invocation parameter string
- user register values
- user storage values
- user variable data
- user program information
- user symbol information

The items in the rest of this section describe the options of the EXTRACT command.

Return codes

The EXTRACT command has these return codes:

0	Operation successful
1	Missing keyword
2	Keyword truncated
3	Keyword unknown
4	Error attempting to set a REXX variable Out of memory or EXECCOMM error
5	Arguments are invalid
6	Unable to honor EXTRACT command at this time This return code will occur if, for example, you attempt to `EXTRACT BREAK` within your PROFILE before the target program has been loaded into memory.
7	The command cannot run in this environment This may be because this command is supported for a specified OS only (CMS, or TSO), or that you must be in an ESA environment.

ADSTOPS (CMS only)

Returns all the currently set storage modification stops.

```
►►──EXTract──ADSTops──────────────────────────────────────────────────────►◄
```

REXX variables set

ADSTOP.0
    Number of AdStops set

ADSTOP.n
    The next AdStop

ALET

Return the ALET used to qualify the dataspace to be displayed in a Dump window.

```
►►──EXTract──LOCATIon──ALET──alet──number-of-bytes──start-address──────────────────────────►◄
```

window
: A Dump window Select by a Window Specification, or by placing the cursor in it. If omitted and the cursor is not in a Dump window, returns the ALET for the first Dump window.

You must be in an ESA environment for this command to work.

REXX variables set
ALET The ALET of the selected window in EBCDIC HEX representation

Example

```
EXTRACT ALET =3
```

AREGS

Returns the Access Registers in EBCDIC HEX representation.

```
►►──EXTract──AREGs─────────────────────────────────────────────────────────────────────────►◄
```

You must be in an ESA environment for this command to work.

REXX variables set
AR.0 - AR.15
: Current values of the Access Registers

OAR.0 - OAR.15
: Previous values of the Access Registers

ARGUMENT | ARGS

Returns address argument from the command line or cursor position as appropriate.

```
►►──EXTract──┬─ARGument─┬──┬──────────┬────────────────────────────────────────────────────►◄
             └─ARGs─────┘  └─argument─┘
```

argument
: An expression. If it is invalid, the EXTRACT ARGUMENT command completes with a nonzero return code. If EXTRACT ARGUMENT completes with a nonzero return code, an appropriate error message is placed on the screen by IDF and the macro should exit with RC=-3, to stop IDF overlaying the error message with another message stating the macro return code (if the macro exits with a nonzero RC), or clearing the command line (if the macro exits with RC=0).

If the command line is empty, IDF attempts to determine an address based on cursor position.

REXX variables set

SOURCE
: Contains the source of the argument:

blank	No argument was available
COMMAND	The argument was obtained from the command line
PSWGPR	The argument was obtained from cursor position within the Current Registers window or the Old Registers window
DISASM	The argument was obtained from cursor position within the Disassembly window
DUMP	The argument was obtained from cursor position within the Dump window

FIELD 8-character hexadecimal address; if determined from cursor position, the address corresponds to the first byte of the field in which the cursor was positioned.

EXACT
8-character hexadecimal address; if determined from cursor position in a dump display, the address corresponds to the exact byte of the field in which the cursor was positioned. Otherwise EXACT is the same as FIELD.

INDIRECT
8-character hexadecimal address. If determined from the cursor position in the Disassembly window, and the cursor was positioned at the first instruction shown, and that instruction is a branch, the value in INDIRECT is the effective address of the branch instruction. If determined from cursor position in a Dump window, and the cursor was in the first field displayed, and that field is a fullword, INDIRECT will contain the contents of the field rather than its address. Otherwise INDIRECT is the same as FIELD.

If the argument was determined from the command line, variables FIELD, EXACT, and INDIRECT will contain the same value.

EXTRACT ARGUMENT follows the rules for obtaining arguments that are described in “Arguments and cursor positioning” on page 83.

ARRAY

Returns information about the indicated array elements.

```
                          ┌─;─────────────────┐
                          ▼                   │
►►──EXTract──ARRay───────array-element-name───┴──────────────────────────────────────────►◄
```

array-element-name
An array element name.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Information about the array elements

BREAK

Returns all the currently set breakpoints or, if an expression was specified, just the breakpoint at that address.

```
►►──EXTract──BREak──┬────────────────────┬──────────────────────────────────►◄
                    └─breakpoint-address─┘
```

breakpoint-address
The address of the breakpoint that is required.

REXX variables set

BREAK.0
Number of breakpoints set

BREAK.n
The next breakpoint

PBREAK.0
Number of PSWSTEAL breakpoints set

PBREAK.n
The next PSWSTEAL breakpoint

CALLERS

Returns information for each generation in the program caller hierarchy, including:

- Current execution location, as:
  - Memory location, in IDF symbolic format

    `(module.CSECT)Stmt#nnnnn+offset`
  - Logical location

    `program-block-name+offset`
- Save Area Header
- Save Area register values, if applicable

```
                       ┌─*──────────────────────────┐
►►──EXTract──CALlers──┼────────────────────────────┼───────────────────────►◄
                       │ ┌─;──────────────────────┐ │
                       │ ▼                        │ │
                       └───program-caller-generation─┴─┘
```

*   Return information for all caller generations.

program-caller-generation
A caller generation.

The program caller generations numbering convention is:

0 Current program

1	Parent (caller)
2	Grandparent (caller of caller), and so on

The SAREGS command (see “SAREGS” on page 172) is used to enable or disable the return of the Save Area header and registers in the CALLERS data. The Save Area registers are formatted according to the ROWSTYLE option setting.

The SALIMIT command (see “SALIMIT” on page 172) controls the maximum depth of the CALLERS data. This is intended to prevent problems when the program call chain is damaged, or is of unexpected depth (due to runaway recursion).

The name of the stemmed variable array defaults to LSM. and is set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) controls the maximum stemmed array index value.

REXX variables set

stemname.**0**
  Number of items in the stemmed array

stemname.**n**
  Information about the program caller hierarchy.

CMDMSG

Returns the current contents of the command line and message lines as currently displayed in the IDF Command window.

```
►►──EXTract──CMDMsg──────────────────────────────────────────────────────────►◄
```

Also see “LASTMSG” on page 236.

All messages generated by IDF are returned with message prefixes, even if the MSGID option is OFF and the message is displayed without the message id. User messages issued with a SET MSG command do not have a message id.

REXX variables set

COMMAND
  The current value of command line

MSG1 The current value of message line 1

MSG2 The current value of message line 2

COLORS

Returns the current color settings.

```
►►──EXTract──┬─COLors──┬─────────────────────────────────────────────────────►◄
             └─COLours─┘
```

REXX variables set

COLORS
: The current color settings (see “COLORS” on page 106 for format).

CURSOR

Returns information about the current position of the cursor.

```
►►──EXTract──CURsor──────────────────────────────────────────────────────────►◄
```

REXX variables set

DISPLAY
: Contains a list of the currently open windows. It is a series of blank separated words with the following meaning:

  ADSTOP
  : The AdStops window.

  AFPR The Additional Floating-Point Registers window.

  BREAK
  : The Break window.

  DISASM
  : A Disassembly window.

  DUMP
  : A Dump window.

  LSMINFO
  : An LSM Information window.

  OREGS
  : The Old Registers window.

  REGS The Current Registers window.

  SKIP The Skipped Subroutines window.

  STATUS
  : The Target Status window.

SOURCE
: Contains the source of a cursor-derived argument address:

  blank No cursor-derived argument was available

  DISASM
  : Cursor was in the Disassembly window

  DUMP
  : Cursor was in the Dump window

  PSWGPR
  : Cursor was in the Current Registers window or the Old Registers window

FIELD Symbolic address as described in “Symbolic addresses” on page 229, corresponding to the first byte of the field in which the cursor was positioned.

EXACT
: Symbolic address as described in “Symbolic addresses” on page 229, corresponding to the exact byte of the field in which the cursor was positioned if the cursor was in a dump field; otherwise EXACT is the same as FIELD.

INDIRECT
: Symbolic address as described in “Symbolic addresses” on page 229. If the cursor was positioned in the disassemble display, and the cursor was positioned at the first instruction shown, and that instruction is a branch, the value in INDIRECT is the effective address of the branch instruction. If the cursor was positioned in a dump display, and the cursor was in the first field displayed,

and that field is a fullword, INDIRECT will contain the contents of the field rather than its address. Otherwise INDIRECT is the same as FIELD.

HEXCURSR
Three, blank separated, 2-character hexadecimal value which represents the exact position of the cursor on the screen. These values represent the window containing the cursor, the row within the window of the cursor, and the column within the window of the cursor.

CPDISASM
Symbolic address as described in “Symbolic addresses,” corresponding to the address of the first byte shown in the first Disassembly window.

CPDUMP
Symbolic address as described in “Symbolic addresses,” corresponding to the address of the first byte shown in the first Dump window.

NPDISASM
Symbolic address as described “Symbolic addresses,” corresponding to the address of the first byte that is shown in the first Disassembly window if it is scrolled forward.

NPDUMP
Symbolic address as described “Symbolic addresses,” corresponding to the address of the first byte that is shown in the first Dump window if it is scrolled forward.

Symbolic addresses

Symbolic addresses consist of several tokens, and can be parsed with the following REXX instruction (where NAME is the variable containing the symbolic address):

```
Parse Var name hexvalue . '(' csect ')' symbolic
```

After this Parse instruction, the following variables contain, for example:

NAME
0002000C (ALLOPEN) ALLOPEN+12

HEXVALUE
0002000C

CSECT
ALLOPEN

SYMBOLIC
ALLOPEN+12

If the address is outside of the target programs defined to IDF, the code section (CSECT) name is omitted and the symbolic name is a second hexadecimal value.

If the address is outside the currently qualified module or the FULLQUAL option is ON, then the code section name as defined above is *module.csect*, where *module* is the name of module containing the address.

EXTRACT CURSOR follows the rules for obtaining arguments that are described in “Arguments and cursor positioning” on page 83 except that it does not examine the command line for an argument.

If a Dump window is opened and the location being dumped is a 64 bit address, the variables FIELD, EXTRACT, INDIRECT, CPDUMP and NPDUMP are 16 byte fields.

DISASM

Returns information about the specified instruction.

```
►►──EXTract──DISasm──instruction-address──────────────────────────────────────►◄
```

instruction-address
Any IDF expression. It is resolved to an address and used to find the instruction to be disassembled.

REXX variables set

INSTR
Disassembled instruction text; blank if the address specified by the expression does not contain a valid instruction, otherwise:

Columns	**Contents**
01-08	Instruction address
10-24	Hexadecimal instruction value in halfwords
26-33	Label
35-39	Operation code
41 on	Operands

NINSTR
Address of the next sequential instruction, as 8-digit hexadecimal (meaningless if INSTR is blank). The address represents the "sequential flow" next instruction.

z/VM IDF knows that a fullword bitstring follows an SVC 201, that a fullword error return address may follow an SVC 202, and that a halfword code follows an SVC 203, and adjusts the returned NINSTR address accordingly. IDF also recognizes when the target program has stolen the SVC New PSW, and if so returns the entry-point address of its SVC FLIH when an SVC is disassembled.

NINSTRB
Address of the non-sequential (that is, a branching) flow instruction, if any, as 8-digit hexadecimal, or blank if NINSTR represents the only possible next instruction. NINSTRB is set for conditional branch instructions, and represents the next instruction if that conditional branch is taken.

NADDR
Address of the next instruction or data which follows the nominated instruction. This is normally the same as the NINSTR address for sequential flow. But the NINSTR may be different due to IDF's recognition (on CMS) of special flows when an SVC is nominated.

CSECT
The name of the code section (CSECT) within which the instruction occurs (meaningless if INSTR is blank).

If the address is outside the currently qualified module or the FULLQUAL option is ON, then the code section name is *module.csect*, where *module* is the name of module containing the address.

Be careful when using addresses obtained from EXTRACT DISASM in other IDF commands. Displacements in disassembled instructions may be shown in either explicit hex notation or implied decimal notation (depending on the HEXDISP option). If you use a displacement that is given in implied decimal notation, but the default input base is hex, the address will not be evaluated as expected.

The recommended procedure for working with expressions obtained from EXTRACT DISASM is to save the default base setting and set it to decimal.

The calculation of the NINSTR and NINSTRB addresses are done using the current target program register values at the time of the EXTRACT DISASM command.

For example a BRANCH instruction might use a base register of R11 for its branching. If the EXTRACT DISASM expression address resolves to the current PSW location, this is no problem whatsoever, since the register values will be correct at that point. However, if the expression represents an address other than the current PSW instruction, the register values might not match what they are at the point of naturally executing that instruction, and so the NINSTR and NINSTRB address might be incorrect.

EVENT

Returns data about the last event which occurred in the target program.

```
►►──EXTract──EVEnt──────────────────────────────────────────────────────►◄
```

REXX variables set
EVENT
Event information, in the same format as the argument string passed to an exit routine; see Chapter 13, "The IDF exit routine," on page 213 for a description of this information.
COMMAND
The name of the command which last started the target executing.

EXITEXEC

Returns the name of the currently assigned exit routine.

```
►►──EXTract──EXItexec───────────────────────────────────────────────────►◄
```

REXX variables set
EXITEXEC
The name of the currently assigned exit routine

GLOBAL

Returns the current setting of the IDF global variable.

```
►►──EXTract──GLObal─────────────────────────────────────────────────────►◄
```

This global variable is initially set to blanks by IDF. It may be reset by any IDF macro through the SET GLOBAL command, and is intended for macro-to-macro communication.

REXX variables set
GLOBAL
The last setting of the IDF global variable

GLOBAL STEM

Reads the data in the specified Global Storage stems, and write it to the same named REXX stemmed arrays.

```
                      ┌──────────┐
                      ▼          │
►►──EXTract──GLObal────stem-name.┴──────────────────────────────────►◄
```

stem-name
: A Global Storage stem name.

The terminating period is needed for each name.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information for the stemmed array element

GLOBAL STEMS

Returns the names of all currently defined Global Storage stems.

```
►►──EXTract──GLObal──STEMs──────────────────────────────────────────►◄
```

REXX variables set

GLOBALS.0
: The number of Global Storage stems defined

GLOBALS.n
: The name of the nth Global Storage stem

GSTATUS

Returns information about the storage used to contain the Global Storage data which has been loaded with SET GLOBAL STEM commands. This includes:

- number of Global Storage stems
- Global Storage storage consumption (total, direct, pooled)
- Global Storage storage pool utilization, including the number of AREAs in the pool which are unused

```
►►──EXTract──GSTAtus────────────────────────────────────────────────►◄
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the LSM Information window command arguments

ICOUNT

Returns the number of instructions executed since the last ICOUNT command.

```
►►──EXTract──ICOunt──────────────────────────────────────────────────────────────►◄
```

REXX variables set

ICOUNT
: The number of instructions executed.

LANGUAGE ARGUMENTS | ARGS

Returns the current command arguments for each LSM Information window.

```
►►──EXTract──LANguage──┬─ARGs──────┬─────────────────────────────────────────────►◄
                       └─ARGuments─┘
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134)

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the LSM Information window commands

LANGUAGE COMMANDS | CMDS

Returns the current command for each LSM Information window.

```
►►──EXTract──LANguage──┬─COMmands─┬────────────────────────────────────────────────►◄
                       └─CMDs─────┘
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the LSM Information window commands

LANGUAGE OPTIONS

Returns information as to the current value of the various IDF Language Support settings.

```
►►──EXTract──LANguage──OPTions─────────────────────────────────────────────────────►◄
```

To obtain the current value of an individual IDF setting, the preferred method is to use the EXTRACT Query command.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the current value of the various IDF Language Support settings.

LANGUAGE STATUS

Returns information about the extract files which have been loaded with LANGUAGE LOAD commands.

```
►►──EXTract──LANguage──STAtus──┬──────────────────────┬──────────────────────────►◄
                               │  ┌─;───────────────┐ │
                               │  ▼                 │ │
                               └───extract-file-name┴─┘
```

extract-file-name
: An extract file name. Information is returned only for those compiles which were contained within the specific extract files. If no extract file names are specified, the default is to extract information for all currently loaded (with LANGUAGE LOAD) language extract files.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set
stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the specified extract files.

LANGUAGE STEM

Returns the name of the REXX stemmed variable array which is used to return information to an IDF macro as a result of all subsequent EXTRACT LANGUAGE commands, as well as a number of other EXTRACT commands.

```
►►──EXTract──LANguage──STEM────────────────────────────────────────────────────►◄
```

REXX variables set
STEM Name of the REXX stemmed variable array

LANGUAGE VERSION

Returns the IDF Language Support Version Information.

```
►►──EXTract──LANguage──VERsion─────────────────────────────────────────────────►◄
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**1**
: IDF Language Support version information text.

  This has the form:

  ```
  "ASMLANG Vn.Rn.nn (generated ccyy.ddd hh:mm)".
  ```

LASTMSG

Returns the last 10 messages that were issued by SET MSG commands, or as a result of the execution of other IDF commands.

```
►►──EXTract──LASTMsg────────────────────────────────────────────────────────────────►◄
```

All messages generated by IDF are returned with message prefixes, even if the MSGID option is OFF and the message is displayed without the message id. User messages issued with a SET MSG command do not have a message id.

All messages generated by IDF are returned, even if the MSGMODE option is OFF and the message was not actually displayed in a Command window message display line.

Example

The LASTMSGM values of 0 and 1 may be used directly in Boolean tests, such as:

```
'EXTRACT LASTMSG'
Do I = 1 to lastmsg.0
  If lastmsgm.i Then
    Say lastmsg.i
End
```

REXX variables set

LASTMSG.0
: Number of items in the stemmed array

LASTMSG.n
: The message text (with message id, if available), of the last *n* messages, with the most recent message in LASTMSG.1.

LASTMSGM.0
: Number of items in the stemmed array

LASTMSGM.n
: The value of the MSGMODE setting at the time the *n*th message was issued. This is one of:

0	MSGMODE was OFF
1	MSGMODE was ON

LOAD

Returns information about the program specified when IDF was invoked and where it is loaded in memory.

```
►►──EXTract──LOAd──────────────────────────────────────────────────────────────►◄
```

REXX variables set

NAME
The name of the program module

AREA The area in which the program is loaded:

- **USER** User program area
- **TRANS**
  CMS Transient area at location X'0E000'-X'0FFFF'
- **NUCEXT**
  CMS Nucleus Extension area

SYMBOL
The number of symbols defined

ORIGIN
The program's origin in memory, 8-digit hexadecimal

EPOFFSET
The offset within the program of its entrypoint, 8-digit hexadecimal

OFFSET
The current value of the offset (set by the OFFSET command), 8-digit hexadecimal

SIZE The size of the program in bytes, 8-digit hexadecimal

LOADLIB
The name of the loadlib specified, or blank

LOCATION

Extracts bytes of main memory.

```
►►──EXTract──LOCATIon──number-of-bytes──start-address──────────────────────────►◄
```

number-of-bytes
Number of bytes of main memory to be extracted.

start-address
An expression giving the starting address.

If the expression contains an access register then the storage that is extracted will come from the dataspace identified by the ALET in the referenced access register.

REXX variables set

MEMAREA
Contents of the specified memory area

The EXTRACT LOCATION command allows you to retrieve storage within your program's defined limits. For more details on your program's defined limits and how to change them, see (CMS) “Your program's defined limits” on page 56 and (TSO) “Your program's defined limits” on page 49.

If the TRACEALL option or the RISK option is set, you may be able to retrieve storage beyond the program's defined limits.

Example

```
EXTRACT LOCATION 8 0(R1)
```

This example sets REXX variable MEMAREA to the contents of eight bytes pointed to by the target program's R1.

```
EXTRACT LOCATION 8 0(AR2)
```

This example sets REXX variable MEMAREA to the contents of eight bytes pointed to by the target program's R2 in the dataspace identified by the ALET in AR2.

LOCATION ALET

Extracts bytes of dataspace.

```
►►──EXTract──LOCATIon──ALET──alet──number-of-bytes──start-address──────────────────────────────►◄
```

access-link-entry-token
: An expression that specifies the ALET for the dataspace.

number-of-bytes
: Number of bytes of main memory to be extracted.

start-address
: An expression giving the starting address.

REXX variables set
MEMAREA
: Contents of the specified memory area

Example

```
EXTRACT LOCATION ALET 10003 4 X'1000'
EXTRACT LOCATION ALET 10004 8 0(R1)
```

MAP

Returns information about the location of all modules and code sections known to IDF, as well as the name of the extract file which may be associated with each code section.

```
               ┌─*─────────────────┐
►►──EXTract──MAP──┼───────────────────┼──────────────────────────────────────────►◄
               │  ┌─;──────────┐   │
               └──▼─module-name─┴───┘
```

*** Extract information for all currently known modules.

`module-name`
: A module name. Information is returned for the nominated modules, and the CSECTs which are a part of those modules.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
: Number of items in the stemmed array

stemname.**n**
: Information about the module and code sections

MODE (CMS only)

Returns the current file mode.

```
►►──EXTract──MODE──────────────────────────────────────────────────────────►◄
```

The initial file mode is A1.

It may be change by specifying the MODE option at invocation, or by issuing the MODE command in the PROFILE macro.

REXX variables set

MODE
: The current file mode

MODULES

Returns information about the modules defined to IDF.

```
►►──EXTract──MODUles───────────────────────────────────────────────────────►◄
```

REXX variables set

MODULES.0
: Number of modules defined to IDF

MODULES.n
: Name origin size symbols, where:

  NAME
  : The name of module.

  ORIGIN
  : The module's origin in memory, eight digit hexadecimal.

  SIZE
  : The size of the module origin in bytes, eight digit hexadecimal.

SYMBOLS
The number of symbols defined to IDF that are associated with this module.

MSTATUS

Returns information about the storage used to contain the extract data information which has been loaded with LANGUAGE LOAD commands. This includes:

- number of compile areas
- extract data storage consumption (total, direct, pooled)
- extract data storage pool utilization, including the number of AREAs in the pool which are unused

```
►►──EXTract──MStatus──────────────────────────────────────────────────────►◄
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Information about the extract data storage

NAMES

Returns information about symbol names.

```
►►──EXTract──NAMes──┬─────────────────────────┬───────────────────────────►◄
                    │ ┌─;───────────────────┐ │
                    │ ▼                     │ │
                    └─symbol-name-pattern───┴─┘
```

symbol-name-pattern
A pattern to match against all the symbol names.

See “NAMES” on page 154 for details about pattern meta-characters. If omitted, all symbol names are returned.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Information about the symbol names

OPTIONS

Returns a list of all IDF options and their current settings.

```
►►──EXTract──OPTions───────────────────────────────────────────────────────────►◄
```

REXX variables set

OPTION
List of options and their current settings

The contents of variable OPTION have the following format:

```
CMDLOG=0 COMMAND=0 ... and so on
```

A value of 1 indicates that the option was last set ON; a value of 0 indicates that the option is OFF. For a list of the options which are returned, see “SET OPTION” on page 178. For information about the meaning of the options, see Chapter 4, “Invoking IDF to debug your program,” on page 21.

The string "AMODE31=0" or "AMODE31=1" is included in the data returned by EXTRACT OPTIONS. This provides a means for macros to determine the current addressing mode.

PER (CMS only)

Returns the value of the PER setting.

```
►►──EXTract──PER───────────────────────────────────────────────────────────────►◄
```

REXX variables set

PER Current PER setting:
- Y if active
- N if inactive
- D if unavailable (TSO, CMS/SP without SET ECMODE ON).

PFK

Extracts all current PFK definitions.

```
►►──EXTract──PFK───────────────────────────────────────────────────────────────►◄
```

REXX variables set

PFK.0 - PFK.24
Current PFK settings

Where:
- Variable PFK.0 contains the assignment of the **ENTER** key.
- Variables PFK.1 through PFK.24 contain the assignments of **PF1** through **PF24**.

If a PF key is undefined, its setting is returned as an asterisk (*).

PLIST

Obtains the arguments provided at IDF invocation.

```
►►──EXTract──PLIST──────────────────────────────────────────────────────────►◄
```

REXX variables set
PLIST Arguments passed to IDF at invocation

PLOCATES

Returns information about variables located though a locator (pointer) variable.

```
                      ┌─;─────────────────────┐
►►──EXTract──PLOcates───▼─locator-variable-name─┴────────────────────────────►◄
```

locator-variable-name
A locator variable names.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set
stemname.**0**
Number of items in the stemmed array
stemname.**n**
Information about the located variables

QUALIFY

Extracts the name of the module that is used with some commands and addresses when no explicit module name is specified.

```
►►──EXTract──QUAlify──────────────────────────────────────────────────────────►◄
```

REXX variables set

QUALIFY
: The currently qualified module.

QUERY SETTING

Returns the current value of the indicated indicator or option item.

```
►►──EXTract──Query──argument──────────────────────────────────────────────────►◄
```

argument
: The name of the indicator or option item. Information is available for:

AMODE	COMMENTS	IMPMACRO	OLDBREAK	SHOW
APROGMSG	COMPACT	INVPSW	OPTIMIZE	STEM
ASCII	DSECTS	LIBE	PACKED	STOPNOP
AUDIT	DCL	MACROS	PADID	STOPSTMT
AUTH	DEBUG	MACROLOG	PASSPGM	SUBSTRING
AUTOLOAD	DECLARE	MAJOR	PATH	SVC97
AUTOSIZE	DETAIL	MODE	PATHFILE	SWAP
BIT	DMS0	MODMAP	QWDUMP	SYSTEM
BCX	ENUM	MSGID	RISK	TRACEALL
BOUNDS	EXLIMIT	MSGMODE	RLOG	TRANS
BRIEF	FASTPATH	NEGATIVE	ROWSTYLE	UNFTDUMP
CHAR	FIXED	NEST	SALIMIT	VSEP
CKSUBCM	FLOAT	NOCODE	SAREGS	XPATH
CMDLOG	FULLQUAL	NUCEXT	SCDACTIV	ZONED
CMPEXIT	HEXDISP	OFFSET	SELFNUCX	1ADSTOP
	HEXINPUT			

This command eliminates the need to parse release-dependent information returned by EXTRACT OPTIONS or EXTRACT LANGUAGE OPTIONS.

REXX variables set

QUERY.0
: Number of items in the stemmed array

QUERY.n
: Information about the indicator or option item current value.

REGS

Extracts the GPRs, FPRs, and PSW in EBCDIC HEX representation.

```
►►──EXTract──REGs──────────────────────────────────────────────────────────►◄
```

REXX variables set

GPR.0 - GPR.15
: Current values of GPRs

OGPR.0 - OGPR.15
: Previous values of GPRs

FPR.0 - FPR.15
: Current values of FPRs, if available

OFPR.0 - OFPR.15
: Previous values of FPRs, if available

PSW Current value of PSW

OPSW
: Previous value of PSW

FPC Floating point control register, if available

If the option AMODE64 is on, the PSW is returned as a 32 byte field, and GPR.0 - GPR.15 are returned as 16 byte fields.

REGSTOPS (CMS only)

Returns a list of registers that have been selected for monitoring.

```
►►──EXTract──REGSTops──────────────────────────────────────────────────────►◄
```

REXX variables set

GPR.0 - GPR.15
: Each variable is set to either:

Y	To indicate that the register is being monitored
N	To indicate that it is not being monitored.

SCOPE

Returns information about a statement scope block.

```
►►──EXTract──SCOpe──┬─────────┬────────────────────────────────────────────►◄
                    └─address─┘
```

address
: An IDF address expression. Defines the statement scope block. If omitted, uses the current execution location as determined from the PSW.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
    Number of items in the stemmed array

stemname.**1**
    Scope block start statement number

stemname.**2**
    Scope block end statement number

stemname.**3**
    Scope block owner symbol name (if any)

SCRVAR

Returns the contents of the LSM Information window.

```
►►──EXTract──SCRvar──────────────────────────────────────────────────────►◄
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
    Number of items in the stemmed array

stemname.**n**
    Information about the LSM Information window contents.

SELFNUCX

Returns the current value of the self-load offset.

```
►►──EXTract──SELFNucx────────────────────────────────────────────────────►◄
```

The self-load offset is set by the SELFNUCX symbol-name option at IDF invocation, or by issuing the SET SELFNUCX command.

REXX variables set

SELFNUCX
    Contents are as follows:
- If the program is *not* self-loading, variable SELFNUCX is blank.

- If the self-load offset was specified as a symbol name, the contents of variable SELFNUCX consist of two tokens. The first token is the symbol, and the second is an 8-digit hexadecimal offset.
- If the self-load offset was specified as an absolute value, the contents of variable SELFNUCX consist of two tokens. The first token is the string "<n/a>", and the second token is an 8-digit hexadecimal offset.

SKIPSTEP

Returns currently skipped subroutines.

```
►►──EXTract──SKIPstep──┬─────────┬──────────────────────────────────────────►◄
                       └─address─┘
```

address
An IDF expression. Address of subroutine to return. If omitted, all skipped subroutines are returned.

REXX variables set

SKIP.0 Number of subroutines skipped

SKIP.n
The next skipped subroutine

SOURCE

Returns source records.

```
►►──EXTract──SOUrce──┬─────────┬────────────────────────────────────────────►◄
                     └─address─┘
```

address
The address. May be specified in the form of an IDF address expression. If omitted, uses the current execution location as determined from the PSW.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
The source records for the specified memory address.

STOREMAP

Returns Storage Allocation Map information.

```
►►──EXTract──STORemap──┬──────────────────┬──────────────────────────────────────────►◄
                       │   ┌─;──────────┐ │
                       │   ▼            │ │
                       └───addr-expr1───┴─┘
```

addr-expr1
A storage address expression. If omitted, returns an overall Storage Allocation Map.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set
stemname.**0**
Number of items in the stemmed array
stemname.**n**
Storage Allocation Map information

STRUCTURE

Returns information about structure or union components.

```
                         ┌─;──────────────┐
                         ▼                │
►►──EXTract──STRucture───component-name───┴──────────────────────────────────────────►◄
```

component-name
A structure or union component name.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set
stemname.**0**
Number of items in the stemmed array
stemname.**n**
Information about the structure or union components

SVC (CMS only)

Returns the current SVC tracing state.

```
►►──EXTract──SVC──────────────────────────────────────────────────────────────────────►◄
```

REXX variables set

SVC Current SVC tracing state:

- **Y** If SVC tracing is in effect
- **N** If SVC tracing is not in effect.

SYMBOLS

Returns information about known symbols.

```
►►──EXTract──SYMbols──┬─────────────┬─────────────────────────────────────────────────►◄
                      └─module-name─┘
```

module-name
A module name. Specifies which module's symbols should be returned. If omitted, returns the symbols for the currently qualified module.

REXX variables set

SYMBOL.0
The number of symbols known to IDF.

Information about individual symbols is returned in SYMBOL.1 to SYMBOL.*nnn*

SYMBOL.n
Information about one symbol known to IDF.

Each REXX variable containing information about a symbol known to IDF will contain the following information:

```
(<module.>csect  ) symbol reloc addr totlen intext full type
```

The fields returned have these meanings:

module
The name of the module within which the symbol occurs
- Present if the FULLQUAL option is ON
- Is followed by a period

csect The name of the code section within which the symbol occurs.

symbol
The name of the actual symbol.

reloc The offset of the symbol within the specified CSECT (hex).

addr The offset of the symbol within the target module (hex).

totlen The total length associated with the symbol (hex).

intext One character, either "I" or "E":
- **I** The symbol is an internal symbol.
- **E** The symbol is externally known.

full One character, either "F" or "U":

F	The symbol is fully defined.
U	The symbol is not fully defined; this may occur if no information is available to define the start of the CSECT within which the symbol occurs.

type This is a 2-digit hex value which describes the type of symbol, and is one of:

00	Space
01	CSECT
02	DSECT
03	COMMON
04	Machine Instruction
05	CCW
06	EQU, LTORG, CNOP, ORG
10	C-con
14	X-con
18	B-con
20	F-con
24	H-con
28	E-con
2C	D-con
30	A/Q-con
34	Y-con
38	S-con
3C	V-con
40	P-con
44	Z-con
48	L-con
FE	Self-defining, `addr` is actual value
FF	Unknown, no symbol type available

TASKS

Returns information about the currently executing tasks.

```
►►──EXTract──TASks────────────────────────────────────────────────────────────────►◄
```

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Task information

TYPE

Returns information about type attributes for variables.

```
                         ┌─;──────────┐
                         ▼            │
►►──EXTract──TYPe───variable-name─┴──────────────────────────────────────────►◄
```

variable-name
  A variable names.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
  Number of items in the stemmed array

stemname.**n**
  Type information for the specified variables

UNION

A synonym for the command STRUCTURE. For details, see "STRUCTURE" on page 247.

VALUE

Returns the value of an expression.

```
►►──EXTract──VALue──address──────────────────────────────────────────────────►◄
```

address
  An IDF address expression.

REXX variables set

EXPR The value of the specified expression

If the address is outside of the programs defined to IDF, the code section (CSECT) name is omitted and the symbolic name is a second hexadecimal value.

If the address is outside the currently qualified module or the FULLQUAL option is ON, then the code section name as defined above is *module.csect*, where *module* is the name of module containing the address.

Numbers in expressions can be specified in explicit (X'123', F'123') or implicit (123) notation. Numbers that do not explicitly specify the base are evaluated according to the current setting of the HEXINPUT option. When writing macros it is recommended that you use explicit base notation.

Example

```
EXTRACT VALUE ALLOPEN+4(R2)
```

If symbol ALLOPEN is at X'20000' and the target program's R2 contained 8, the value returned is X'2000C'. This value is returned in REXX variable EXPR as several tokens, and can be parsed with the following instruction:

```
Parse Var EXPR hexvalue . '(' csect ')' symbolic
```

After this Parse instruction, the following variables are set:

EXPR 0002000C (ALLOPEN) ALLOPEN+12

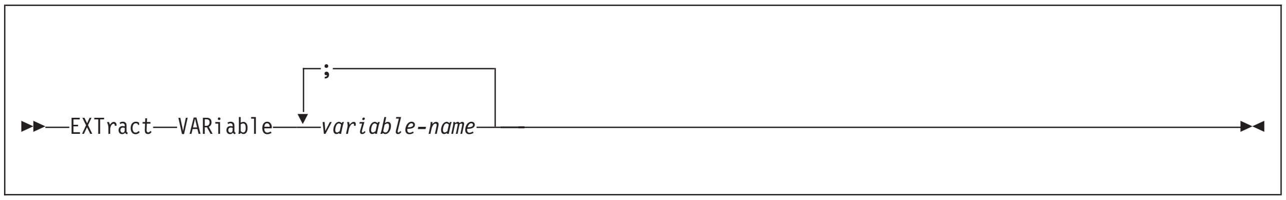
HEXVALUE
0002000C

CSECT
ALLOPEN

SYMBOLIC
ALLOPEN+12

VARIABLE

Returns information about variables.



variable-name
A variable names.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

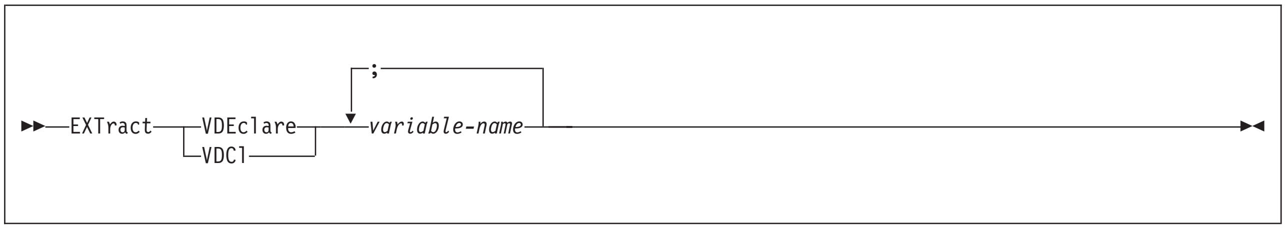
REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Information about the variables

VDECLARE | VDCL

Returns attribute information about variables.



variable-name
A variable name.

This command provides information which is independent of display format settings, except for SPACE.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
Number of items in the stemmed array

stemname.**n**
Attribute information for the specified variables

VERSION

Returns the IDF version information.

```
►►──EXTract──VERsion──────────────────────────────────────────────────────►◄
```

REXX variables set

VERSION
IDF version information text.

For more information, see “LANGUAGE VERSION” on page 235.

VLOC

Returns location information about the indicated variables.

```
                   ┌─;─────────────┐
►►──EXTract──VLOc──▼─variable-name─┴──────────────────────────────────────►◄
```

variable-name
A variable names.

This command provides information which is independent of display format settings, except for SPACE.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see “LANGUAGE STEM” on page 134).

The EXLIMIT command (see “EXLIMIT” on page 116) is used to control the maximum stemmed array index value.

REXX variables set

stemname.**0**
  Number of items in the stemmed array
stemname.**n**
  Location information for the specified variables

VVALUE

Returns data value information about variables.

```
                          ┌─;─────────────┐
                          ▼               │
►►──EXTract──VVAlues────variable-name─┴───────────────────────────────────────────────►◄
```

`variable-name`
  A variable name.

This command provides information which is independent of the COMPACT and BRIEF settings.

The name of the stemmed variable array defaults to LSM. and may be set by the LANGUAGE STEM command (see "LANGUAGE STEM" on page 134).

The EXLIMIT command (see "EXLIMIT" on page 116) is used to control the maximum stemmed array index value.

REXX variables set
stemname.**0**
  Number of items in the stemmed array
stemname.**n**
  Data value information for the specified variables

WINDOWS

Returns information about the screen and open windows.

```
►►──EXTract──WINdows──────────────────────────────────────────────────────────────────►◄
```

REXX variables set
WINDOW.0
  n rows cols
  - Where:
    n The number of windows open on the screen.
    rows The number of rows on the screen that can be filled with windows.
    cols The number of columns on the screen.
  - Information about individual windows is returned in variables of the form WINDOW.n.

WINDOW.n
  type orow ocol rows cols start:next
  - where:
    type The type of the open window. It is one of:

	ADSTOP	The AdStops window.
	AFPR	The Additional Floating-Point Registers window.
	BREAK	The Break window.
	DISASM	A Disassembly window.
	DUMP	A Dump window.
	LSMINFO	An LSM Information window.
	MINIMIZE	Minimized Windows viewer.
	OPTIONS	The Options window.
	OREGS	The Old Registers window.
	REGS	The Current Registers window.
	SKIP	Skipped Subroutines window.
	STAT	The Target Status window.

orow The row on which the upper left corner of the window is located.

ocol The column on which the upper left corner of the window is located.

rows The number of rows in the window.

cols The number of columns in the window.

start The starting address for storage displayed in the window if it is a Disassembly window or a Dump window.

next The address of storage to be displayed in the window if it is a Disassembly window or a Dump window and it is scrolled. It is separated from the start address by a colon (:).

Part 4. Appendixes

Appendix A. ASMLANGX options

This section lists all of the ASMLANGX options, and provides information about each option.

ASM

```
►►──ASM──────────────────────────────────────────────────────────────►◄
```

This option indicates that ASM is the language being extracted. Since ASM is the default (and only) language extracted, you need never use this option.

CONDASM | NOCONDASM

```
     ┌─NOCONDASM─┐
►►───┤           ├─────────────────────────────────────────────────►◄
     └─CONDASM───┘
```

Use CONDASM to include conditional assembly statements and `.*` comments during extraction, NOCONDASM to suppress them.

DCL | NODCL

```
     ┌─DCL───┐
►►───┤       ├─────────────────────────────────────────────────────►◄
     └─NODCL─┘
```

Use DCL to extract source code for declarations, including associated block comments, NODCL to not extract them. (Variable information is extracted regardless.)

DEBUG

```
►►──DEBUG────────────────────────────────────────────────────────────►◄
```

DEBUG is *not* intended for general use. It adds messages to the processing log file. These messages hold internal diagnostic information for IDF Service problem analysis.

ERROR

```
►►──ERROR──────────────────────────────────────────────────────────────────►◄
```

Use ERROR to list variables for which information is incomplete.

IFM (CMS only)

```
►►──IFM──fm────────────────────────────────────────────────────────────────►◄
```

Default: IFM *

fm is the file mode to search initially for all input files, for a non-standard search path. If not found, revert to the standard search path.

INCL | NOINCL

```
     ┌─INCL───┐
►►───┤        ├────────────────────────────────────────────────────────────►◄
     └─NOINCL─┘
```

Use INCL to extract source code for includes, NOINCL to not extract it. (Variable information from included files is extracted regardless.)

LOUD | QUIET

```
     ┌─QUIET─┐
►►───┤       ├─────────────────────────────────────────────────────────────►◄
     └─LOUD──┘
```

Use LOUD to issue messages to the terminal containing:
- the version and release of ASMLANGX
- the output file identifier

- the input file identifier
- processing progress information

Terminal messages are modified by the execution environment and OS runtime settings:
- On TSO, the messages respect the TSO MSGID setting:

  MSGID
  : Message identifier and text are displayed

  NOMSGID
  : Message text is displayed
- On z/OS Batch, the messages are written to the JES message log with `WTO ROUTCDE=11`. The message identifier and text are shown.
- On CMS, the messages respect the CMS EMSG setting:

  ON Message identifier and text are displayed

  TEXT Message text is displayed

  OFF Message is suppressed
- On z/VSE Batch, the messages are written to the console log. The message identifier and text are shown.

A header is added to the output file containing:
- the version and release of ASMLANGX
- the input file identifier

Use QUIET to suppress the display of **I** (Informational), **W** (Warning) and **E** (Error) messages. QUIETLY is an alias for QUIET.

MACDEF | NOMACDEF

```
    ┌─NOMACDEF─┐
►►──┴─MACDEF───┴──────────────────────────────────────────────►◄
```

Use MACDEF to include inline macro definitions during extraction, NOMACDEF to suppress them.

OFM (CMS only)

```
►►──OFM──fm──────────────────────────────────────────────────►◄
```

Default: OFM A1

fm specifies the file mode (FM) of the output file, if other than the default `A1` is desired.

OFN

```
►►──OFN──fn──────────────────────────────────────────────────────────►◄
```

Default: OFN `output-file-name`

fn specifies:
- On z/OS, the PDS member name of the output file.
- On CMS, the file name (FN) of the output file.
- On z/VSE, the librarian member name of the output member.

OFT

```
►►──OFT──ft──────────────────────────────────────────────────────────►◄
```

Default: OFT ASMLANGX

ft specifies:
- On z/OS, the DD name of the output file.
- On CMS, the file type (FT) of the output file.
- On z/VSE, not used.

PACK | NOPACK

```
     ┌─PACK───┐
►►───┼────────┼──────────────────────────────────────────────────────►◄
     └─NOPACK─┘
```

Use PACK to compact redundant characters in source statement text, NOPACK to leave it as is.

PFM (CMS only)

```
►►──PFM──fm──────────────────────────────────────────────────────────►◄
```

Default: PFM *

fm specifies the file mode (FM) to initially search for the input file, if other than the standard search path is desired. If not found, revert to the standard search path. This option overrides the IFM option.

PFT

```
►►──PFT──ft─────────────────────────────────────────────────────────────►◄
```

Default: PFT SYSADATA for TSO and CMS, PFT SYSADAT for z/VSE

ft specifies:
- On CMS, the file type (FT) of the input file.
- On z/OS, the DD name of the input file.
- On z/VSE, the DLBL name of the input file.

SEQ | NOSEQ

```
     ┌─NOSEQ─┐
►►───┴─SEQ───┴──────────────────────────────────────────────────────────►◄
```

Use SEQ to retain source sequence numbers in columns 73 to 80, NOSEQ to not write them to the extract file. If the PACK option is selected, this significantly reduces the size of a typical extract file.

Appendix B. Diagnostic messages

Message numbers and severity levels

Each of the messages issued by IDF is of the form:

Message format

ASMmssnnnnl text

where:

ASM Is the component name for IDF and related utilities.

m Is the program identifier.

This is one of:

K	ASMLKEDT (z/VSE Phase MAP Creation Utility)
L	IDF Language Support
M	IDF Base Debugger
X	ASMLANGX (ADATA Extraction Utility)

ss Is the subsystem identifier.

This identifies the subsystem within the program which issued the message. Its main use is to help IBM Service Personnel in tracking potential problems.

nnnn Is the message number.

l Is the message severity level.

This is one of:

I	Informational message
A	Action message Used to prompt for user input
W	Warning message
E	Error message
S	Severe error message
T	Terminating error message

ASMLKEDT messages (z/VSE only)

ASMKLK000W Invalid LNKEDT parameter: *txt1*

Explanation: The parameter specified is not a valid LNKEDT parameter.

User response: Correct the parameter to a valid LNKEDT value.

ASMKLK001S No MAP produced

Explanation: A MAP file has not been produced.

User response: This message should be accompanied by another error message to explain why this may have occurred.

ASMKLK002S No Librarian DCB generated

Explanation: The librarian environment could not be setup to create a MAP.

User response: This is an internal problem. Look for any accompanying messages and if the problem cannot be determined contact your IBM Support Center.

ASMKLK003S OPEN member name: *phasename* **failed RC=***retcode*

Explanation: A LIBRM OPEN request has failed for the member specified with the return code specified.

User response: See return codes for the LIBRM OPEN macro in the *z/VSE: System Macros Reference*.

ASMKLK004S PUT member name: *phasename* **failed RC=***retcode*

Explanation: A LIBRM PUT request has failed for the member specified with the return code specified.

User response: See return codes for the LIBRM PUT macro in the *z/VSE: System Macros Reference*.

ASMKLK005I Member name: *phasename***.MAP cataloged into sublibrary:** *sublib***.***name*

Explanation: Informational message describing the member name and the sublibrary where the member has been cataloged.

User response: None

ASMKLK006S CLOSE member name: *phasename* **failed RC=***retcode*

Explanation: A LIBRM CLOSE request has failed for the member specified with the return code specified.

User response: See return codes for the LIBRM CLOSE macro in the *z/VSE: System Macros Reference*.

ASMKLK007S LIBRM STATE for CATALOG library failed RC=*retcode*

Explanation: An error has occurred in locating the phase catalog library.

User response: Ensure a `// LIBDEF PHASE,CATALOG=` statement has been coded. If the statement has been provided then check the return codes for the LIBRM STATE macro in the *z/VSE: System Macros Reference*.

IDF Language Support messages

ASMLVR001E Keyword missing

Explanation: One or more extra keywords were expected, but were not present.

User response: Change the command specification to add the appropriate keywords.

ASMLVR002E Keyword too long, "*txt1***"**

Explanation: *txt1* is not recognized as a valid keyword. It exceeds the maximum length of a valid keyword, and may be spelled incorrectly.

User response: Change the command specification to use the appropriate keyword.

ASMLVR003E Keyword not recognized, "*txt1***"**

Explanation: *txt1* is not recognized as a valid keyword.

User response: Change the command specification to use the appropriate keyword.

ASMLVR004E Keyword not valid in this context, "*txt1***"**

Explanation: *txt1* is recognized as a valid keyword, but is not valid in the context in which it has been specified.

For example, the command SHOW BOTH is valid, but the command SHOW SHOW will produce this message.

User response: Change the command specification to use the appropriate keyword.

ASMLVR005E Additional parameters expected

Explanation: One or more extra parameters were expected, but were not present.

User response: Change the command specification to add the appropriate parameters.

ASMLVR006E Extraneous keyword, "*txt1*"

Explanation: The keyword *txt1* was specified, but no keyword was expected in this context.

User response: Correct the command specification to eliminate any extraneous keywords.

ASMLVR010I *txt1*

ASMLVR011I One of: *txt1*

Explanation: The "?" keyword was specified in the command, to request command prompting. The list of valid keywords in this context are shown in this message.

User response: Change the command specification to use the appropriate keyword in place of the "?" keyword.

ASMLVR012I Debug mode is now *txt1*

Explanation: A LANGUAGE DEBUG command has been successfully executed. The resulting debug options are shown in this message.

User response: None

ASMLVR013I *txt1* **subsystem not available in this version of** *txt2*

ASMLVR020E Unknown color, "*txt1*"

Explanation: The indicated keyword *txt1* is not recognized as a valid LANGUAGE COLOR keyword.

User response: Change the command specification to use the appropriate keyword.

ASMLVR030E No Extract data is currently loaded

Explanation: A command has been executed which required that ASMLANGX extract data be previously loaded with LANGUAGE LOAD. None was present, so the command did not complete successfully.

User response: Load the appropriate ASMLANGX extract data files.

ASMLVR031E Extract file *filename* **not LOADed**

Explanation: An attempt was made to reference ASMLANGX extract data file *filename*. This file has not yet been loaded with LANGUAGE LOAD.

User response: Load the ASMLANGX extract data file, if required.

ASMLVR032E Module *txt1* **not known to** *txt2*

Explanation: An attempt was made to reference module *txt1*, which has not been defined to IDF.

User response: Define the module to IDF using the MODULE command.

ASMLVR033E The PSW indicates an address which is not within any known statement.

ASMLVR034E REXX variable not found: *var1*

Explanation: A command has attempted to access the specified REXX variable, but it was not present.

User response: None

ASMLVR035E Unable to update REXX variable: *var1*

Explanation: A command has attempted to update the specified REXX variable, but was unsuccessful.

User response: Ensure sufficient virtual storage is present

ASMLVR036E Stem ".0" value not positive decimal number: *txt1*

Explanation: A command has attempted to access the specified REXX stemmed array variable, but the value of the "stemname.0" control variable was not a positive decimal number.

User response: None

ASMLVR037E REXX variable data length *dec1* **exceeds limit of** *dec2* **bytes:** *var3*

Explanation: A command has attempted to access the specified REXX variable, but the contents were too large to fit in the data buffer.

User response: None

ASMLVR040I Variable separator blank line will be *txt1*

ASMLVR041I Variable location audit will be *txt1*

ASMLVR042I *txt1* **variable declaration information will be** *txt2*

ASMLVR043I Variable information will be displayed in *txt1* **format**

ASMLVR050I Structure major component data will be *txt1*

ASMLVR051I LSM window detail level is now "*txt1*, *txt2*"

ASMLVR052E Invalid LSM window detail level, "*txt1*"

ASMLVR053I Structure pad variable names will be *txt1*

ASMLVR060E No Storage Allocation Map information is available

Explanation: A STOREMAP command was executed, but no Storage Allocation Map information was available for display.

User response: None

ASMLVR061E Storage Allocation Map error: *txt1*

Explanation: The specified error occurred while building the Storage Allocation Map information display. STOREMAP processing has terminated.

User response: None

ASMLVR065E No Task information is available

Explanation: A TASKS command was executed, but no information about tasks was available for display.

User response: None

ASMLVR066E Task information error: *txt1*

Explanation: The specified error occurred while building the Task information display. TASKS processing has terminated.

User response: None

ASMLVR070E No Global Storage stems are currently defined

Explanation: A command has been executed which required that Global Storage stems be defined previously by **SET GLOBAL** commands. None was present, so the command did not complete successfully.

User response: None

ASMLVR072W Global Storage stem not found: *txt1*

Explanation: A command has attempted to access the specified Global Storage stem, but it was not defined.

User response: Create the Global Storage stem with the **SET GLOBAL** command.

ASMLVR080I *txt1* checking is now *txt2*

ASMLVR090I LSM Status/Settings *txt1*

ASMLVR100E No variable names match pattern: *txt1*

ASMLVR110E No LongName defined for *txt1* in Module "*txt2*"

ASMLVR120I Current *txt1* display format is "*txt2*"

ASMLVR121E Unknown *txt1* display format, "*txt2*"

ASMLVR122E Display format "*txt1*" not valid for *txt2*

ASMLVR123E Multiple variables not valid *txt1*

ASMLVR130I Source at *txt1* *txt2* shown

ASMLVR131W Unable to perform *txt1* command

ASMLVR132E Function not valid if source is suppressed

ASMLVR140I Target string displayed at top of DISASM area

ASMLVR141W Target not found

ASMLVR142E Search not possible, since no source information is available

ASMLVR143E Search parameter not recognized: *txt1*

ASMLVR144E Search string missing or all-blank

ASMLVR145E Unexpected characters follow ending delimiter

ASMLVR146I Unused extract data memory blocks *txt1*

ASMLVR147I Unused Global Storage memory blocks *txt1*

ASMLVR148W Autowrap performed to display target string.

ASMLVR150I Display now limited to source code only

ASMLVR151I Display of source code disabled

ASMLVR152I Interspersed source/disassembly now displayed

ASMLVR153I ***txt1*** **will be** ***txt2***

ASMLVR154I ***txt1*** **generating code will be shown**

ASMLVR155I Display reset to show all program information

ASMLVR160I LSM window will scroll by ***txt1 txt2***

ASMLVR161I LSM window scrolling disabled

ASMLVR162E Invalid LSM window scroll amount, "*txt1*"

ASMLVR170I LSM REXX stem variable name is now "*txt1*"

ASMLVR171I LSM REXX stem limit is now "*dec1*"

ASMLVR172E LSM REXX stem limit ***dec1*** **is outside of range** ***dec2*** **to** ***dec3***

ASMLVR173E LSM REXX stem index limit reached during EXTRACT

ASMLVR180I Maximum Caller level is now ***dec1***

ASMLVR181I CALLERS Save Area registers ***txt1***

ASMLVR182E Caller level ***dec1*** **is outside of range** ***dec2*** **to** ***dec3***

ASMLVR183E Unable to access Caller level ***txt1***

ASMLVR190I XPATh reset to default value of "*txt1*"

ASMLVR191I New XPATh accepted. Review using "LAN OPTIONS"

ASMLVR192E Invalid character "*txt1*" in XPATH entry ***txt2*****. XPATH changes ignored**

ASMLVR193E Maximum number of XPATH entries (*txt1*) exceeded. XPATH changes ignored

ASMLVR200E Var is not ***txt1*****:** ***txt2***

ASMLVR201E Invalid expression syntax: ***txt1***

ASMLVR202E Invalid ***txt2*** **in expression:** ***txt1***

ASMLVR203E Missing ***txt2*** **in expression:** ***txt1***

ASMLVR204E ***txt2*** **invalid as operand:** ***txt1***

ASMLVR205E ***txt2*** **occurred while** ***txt3*** **in expression:** ***txt1***

ASMLVR206E Expression ***txt2*** **not yet supported:** ***txt1***

ASMLVR210E Unknown characters within expression: ***txt1***

ASMLVR211E Extract data is incomplete for: ***txt1***

ASMLVR212E Structure declaration invalid: ***txt1***

ASMLVR213E Array declaration invalid: ***txt1***

ASMLVR214E Internal error processing expression: ***txt1***

ASMLVR215E Insufficient free storage to process expression

ASMLVR220E Error: Var storage at X'*txt1*' not accessible: ***txt2***

ASMLVR221E Automatic storage (DSA) base is R0 - invalid: ***txt1***

ASMLVR222W Warning: Value of ***txt1*** **is** ***txt2*****:** ***txt3***

ASMLVR223E Var is a constant but value not extracted: ***txt1***

ASMLVR224W Warning: Bdy(*txt1*) expected for *txt2*: *txt3*

ASMLVR225W Warning: Optimized var *txt1txt2*

ASMLVR226E Error: Optimized var *txt1txt2*

ASMLVR227E Error: Register *txt1* is invalid: *txt2*

ASMLVR230E No variable name supplied

ASMLVR231E Variable name exceeds *dec1* chars, or excess data follows variable name

ASMLVR232E Variable unknown in current compile data: *txt1*

ASMLVR233E Missing variable name in dot-qualified expression: *txt1*

ASMLVR234E Ambiguous in current scope; dot-qualify: *txt1*

ASMLVR235E Not component of indicated structure: *txt1*

ASMLVR236E Variable unknown in current block: *txt1*

ASMLVR240E Locating expression required, var is BASED(*): *txt1*

ASMLVR241E Var neither based nor part of based structure: *txt1*

ASMLVR242E Non-pointer specified in locator expression: *txt1*

ASMLVR243E Pointer name missing from locating expression

ASMLVR244E Excess locating expressions, maximum allowed is *dec1*

ASMLVR250E String is not multi-dimensioned array: *txt1*

ASMLVR251W Subscript *txt1* for array dimension *txt2* assumed: *txt3*

ASMLVR252W Extraneous array subscript *txt1* ignored: *txt2*

ASMLVR253E Subscripted variable not array or string: *txt1*

ASMLVR254E Substring valid only for string variables: *txt1*

ASMLVR255E *txt1 dec2* out of range *dec3* to *dec4* for: *txt5*

ASMLVR256E Wrap-around substring *dec1*:*dec2* not allowed: *txt3*

ASMLVR257E *txt1 dec2* not positive: *txt3*

ASMLVR258E *txt1 dec2* is below lower bound of *dec3* for: *txt4*

ASMLVR270E Function argument cannot be evaluated: *txt1*

ASMLVR271E Maximum of *dec2* function argument*txt3* exceeded: *txt1*

ASMLVR272E Function argument was omitted: *txt1*

ASMLVR273E Unable to resolve to storage address: *txt1*

ASMLVR274E Recursive *txt1* ADDR() is not allowed: *txt2*

ASMLVR300E Read/Only storage at X'*txt1*' - changes ignored, display line *dec2*

ASMLVR310E All-blank data entered on display line *dec1*

ASMLVR311E Value entered exceeds maximum allowed by declared precision, display line *dec1*

ASMLVR312E UnSigned value *txt1* not be negative, display line *dec2*

ASMLVR313E Length entered exceeds declared string length, display line *dec1*

ASMLVR320E Non-decimal character entered on display line *dec1*, col *dec2*: *txt3*

ASMLVR321E Invalid Float syntax detected on display line *dec1*, col *dec2*: *txt3*

ASMLVR322E Non-hexadecimal character entered on display line *dec1*, col *dec2*: *txt3*

ASMLVR323E Non-binary character entered on display line *dec1*, col *dec2*: *txt3*

ASMLVR324E Extraneous bits in last hex character ignored, display line *dec1*

ASMLVR325E Non-displayable data entered on display line *dec1*, col *dec2*: *txt3*

ASMLVR326E Non-ASCII character entered on display line *dec1*, col *dec2*: *txt3*

ASMLVR400E Extra right parenthesis detected

ASMLVR410E Extract file not found: *txt1*

ASMLVR411E Extract file DD not allocated: *txt1*

ASMLVR412E Extract file already LANg LOADed: *txt1*

ASMLVR413E LongName extract data *txt1* loaded for Module "*txt2*"

ASMLVR414E Insufficient free storage to LOAD: *txt1*

ASMLVR420E Extract file version (*txt1*) higher than LSM supports (*txt2*)

ASMLVR421S Extract file load not successful (RC=*dec1*)

ASMLVR422E Extract file format invalid: *txt1*

ASMLVR423E Extract file contains invalid records (RC=*dec1*)

ASMLVR424E Extract file contains NO supported records

ASMLVR425E Unable to locate *txt1* *txt2* referenced in extract file

ASMLVR426W Warning: CSECT *txt1* length is X'*txt2* bytes, X'*txt3*' expected

IDF base debugger messages

ASMMAI001W Operation exception (program check code X'01')

ASMMAI002W Privileged operation exception (program check code X'02')

ASMMAI003W Execute exception (program check code X'03')

ASMMAI004W Protection exception (program check code X'04')

ASMMAI005W Addressing exception (program check code X'05')

ASMMAI006W Specification exception (program check code X'06')

ASMMAI007W Data exception (program check code X'07')

ASMMAI008W Fixed-point overflow exception (program check code X'08')

ASMMAI009W Fixed-point divide exception (program check code X'09')

ASMMAI010W Decimal overflow exception (program check code X'0A')

ASMMAI011W Decimal divide exception (program check code X'0B')

ASMMAI012W Exponent overflow exception (program check code X'0C')

ASMMAI013W Exponent underflow exception (program check code X'0D')

ASMMAI014W Significance exception (program check code X'0E')

ASMMAI015W Floating-point divide exception (program check code X'0F')

Explanation: IDF received control as a result of the exception which occurred during user program execution.

User response: Determine the cause of the exception, and perform any required corrective action.

The PSW and register values when the exception occurred are available from the Current Registers window.

ASMMAI016W Target program ABENDed; *code*

Explanation: IDF received control as a result of an ABEND which occurred during user program execution.

User response: Determine the cause of the ABEND, and perform any required corrective action.

More diagnostic information is supplied if available:
- For system ABENDs, the system code value is shown in hexadecimal. If the reason code value is available, it is shown in hexadecimal.
- For user ABENDs, the user code value is shown in decimal. If the reason code value is available, it is shown in decimal.
- For other ABENDs, the ABEND code is shown in decimal.

The PSW and register values when the ABEND occurred are available from the Current Registers window.

ASMMAI017I SVC instruction trapped

Explanation: SVC instruction trapping has been enabled on CMS using the SET SVC Y command, or by changing the SVC setting in the Breakpoint window to "Y".

Subsequently, an SVC instruction has been executed in a user program, and IDF has received control.

User response: None

ASMMAI018I BREAK point has been reached but NOT executed

Explanation: An instruction is about to be executed for which an IDF breakpoint exists. IDF has received control.

User response: None

ASMMAI019I Target program has completed and returned control

Explanation: The initial target program has completed execution. IDF has received control.

User response: None

ASMMAI020W Program exception, code X'*xxxx***'**

Explanation: IDF received control as a result of the exception which occurred during user program execution.

User response: Determine the type and cause of the exception, and perform any required corrective action.

The PSW and register values when the exception occurred are available from the Current Registers window.

ASMMAI021I PER storage alter event by LAST EXECUTED instruction, at *addr*

Explanation: IDF received control as a result of the execution of the previous instruction. This instruction had altered a storage location for which an AdStop storage alteration stop range was active.

User response: None

ASMMAI022I PER register alter event by LAST EXECUTED instruction, at *addr*

Explanation: IDF received control as a result of the execution of the previous instruction. This instruction had altered a general purpose register (GPR) for which a RegStop register alteration stop was active.

User response: None

ASMMAI023E Requested function is not supported in this environment

Explanation: IDF does not support the requested function in this execution environment.

User response: None

ASMMAI025W Warning: Next instruction is outside of target program's boundaries

ASMMAI026W Warning: Can't insert BREAK in R/O storage; IDF may lose control

Explanation: IDF attempted to modify the user program instruction to establish a breakpoint, but was not able to do so as the instruction address is within Read/Only storage.

User response: Establish a suitable breakpoint at an alternate location which can be modified by IDF, and which will be executed as part of the current execution path.

If a suitable alternative location is not available, you must provide a debug execution environment in which this program is in Read/Write storage.

ASMMAI027W No more memory available for PATH function; recording disabled

Explanation: There is insufficient free virtual storage available to allow IDF to record PATH information.

User response: Obtain extra virtual storage.

ASMMAI028E Function not valid when PER is enabled

Explanation: This function is incompatible with the CMS Program Event Recording (PER) feature.

User response: If the function is required, use IDF with PER disabled.

ASMMAI029I Following: *txt1*

ASMMAI030W FOLLOW address above virtual storage maximum; FOLLOW is now OFF

ASMMAI032W Logging of changes to control registers not supported

ASMMAI033E IDF initialization failed. Check your environment, or if IDF ABENDed, Logoff, Logon and try again

ASMMAI034E IDF is already active

ASMMAI035E USER area program not valid in SUBSET mode

ASMMAI037E Unable to load target program into user/trans area

ASMMAI038E SCBLOCK length value does not match expected value

Explanation: IDF has discovered that its subcommand environment has been damaged. This checking is being performed before executing a REXX exit routine, with the CKSUBCM option specified.

IDF was unable to execute the REXX exit routine for this event, but will try again at the next event.

User response: None.

ASMMAI039E SCBLOCK origin value does not match expected value

Explanation: IDF has discovered that its subcommand environment has been damaged. This checking is being performed before executing a REXX exit routine, with the CKSUBCM option specified.

IDF was unable to execute the REXX exit routine for this event, but will try again at the next event.

User response: None.

ASMMAI040E Nucleus extension is not loaded

Explanation: IDF was invoked with the NUCEXT option, but the target program is not currently present in storage as a CMS Nucleus extension.

IDF processing is terminated.

User response: Use the CMS NUCXLOAD command to load the target program as a CMS Nucleus extension, then invoke IDF.

ASMMAI041E TRANS and NUCEXT options are mutually exclusive

Explanation: IDF was invoked with both the TRANS and NUCEXT options specified. This is not a valid combination.

IDF processing is terminated.

User response: Correct the options specification.

ASMMAI042E NUCEXT specified but file built as transient

Explanation: IDF was invoked with the NUCEXT option specified, but the MODULE file has only one record and is a transient.

IDF processing is terminated.

User response: Correct the options specification.

ASMMAI044E Map origin does not match origin in executable file

Explanation:

User response:

ASMMAI045E MODULE file transient; map/options indicate user

Explanation: IDF was invoked without the TRANS option, but the MODULE file has only one record and is a transient.

IDF processing is terminated.

User response: Correct the options specification.

ASMMAI046E Map/options indicate transient, MODULE file does not

Explanation: IDF was invoked with the TRANS option, but the MODULE file has more than one record and is **not** a transient.

IDF processing is terminated.

User response: Correct the options specification.

ASMMAI048E Option not recognized or exceeds 8 characters: ***txt1***

Explanation:

User response:

ASMMAI049E No module name specified

ASMMAI050E "*txt1*" module not found

ASMMAI051E Module name exceeds 8 characters in length

ASMMAI057E COMMAND option specified, but no command was provided

ASMMAI058E Option invalid in this environment: ***txt1***

Explanation: The position of the cursor on the IDF display is not suitable for use as an implicit operand.

User response: Supply an explicit operand, or move the cursor to an acceptable location

ASMMAI060E Unable to load ASMADOP

Explanation: An error occurred when attempting to load the module ASMADOP.

User response: Ensure the module ASMADOP is available to IDF.

ASMMAI061E SELFNUCX specified, but no start symbol provided

ASMMAI062E Start symbol specified on SELFNUCX is not defined: ***txt1***

ASMMAI064E Specified file mode is not valid

ASMMAI065E Not enough free storage; increase virtual memory size and try again

ASMMAI066E Specified PROFILE not found:***fn***

ASMMAI067E Last option is incomplete

ASMMAI068E Unable to create subcommand environment

ASMMAI069E Unable to load module "*txt1*" via OS LOAD

ASMMAI070E IDF is not supported in this environment; RC=*dec1*

ASMMAI071E Terminal not connected at specified address for *txt1*; RC=*dec2*

ASMMAI073E Error in screen manager *txt1* call; RC=*dec2*, Diagnostic info: *txt3*

ASMMAI074E Error reading symbol file *fn ft*; RC=*dec2*

ASMMAI075E Error reading module file *fn ft*; RC=*dec2*

ASMMAI080E SCBLOCK value: *txt1* Expected value: *txt2*

ASMMAI081E SCBLOCK value: *txt1* Expected value: *txt2* SELFNUCX offset: *txt3*

ASMMAI083E Specified address does not contain a valid instruction

ASMMAI084W Main() BREAK adjusted to *txt1*

ASMMAI085E Cursor position not valid for supplying an implicit operand

Explanation: The position of the cursor on the IDF display is not suitable for use as an implicit operand.

User response: Supply an explicit operand, or move the cursor to an acceptable location

ASMMAI087I BREAK set at *txt1*

ASMMAI088I BREAK cleared at *txt1*

ASMMAI089E No BREAK table entries available; clear one and try again

ASMMAI090W BREAK already set at *txt1*

ASMMAI091W Warning: Next instruction is a branch-to-here

ASMMAI095E No ADSTOP table entries available; clear one and try again

ASMMAI096E ADSTOP already set at *txt1*

ASMMAI097E FFFFFFFF is not a valid ADSTOP address

ASMMAI098I Start of ADSTOP range *dec1* set to *txt2*

ASMMAI100I End of ADSTOP range *dec1* set to *txt2*

ASMMAI102E Minimum 22 line screen depth required to run IDF

ASMMAI103E SKIPSTEPs may be set from command line or disassembly window only

ASMMAI104E No SKIPSTEP table entries available; clear one and try again

ASMMAI105I SKIPSTEP set at *txt1*

ASMMAI106I SKIPSTEP cleared at *txt1*

ASMMAI107E Retry count exceeded, *txt1* during *txt2* operation: *txt3*

ASMMAI108W Segment translation exception (program check code X'10')

Explanation: IDF received control as a result of the exception which occurred during user program execution.

User response: Determine the cause of the exception, and perform any required corrective action.

The PSW and register values when the exception occurred are available from the Current Registers window.

ASMMAI109W Page translation exception (program check code X'11')

Explanation: IDF received control as a result of the exception which occurred during user program execution.

User response: Determine the cause of the exception, and perform any required corrective action.

The PSW and register values when the exception occurred are available from the Current Registers window.

ASMMAI112W REPLAY cancelled; terminating as I/O error 99

ASMMAI113A Playback ended; press ENTER to continue

ASMMAI114E Error reading recorded data; RC=*dec1*; terminating as I/O error 99

ASMMAI115E Unknown character entered in *txt1*

ASMMAI116E Address specified for PSW was not on halfword boundary

ASMMAI117E Address specified for PSW was outside of module

ASMMAI118E Wait-state and Mode PSW bits may not be changed

ASMMAI119E Attempt to modify IDF code; changes discarded from *txt2*

ASMMAI120E Non-hex character in hex field; changes discarded from *txt2*

ASMMAI121W Register/PSW display locked out pending execution of startup command

ASMMAI126E BREAKs may be cleared on this panel, but not changed

Explanation: An attempt has been made to alter either the address, commands, or conditions associated with a breakpoint or watchpoint using typeover modification on the Break window. The only supported typeover modification is the clearing of a breakpoint by overtyping it with blanks.

User response: Use the BREAK command to alter breakpoint address or command values. Use the WATCH command to alter watchpoint address, command, or condition values.

ASMMAI127E ADSTOP ranges may be cleared on this panel, but not changed

Explanation: An attempt has been made to alter an AdStop address range using typeover modification on the AdStops window. The only supported typeover modification is the elimination of an AdStop range by overtyping it with blanks.

User response: Use the ADSTOP command to alter the AdStop address range.

ASMMAI128E RLOG ended because command caused nonzero return code or alarm; RC=*dec1*

ASMMAI129E Undefined PF/PA key

ASMMAI134I Oldest item retrieved; next RETRIEVE will wrap to most recent

ASMMAI135E Symbol is undefined

ASMMAI136E Symbol is ambiguous; CSECT qualification is required

ASMMAI137E Expression contains unknown syntax

ASMMAI138E SUBSET is not valid when debugging a transient

Explanation: The SUBSET command is disabled when the target program resides in the CMS transient program area, since the SUBSET command execution causes the target program storage to be overlaid.

User response: Exit the current IDF debug session before executing the command.

ASMMAI139I Returned from saved screen image

ASMMAI140E No screen image has been saved

ASMMAI141E SWAP function is not enabled

ASMMAI142A Press QUIT again to return to *txt1*

Explanation: IDF protects you from accidentally terminating your debug session by requiring confirmation if the QUIT PF key is pressed.

User response: Press QUIT again to exit IDF, or execute any other command to continue this IDF session.

ASMMAI143E The exit routine is compiled-code; XEDEXIT is not relevant

Explanation: The CMPEXIT option indicates that the exit routine is a compiled-code routine, not a REXX exit routine. The XEDEXIT command is only used to invoke XEDIT to edit an interpreted REXX exit routine.

User response: Terminate your IDF session and modify your compiled-code exit routine through other means.

ASMMAI144I Storage pointed to by indicated field is now shown

Explanation: The IDF display has been successfully updated to show storage at the address indicated by the cursor.

User response: None

ASMMAI145E No BREAK address specified

Explanation: The breakpoint address value is required, but was not specified.

User response: Specify a breakpoint address expression

ASMMAI146E Non-ASCII character in text field; changes discarded from *txt1*

Explanation: The ASCII option is ON, so that the text field uses an ASCII data interpretation. An attempt was made to modify one or more text field characters to values which are not valid ASCII characters. Any modifications of the text field data have been discarded.

User response: Retry using valid ASCII characters, or modify the storage location by typeover of the hexadecimal portion of the DUMP data display

ASMMAI148W Display is at end of virtual storage

ASMMAI149W Display is at location zero

ASMMAI150E Function has no meaning in this display mode

ASMMAI151E Function is not valid while PER is disabled

Explanation: This function requires the CMS Program Event Recording (PER) feature.

User response: If the function is required, enable the IDF PER exploitation using the SET PER Y command, or by changing the PER setting in the Breakpoint window to "Y".

ASMMAI152W PATH option is implied and has been set ON

ASMMAI153I Storage containing the specified data is now shown

ASMMAI154W Specified data does not exist at higher addresses

ASMMAI155W Data not found within target program; repeat to continue search

ASMMAI156E Search argument missing or contains unknown syntax

ASMMAI157E Hexadecimal string must contain even number of digits

ASMMAI158E IDF cannot infer an unambiguous starting location for the search

Explanation: A SEARCH command has been issued without an explicit window specification, or cursor indication of the window at whose start location the search is to begin.

Both Dump and Disassembly windows are open, but each has a different start location. IDF is unable to infer which start address to use in this case.

User response: Specify an explicit window specification or place the cursor in the window with the desired search starting location.

ASMMAI159I Exit exec is now ACTIVE

Explanation: Exit routine processing has been enabled. The exit exec will now receive control when an event occurs.

User response: None

ASMMAI160I Exit exec is now disabled

Explanation: Exit routine processing has been disabled. The exit exec will no longer receive control when an event occurs.

User response: None

ASMMAI161E Command not recognized

Explanation: The keyword is not recognized as a valid command.

User response: Change the command specification to use the appropriate keyword.

ASMMAI162E Missing keyword

Explanation: One or more extra keywords were expected, but were not present.

User response: Change the command specification to add the appropriate keywords.

ASMMAI163E Keyword too long

Explanation: The keyword is not recognized as a valid keyword. It exceeds the maximum length of a valid keyword, and may be spelled incorrectly.

User response: Change the command specification to use the appropriate keyword.

ASMMAI164E Keyword not recognized

Explanation: The keyword is not recognized as a valid keyword.

User response: Change the command specification to use the appropriate keyword.

ASMMAI165E Arguments not recognized

Explanation: One or more arguments were not recognized as valid for this command.

User response: Change the command specification to use the appropriate arguments.

ASMMAI166E Conditions do not permit successful execution of the given command

ASMMAI167W STEP function locked out pending execution of startup command

ASMMAI168W Register/PSW display locked out pending execution of startup command

ASMMAI169E PFK may not be set to MACRO without macro name

ASMMAI172E Address missing or does not contain a valid instruction

ASMMAI173I Storage key for address *txt1* **is X'***txt2***'**

ASMMAI176E Specified address *txt1* **exceeds virtual storage size** *txt2*

ASMMAI178I Offset reset to *txt1*

ASMMAI179I Current offset is *txt1*

ASMMAI180I Instructions executed since last ICOUNT command: *dec1*

ASMMAI181E Macro not found: *txt1*

ASMMAI182E SET command RC=*dec1*

ASMMAI183E Exit macro not found: *txt1*

ASMMAI185E *txt1* **is not supported by the PSWSTEAL command**

ASMMAI187E Maximum number of PSWSTEALs already set

ASMMAI188W PER disabled

Explanation: IDF exploitation of the CMS Program Event Recording (PER) feature has been disabled.

User response: None

ASMMAI189I PSWSTEAL set at *txt1*

ASMMAI190E The specified address is in nonexistent or read/only storage

ASMMAI191I PSWSTEAL cleared at *txt1*

ASMMAI192E BREAKs cannot be set within the active copy of IDF

ASMMAI193E Replacement PSW is not valid

ASMMAI194I IDF interrupt save area is 16 bytes at hex location *txt2*

ASMMAI195E Wait-state bit may not be set

ASMMAI196I *txt1* **has been set to** *txt2*

ASMMAI197E No address provided

ASMMAI198I Interrupt Save Area modified; breakin recognized

ASMMAI199E Macro RC=*dec1*

ASMMAI200E LOAD RC=*dec1*

ASMMAI201E RLOG command not allowed while CMDLOG option is set

ASMMAI202E Command file "IDF CMDLOG" not found on any disk

Explanation: IDF was invoked with the RLOG parameter, but a log file was not found.

User response: Create a log file before using the RLOG parameter.

ASMMAI203E Error attempting to read "IDF CMDLOG"; RC=*dec2*

ASMMAI204I *dec1***commands successfully executed; stopped after first "***txt2***"**

ASMMAI205I RLOG terminated by normal end-of-file on "IDF CMDLOG"

Explanation: You issued an RLOG command, but there were no more log records on the log file.

User response: None.

ASMMAI207I Command logging is now active

Explanation: The IDF command logging is now writing commands to the log file.

User response: None.

ASMMAI208E "Old" data does not match the specified value

ASMMAI209E Hex data contains odd number of digits or invalid hex digit

ASMMAI210E Missing "/", type not X/C, or mismatching field sizes

ASMMAI211E Function is only valid when PATH option is on

ASMMAI212E No history available yet

ASMMAI213W No more history information available in that direction

ASMMAI214E PER bit in PSW may not be changed; when PER=Y, IDF controls PER

ASMMAI215E Attempt to access or modify nonexistent or read/only storage

ASMMAI216E WATCH comparator not one of: *txt1*

ASMMAI217E WATCH instruction missing / not one of:*txt1*

ASMMAI218E WATCH instruction operands are missing

ASMMAI219I WATCH condition: *txt1*

ASMMAI221W Warning: WATCH operand address is not valid; unable to evaluate

ASMMAI222E Second operand has incorrect length for the specified opcode

ASMMAI223E Specified window number is invalid

ASMMAI224E Specified window number is not the right type for the command

ASMMAI225E No more windows can be opened

ASMMAI226E Window type invalid for OPEN

ASMMAI227W Attention interrupt trapped

ASMMAI228I Extract file automatically loaded for CSECT *txt1*

ASMMAI229S IDF *txt2* **support component failed to initialize; RC=***dec2*

Explanation: This message indicates that a required component failed initialization.

User response: Ensure invocation parameters are correctly specified, or determine reason for failure (eg, check the system log for accompanying messages). If an accompanying message IFA104I is issued, please refer to *z/OS Planning for Installation* for product enablement.

ASMMAI237E No module name specified

ASMMAI238E Module name too long

ASMMAI239E Macro name too long

ASMMAI241E Module name not valid

ASMMAI242E Option value must be ON or OFF or cleared

ASMMAI243E A Deferred BREAK for module *txt1* failed to install; RC=*dec2*

ASMMAI245I Deferred BREAK set at *txt1*

ASMMAI246I Deferred BREAK cleared at *txt1*

ASMMAI247W Deferred BREAK already set for *txt1*

ASMMAI248I Module *txt1* loaded and Deferred BREAKs installed

ASMMAI250E Deferred BREAKs must be set by the DBREAK command

ASMMAI252E SVC 97 used; only the address and ASC bits of PSW may be modified

ASMMAI253E PROGCK supports only decimal codes 1 through 15

ASMMAI254E Invalid number

ASMMAI255E Language support window not open

ASMMAI256E Extraneous parameters

Explanation: One or more extraneous parameters was encountered during processing of the IDF invocation parameters.

IDF processing is terminated.

User response: Correct the invocation parameters specification.

ASMMAI257W Target program check (code X'*txt1*'); intercepted by ESPIE

ASMMAI262W Program apparently self-modifying; IDF BREAK instruction overlaid

ASMMAI264E Target program not yet loaded; command rejected

ASMMAI265E Too late for "*txt1*" command

Explanation: The command cannot be requested after IDF has started.

User response: Include the command as an invocation parameter, or in the PROFILE macro.

ASMMAI266E Conflicting options specified

ASMMAI272I Value is *txt1* *

ASMMAI275E AMODE option cannot be set off

Explanation: The SET OPTION OFF command was used to attempt to turn off any of the AMODE options AMODE24, AMODE31, or AMODE64. It is not possible to turn these options off.

User response: Use the SET OPTION ON command to change the AMODE to the required setting.

ASMMAI500E LOSTERM for LU *luid* - reason code *code*x

Explanation: LOSTERM was driven for the specified VTAM Logical Unit name *luid* with the reason code *code* (in hexadecimal).

User response: Determine from the reason code, why the session with the LU was terminated and correct before resubmitting IDF job.

ASMMAI501E *function* error for LU *luid* - RTNCD *rtncd*x FDB2 *fdb2*x

Explanation: The RPL-based function *function* failed for the specified VTAM Logical Unit name *luid* with the indicated RTNCD and FDB2 fields from the RPL.

User response: Determine from the reported RPL feedback information, why the function with the LU failed and correct before resubmitting IDF job.

ASMMAI502E ACB error for APPLID *applid* - ACBERFLG *code*x

Explanation: An OPEN or CLOSE for the VTAM application ID *applid* failed with the indicated ACBERFLG field from the ACB. If the ACBERFLG code is x'5A', IDF attempts OPEN for up to 10 successive application IDs (of the form ASMTL*nnn*) before issuing the message and hence the reported *applid* may be one that is not actually defined.

User response: Determine from the reported ACBERFLG, why the OPEN or CLOSE failed.

ADATA extraction utility messages

ASMXMA001I ASMLANGX Version 1 (Release 4)

Explanation: This message shows the ASMLANGX program identification, version, and release date.

This message, as well as most other ASMLANGX messages, is normally only displayed if the ASMLANGX LOUD option is specified.

User response: None

ASMXMA002I Output file: *fn ft fm*

Explanation: This message identifies the file to which the extract data information will be written by ASMLANGX.
- On CMS, this is a standard CMS file identifier
- On z/OS, this is mapped as follows:

CMS	**Equivalent on TSO**
fn	PDS member name (ignored if using sequential file)
ft	DDNAME, which in turn points to the TSO data set name
fm	Not used on TSO

User response: None

ASMXMA003I ... scanning *txt1*

Explanation: This message indicates that the information specified in *txt1* is being read from the associated file and processed.

User response: None

ASMXMA004I ... checking *txt1*

Explanation: This message indicates that the information specified in *txt1* is checked for consistency.

User response: None

ASMXMA005I *txt1* **Pass** *dec2* **processing begins**

Explanation: This message indicates pass **dec2** of the multi-pass processing task specified in *txt1* is now being performed.

User response: None

ASMXMA006I Post-processing begins

Explanation: This message indicates that all the needed information has been read from the associated files, and post-processing of this information is being performed.

User response: None

ASMXMA007I ... matching *txt1*

Explanation: This message indicates that the information specified in *txt1* is now being correlated.

User response: None

ASMXMA008I ... performing *txt1*

Explanation: This message indicates that the processing step specified in *txt1* is now being performed.

User response: None

ASMXMA010I *txt1* **records scanned:** *dec2*

Explanation: This message indicates that *dec2* records were read from the *txt1* file when the current compile unit was processed by ASMLANGX.

User response: None

ASMXMA011I ...Symbols *txt1*.. *dec2*

Explanation: This message indicates that the current compile unit contained *dec2* symbols with characteristics of type *txt1*

User response: None

ASMXMA013I ...Total symbols: *dec1*

Explanation: This message indicates that the current compile unit contained *dec1* symbols.

User response: None

ASMXMA014I Records written to output file: *dec1*

Explanation: This message shows the number of records of extract data information which were written to the output file.

User response: None

ASMXMA015I Operation completed for this compile unit

Explanation: Processing has completed for the current compile unit. If more compile units are present, processing continues.

User response: None

ASMXMA016I *txt1 txt2*

Explanation: This message identifies the input files which were processed by ASMLANGX

The *txt1* field is normally "Input file:" or "Input files:".

See the explanation for message ASMX002I for a full description of the *txt2* information.

User response: None

ASMXMA017I Operation completed for this extract file

Explanation: This is the last message to be displayed by ASMLANGX, and indicates that processing has completed for this ASMLANGX extract data file.

User response: None

ASMXMA018I *txt1* **bytes scanned:** *dec2*

Explanation: This message indicates that *dec2* bytes of data were read from the *txt1* file when the current compile unit was processed by ASMLANGX.

User response: None

ASMXMA050W Argument missing for *txt1* **option.** *txt2*

Explanation: The argument for ASMLANGX option *txt1* was not found during processing of the ASMLANGX invocation parameters.

The default argument for the *txt1* option is assumed.

User response: Specify the missing argument, or remove the option from the invocation parameters.

ASMXMA051S Argument/Option too long, "*txt1***"**

Explanation: The invocation parameter *txt1* is not recognized as a valid ASMLANGX argument (or option). It exceeds the maximum length of a valid argument (or option), and may be spelled incorrectly.

ASMLANGX processing is terminated.

User response: Correct the invocation parameter.

ASMXMA052S Argument/Option not recognized, "*txt1***"**

Explanation: The invocation parameter *txt1* is not recognized as a valid ASMLANGX argument (or option).

ASMLANGX processing is terminated.

User response: Correct the invocation parameter.

ASMXMA054S File mode is too long, "*txt1***"**

Explanation: The CMS file mode specification *txt1*, which is longer than 2 characters, was encountered during processing of the ASMLANGX invocation parameters.

ASMLANGX processing is terminated.

User response: Correct the file mode specification.

ASMXMA055S A left parenthesis was found inside options

Explanation: An extra left parenthesis, after the initial left parenthesis which signals the start of the ASMLANGX options, was encountered during processing of the ASMLANGX invocation parameters.

ASMLANGX processing is terminated.

User response: Correct the options specification.

ASMXMA056S No file name was specified

Explanation: The file name (on CMS, PDS member name on TSO) of the primary program information file from which source and variable data is to be extracted was not found during processing of the ASMLANGX invocation parameters.

ASMLANGX processing is terminated.

User response: Specify the name of the primary program information file.

ASMXMA057S Argument/Option already specified, "*txt1***"**

Explanation: The argument (or option) *txt1* has been encountered more than once during processing of the ASMLANGX invocation parameters.

ASMLANGX processing is terminated.

User response: Remove the duplicate argument (or option).

ASMXMA058S Argument/Option "*txt1***" conflicts with previous Argument/Option**

Explanation: A conflict between the argument (or option) *txt1* and another previously specified argument (or option) has been detected during processing of the ASMLANGX invocation parameters.

ASMLANGX processing is terminated.

User response: Remove the conflicting argument (or option).

ASMXMA059I Application language not specified, option "*txt1***" assumed**

Explanation: In the absence of an ASMLANGX option which explicitly specifies the application programming language, the ASMLANGX option *txt1* was assumed.

ASMLANGX processing continues.

User response: Specify the application language, using the appropriate ASMLANGX option.

ASMXMA100S ***txt1*** **contains NO recognized records**

ASMXMA101W Warning - ***txt1*** **file newer than** ***txt2*** **file**

ASMXMA102T ***txt1*** **file has more** ***txt2*** **than** ***txt3*** **file**

ASMXMA103S ***txt1*** **has unrecognized records following last valid section**

ASMXMA104W Expected: ***dec1*****, actual:** ***dec2*** **at line** ***dec3***

ASMXMA105W ...Symbols ***txt1*****..** ***dec2***

ASMXMA111S ***txt1*** **not supported - fatal**

ASMXMA114S ***txt1*** **required for source support - fatal**

ASMXMA115W ***txt1*** **required for symbol support**

ASMXMA116W ***txt1*** **required for structure/union support**

ASMXMA120W ***txt1*** **detected.** ***txt2*** **option assumed**

Explanation: The format of the input file indicates that the specified option is no longer in effect.

ASMLANGX processing continues, assuming an appropriate option to match the format of the input file.

User response: Use the correct compiler option, or make the compiler directive which adjusted the compiler option visible to ASMLANGX, as appropriate.

ASMXMA130S File not found "*txt1*"

Explanation: The ASMLANGX extract data file *txt1* could not be found to allow ASMLANGX processing to begin.

ASMLANGX processing is terminated.

User response: Correct the file specification, or make the file available to ASMLANGX, as appropriate.

ASMXMA131S Files not found "*txt1*", and "*txt2*"

Explanation: The ASMLANGX extract data file could not be found using either the primary file identifier *txt1*, or the alternative file identifier *txt2* to allow ASMLANGX processing to begin.

ASMLANGX processing is terminated.

User response: Correct the file specification, or make the file available to ASMLANGX, as appropriate.

ASMXMA132S Input or Output file format invalid

Explanation: The attributes or contents of a file have been found to be inappropriate, during ASMLANGX processing.

One or more preceding messages will identify the file which was being processed when the error occurred.

ASMLANGX processing is terminated.

User response: Correct the problem identified in the preceding message.

ASMXMA133S File DD not allocated "*txt1*"

Explanation: The Data Definition (DD) for the *txt1* file was found to be unallocated.

ASMLANGX processing is terminated.

User response: Allocate the file, using a JCL DD statement, or TSO ALLOCATE statement, as appropriate.

ASMXMA134S File DDs not allocated "*txt1*", and "*txt2*"

Explanation: The Data Definitions (DDs) for the both the primary *txt1* file and the alternative *txt2* file were found to be unallocated.

ASMLANGX processing is terminated.

User response: Allocate the file, using a JCL DD statement, or TSO ALLOCATE statement, as appropriate.

ASMXMA135S ***txt1*** **file incorrectly defined**

Explanation: The attributes of the *txt1* file have been examined, and found to be inappropriate.

ASMLANGX processing is terminated.

User response: Ensure that the correct data set has been specified in the *txt1* file allocation. If the correct data set was specified, the data set has been allocated with incorrect attributes, in which case it must be reallocated.

ASMXMA136S Premature ***txt1*** **End-of-File encountered**

Explanation: ASMLANGX had begun scanning the *txt1* file data, but the file ended before all expected data records had been scanned.

ASMLANGX processing is terminated.

User response: Ensure that the correct data set has been specified in the *txt1* file allocation. If the correct data set was specified, the file may have been truncated and must be replaced with the complete data.

ASMXMA137S *txt1* **disk/directory is full**

Explanation: There is insufficient space to write further records to the *txt1* file.

- On CMS, this may be caused by:
  - minidisk free space being exhausted
- On z/OS, this may be caused by:
  - PDS directory having no free entries
  - data set having maximum number of extents
  - insufficient free space on the DASD volume for another extent
- On z/VSE, this may be caused by:

ASMLANGX processing is terminated.

User response: Determine the resource which has been exhausted, and correct as appropriate.

ASMXMA138T Insufficient virtual memory available

Explanation: There is insufficient free storage for ASMLANGX to continue processing.

ASMLANGX processing is terminated.

User response: Free up virtual storage which is currently in use, or make more virtual storage available, as appropriate.

ASMLANGX exploits storage above the 16 MB line, if it is available.

ASMXMA139S File is TERSEd or PACKed "*txt1***"**

Explanation: The specified file was found to have a Fixed record format, and 1024-byte record length. It was likely compressed using TERSE or COPYFILE.

ASMLANGX processing is terminated.

User response: Restore the file to its original format, using the appropriate utility program.

ASMXMA150T Maximum number of symbols exceeded

Explanation: The maximum number of symbols that a single compile unit can contain is 65534. This limit is exceeded by the current compile unit.

ASMLANGX processing is terminated.

User response: Reduce the number of symbols below the limit.

ASMXMA151T Extract information not available for symbol *dec1*

Explanation:

User response:

ASMXMA152W Incomplete information for symbol "*txt1***" (ident:** *dec2***)**

Explanation: During the extraction process, complete information was not available for the symbol shown. The extract data for unrelated symbols and program source is not affected.

ASMLANGX processing continues.

ASMXMA153W Declared at line *dec1***, stmt** *dec2*

Explanation: This message is associated with ASMX152W, and provides more information to allow the symbol to be easily identified.

User response: None

ASMXMA154W Unknown *txt1 txt2* **token, "***txt3***" at line** *dec4* **of** *txt5*

ASMXMA155W Member *txt1* **dot qualification invalid at line** *dec2* **of** *txt3*

ASMXMA156W Incomplete *txt1* **information detected at line** *dec2* **of** *txt3*

ASMXMA157W No ESD entry for external definition "*txt1***"**

ASMXMA159W Pseudo-Assembly synchronization error at line *dec1* **of** *txt2*

ASMXMA160W Unable to locate *txt1* **for label "***txt2***"**

ASMXMA161W *txt1* **ignored at line** *dec2* **of** *txt3*

ASMXMA162W - check for *txt1* **suppression via** *txt2*

ASMXMA163W Possible candidate procedure/functions:

ASMXMA164W - name: *txt1*

ASMXMA165W defined at *txt1* **(***txt2***)**

ASMXMA166W cross reference entry at line *dec1* **of listing**

ASMXMA167W Within dead code, *txt1*

ASMXMA168W ***txt1*** **attribute conflict for symbol "*txt2*" (ident:** ***dec3*****)**

ASMXMA200W ***txt1*****. Contact IBM Support Center**

ASMXMA201T ***txt1*** **is terminated. Contact IBM Support Center**

ASMXMA210S Extraction for ***txt1*** **is not supported (by this version of** ***txt2*****)**

ASMXMA211W Extraction for your release of ***txt1*** **is not supported (by this version of** ***txt2*****)**

ASMXMA212W The ***txt1*** **which produced** ***txt2*** **could not be identified**

ASMXMA213W Verify that mandatory ***txt1*** **maintenance has been applied**

ASMXMA214S ***txt1*** **error messages detected at record** ***dec2*** **of** ***txt3***

ASMXMA220W ***txt1*** **overflowed**

ASMXMA221W ***txt1*** **overflowed at record** ***dec2*** **of** ***txt3***

ASMXMA222W ***txt1*** **encountered at record** ***dec2*** **of** ***txt3***

ASMXMA223W ***txt1*** **encountered for stmt** ***dec2*** **at record** ***dec3*** **of** ***txt4***

ASMXMA224W ***txt1dec2*** **encountered for stmt** ***dec3*** **at record** ***dec4*** **of** ***txt5***

ASMXMA225W ***txt1 txt2*** **encountered at record** ***dec3*** **of** ***txt4***

ASMXMA226W ***txt1 dec2*** **mismatch at record** ***dec3*** **of** ***txt4***

ASMXMA228W Alias missing, id=*dec1*

ASMXMA229W No ***txt1*** **were detected**

ASMXMA231S Missing ***txt1*** **ESD information**

ASMXMA232S Unable to determine location of program code/data

ASMXMA233S Unable to determine identity of unnamed PC Section

ASMXMA235W The DSA base reg ***txt1*** **is not defined to** ***txt2***

ASMXMA240S ***txt1*** **extraction is only supported for** ***txt2***

ASMXMA241S For ***txt1*****, enable the** ***txt2*** **"*txt3*" option**

ASMXMA242S and make the ***txt1*** **file available to ASMLANGX**

ASMXMA311T Maximum number of statements exceeded

Explanation: The maximum number of statements that a single compile unit can contain is 65535. This limit is exceeded by the current compile unit. ASMLANGX processing is terminated.

User response: Reduce the number of statements below the limit.

Appendix C. Abbreviations

The following list shows the minimum abbreviation for all IDF keywords in UPPER case. Keywords added or changed in this version are flagged with an asterisk:

$$	CMDLog	LIKE	OPTions	R2
$WHERE	CMDMsg	LINE	ORDer	R3
?	CMPEXIT	LOad	OREGs	R4
=	COLors	LOCATIon	PASspgm	R5
ABEND	COMmand	LSM	PATH	R6
ADStops	CREGs	LSMDebug	PATHFile	R7
AFPR	CURsor	LSMProf	PAUSe	R8
ALArm	DFLTLsm	LUtype	PER	R9
ALet	DISasm	MACro	PFK	SBORDer
AMODE24	DMS0	MACROLog	PFKDisp	SCDactiv
AMODE31	DROp	MODE	PLIST	SELFNucx
AMODE64	DUMP	MODMap	Previous	SET
AREGS	DUMPMode	MODUles	PROfile	SIZe
ARGument	EPNAMES	MOVe	PROGchk	SKIPstep
AR0	EPOffset	MRUn	PROGck	STATus
AR1	EVEnt	MSG	PSW	STAY
AR10	EXItexec	MSG1	PSWSTEAL	STEp
AR11	EXTract	MSG2	QUAlify	STMTstep
AR12	FASTPath	MSTep	QUIt	STOKey
AR13	FINd	Next	QWDump	STOPNOP
AR14	FOllow	NLS	RCQuit	STOPSTmt
AR15	FPC	NOAUTOLd	REFresh	STRucture
AR2	FPR	NOAUTOSz	REGs	SUBset
AR3	FULLQual	NOBcx	REGS64	SVC
AR4	GLObal	NODSects	REGSTops	SVC97
AR5	GPR	NOIMPMac	RETRieve	SWAp
AR6	GPRG	NOINVPsw	RIght	SYMbols
AR7	GPRH	NOMODMap	RISk	SYStem
AR8	HEXDisp	NOPROfil	RLog	TRACeall
AR9	HEXInput	NOSTOPNp	ROWstyle	TRANs
ASCii	HISTory	NOSTOPSt	RUN	UNFtdump
AUTOLoad	ICOunt	NOSVC97	RUNExit	UNTil
AUTOSize	IMPMacro	NUCext	R0	VALue
BACK	INVPsw	OFF	R1	VARiable
BASe	ISA	OFFSet	R10	VChange
BCX	KWDSYN	OLDBREAK	R11	VERsion
BREak	LANguage	ON	R12	VS
CDE	LASTmsg	OPCODE	R13	WATch
CKSubcm	LEft	OPEn	R14	WINDows
CLOse	LIBE	OPTion	R15	XEDexit
				1ADStop

Note: The following keywords are used for maintenance commands and can therefore not be used as macro names: $WHERE, $$.

Appendix D. Performance considerations

When IDF displays an area of memory, in either disassemble or dump format, the screen display is created dynamically, but a page-forward operation should occur at essentially the data transfer rate of the terminal.

If statement source is being displayed, the actual source text is obtained dynamically from the IDF Language extract data file. The extra overhead compared to pure disassembly is minimized by the design of the IDF Language Support.

Some delays may be noticed on heavily loaded systems, but these are generally in the nature of scheduling delays, where IDF is waiting its turn at the processor.

There is a general belief that using PER will cause a serious system-wide performance impact. When using CP PER this belief is probably well justified, since there is significant processing for any instruction that modifies storage, because CP performs a partial disassembly to determine where storage was modified. Since this processing is happening on the CP level it can seriously affect performance on a system-wide basis.

When IDF is used for a similar function, there should be no significant system-wide performance penalty above that imposed by any processor-bound application. Instead of disassembling instructions, CP needs only to report an interrupt to the virtual machine in which IDF is operating.

When the PATH option is OFF, the target program runs at what amounts to full speed between breakpoints. When the PATH option is ON, IDF causes an interrupt following every target program instruction.

Overhead per breakpoint varies in the approximate range of 300-500 IDF instructions per breakpoint. This includes prioritized breakpoint insertion, target activation, interrupt occurrence, event recognition, and saving the target program's state. The variation is as follows, with the first item shown being the least costly and the last item shown being the most costly:
1. PER OFF, PATH OFF
2. PER ON, PATH OFF
3. PER OFF, PATH ON
4. PER ON, PATH ON

Even though the PATH option can be relatively costly in terms of processor usage, it is extremely useful both for collecting code coverage information and for "back-tracking" by means of the HISTORY command.

Exit routines should also be used with some caution, since a REXX exec is executed each time a breakpoint event occurs.

When the list of subroutines to be skipped when single-stepping, statement stepping, or running with PATH or FASTPATH active is not empty there is more overhead on every "subroutine call instruction" (for example, BALR, BAL, BASSM) that is executed. There is a binary search of the list of subroutines to be skipped to determine if the subroutine being called is supposed to be skipped.

Appendix E. Migrating from TSO/E TEST to IDF

This section describes some considerations for those users of IDF who are migrating from a debug setup which used the TSO/E TEST debugger.

General considerations

TSO/E TEST and IDF have different command syntaxes. This section does not provide a detailed comparison of the two command sets. However, procedures are described to perform operations equivalent to those under TSO/E TEST.

The expression syntaxes are somewhat different. Instead of TSO/E TEST's "modname.csect.address" IDF uses "(modname.csect)address". The CSECT name may be omitted if the address is unique within the module. The module name may be omitted, if the address is in the qualified module, which is set by the QUALIFY command. For register specifications in expressions IDF uses "n(Rn)" instead of TSO/E TEST's "nR%+n". IDF supports the same "%" and "?" indirection operators as TSO/E TEST. However, the first indirection operator following a register specification is used to determine the size of the address in the register. For more details on expressions in IDF see “Address expressions” on page 80.

Some significant advantages of IDF over TSO/E TEST are:
- IDF is a full screen debugger
- IDF can single-step
- IDF is easy to customize through its REXX macro capabilities.
- IDF has a sophisticated cursor addressing support which minimizes the amount of typing you have to do. For more details see “Intelligent cursor sensing” on page 5.

TSO/E TEST does have some advantages over IDF:
- TSO/E TEST has automatic symbol support for all load modules loaded.

  ASMIDF provides the following alternatives:
  - For dynamically loaded modules, the IDF DBREAK command should be used. When the module is fetched into storage, IDF *will* automatically load symbols.
  - The process of manually identifying extra modules in storage to IDF and loading symbols requires only two commands: MODULE CDE and LOAD SYMBOLS.

Invoking the target program

Properly invoking IDF can easily get you breakpoints in modules that get loaded by another. For example, here is what you do if you want to debug the program XYZZY that runs as an ISPF dialog, using TSO TEST:

```
TEST 'SYS1.ISP.LINKLIB(ISPF)' CP
ISPSTART PGM(XYZZY)
AT XYZZY.XYZZY.+0 DEFER
```

Figure 28. Invoking an ISPF dialog using TSO TEST

Here is what you do using using ASMIDF:

```
ASMIDF ISPF (COMMAND/ISPSTART PGM(XYZZY)
```

once in IDF,

```
DBREAK (XYZZY.)
```

Figure 29. Invoking an ISPF dialog using IDF

This can only be done if you are using SVC 97 for breakpoints.

Specifying the target program parameters

As shown in the example above, another difference between IDF and TSO/E TEST is in the debugging of TSO command processors. With TSO/E TEST adding the option CP to the TEST command tells TSO/E TEST that the program being debugged is a TSO command processor. TSO/E TEST then prompts you for the command to be passed to the target in the form of a Command Processor Parameter List (CPPL). Under IDF, the COMMAND option instructs IDF that the parameter string is not a parameter for the target but is a command that should be executed when the target is first started (with the RUN or MRUN commands). If the command being executed is the target program, then you can debug a TSO command which is passed a CPPL. A few examples may help illustrate this:

1. If you wanted to debug a REXX/TSO function package, for example, RXLOCFN:
   - Here is what you do using TSO TEST:
     ```
     TEST 'SYS1.LINKLIB(REXX)' CP
     EXEC REXTRY
     AT RXLOCFN.RXLOCFN.+0 DEFER
     ```
   - Here is what you do using IDF:
     ```
     ASMIDF RXLOCFN (COMMAND/EXEC REXTRY
     ```

     once in IDF,
     ```
     BREAK (RXLOCFN).
     ```
   - In this example IDF is not passing control to RXLOCFN but to EXEC and the parameter string that is passed as a CPPL is for EXEC.
2. If you wanted to debug a command processor, INVOKE in this case:
   - Here is what you do using TSO TEST:
     ```
     TEST LOAD(INVOKE) CP
     INVOKE IEBGENER
     ```
   - Here is what you do using IDF:
     ```
     ASMIDF INVOKE (COMMAND/INVOKE IEBGENER
     ```
   - In this example IDF is passing control to INVOKE and the parameter string that is passed as a CPPL is for INVOKE.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Trademarks

Bibliography

High Level Assembler Documents

HLASM General Information, GC26-4943

HLASM Installation and Customization Guide, SC26-3494

HLASM Language Reference, SC26-4940

HLASM Programmer's Guide, SC26-4941

Toolkit Feature document

HLASM Toolkit Feature User's Guide, GC26-8710

HLASM Toolkit Feature Debug Reference Summary, GC26-8712

HLASM Toolkit Feature Interactive Debug Facility User's Guide, GC26-8709

HLASM Toolkit Feature Installation and Customization Guide, GC26-8711

Related documents (Architecture)

z/Architecture Principles of Operation, SA22-7832

Related documents for z/OS

z/OS:

z/OS MVS JCL Reference, SA23-1385

z/OS MVS JCL User's Guide, SA23-1386

z/OS MVS Programming: Assembler Services Guide, SA23-1368

z/OS MVS Programming: Assembler Services Reference, Volume 1 (ABE-HSP), SA23-1369

z/OS MVS Programming: Assembler Services Reference, Volume 2 (IAR-XCT), SA23-1370

z/OS MVS Programming: Authorized Assembler Services Guide, SA23-1371

z/OS MVS Programming: Authorized Assembler Services Reference, Volumes 1 - 4, SA23-1372 - SA23-1375

z/OS MVS Program Management: User's Guide and Reference, SA23-1393

z/OS MVS System Codes, SA38-0665

z/OS MVS System Commands, SA38-0666

z/OS MVS System Messages, Volumes 1 - 10, SA38-0668 - SA38-0677

z/OS V2R1.0 Communications Server: SNA Programming, SC27-3674

UNIX System Services:

z/OS V2R1.0 UNIX System Services User's Guide, SA23-2279

DFSMS/MVS:

z/OS DFSMS Program Management, SC27-1130

z/OS DFSMSdfp Utilities, SC23-6864

TSO/E (z/OS):

z/OS TSO/E Command Reference, SA32-0975

SMP/E (z/OS):

SMP/E for z/OS Messages, Codes, and Diagnosis, GA32-0883

SMP/E for z/OS Reference, SA23-2276

SMP/E for z/OS User's Guide, SA23-2277

Related documents for z/VM

z/VM: VMSES/E Introduction and Reference, GC24-6243

z/VM: Service Guide, GC24-6247

z/VM: CMS Commands and Utilities Reference, SC24-6166

z/VM: CMS File Pool Planning, Administration, and Operation, SC24-6167

z/VM: CP Planning and Administration, SC24-6178

z/VM: Saved Segments Planning and Administration, SC24-6229

z/VM: Other Components Messages and Codes, GC24-6207

z/VM: CMS and REXX/VM Messages and Codes, GC24-6161

z/VM: CP System Messages and Codes, GC24-6177

z/VM: CMS Application Development Guide, SC24-6162

z/VM: CMS Application Development Guide for Assembler, SC24-6163

z/VM: CMS User's Guide, SC24-6173

z/VM: XEDIT User's Guide, SC24-6245

z/VM: XEDIT Commands and Macros Reference, SC24-6244

z/VM: CP Commands and Utilities Reference, SC24-6175

Related documents for z/VSE

z/VSE: Guide to System Functions, SC33-8312

z/VSE: Administration, SC34-2627

z/VSE: Installation, SC34-2631

z/VSE: Planning, SC34-2635

z/VSE: System Control Statements, SC34-2637

z/VSE: Messages and Codes, Vol.1 , SC34-2632

z/VSE: Messages and Codes, Vol.2, SC34-2633

z/VSE: Messages and Codes, Vol.3, SC34-2634

REXX/VSE Reference, SC33-6642

REXX/VSE User's Guide, SC33-6641

Glossary

A

ABEND
An acronym for ABnormal END, the termination of a task or job before its completion because of an error condition that cannot be resolved by error recovery facilities while the task or job is running.

Access Register
Specialized high speed storage location, 32 bits in size. A z/Architecture processor has 16 ARs, A0-A15. The AR contents are used in ESA/370 (or ESA/390) mode to specify the dataspace to be used by an ESA-exploitative machine instruction.

active block
The currently executing block that invokes the Interactive Debug Facility or any of the blocks in the CALL chain that leads up to this one.

Address Stop
See *Storage Alteration Stop*.

AdStop
See *Storage Alteration Stop*.

AdStops window
The IDF window which lists the currently active Storage Alteration Stops (AdStops) and Register Alteration Stops (RegStops). This window is only available on CMS, when IDF's of PER is enabled.

alias An alternative name for a field.

AR See *Access Register*.

argument
Data passed from one program or procedure to another. Contrast with *parameter*.

ASMIDF
See *Interactive Debug Facility*.

assemble
To translate a program written in assembly language into a machine-language program.

assembler
A program that translates instructions written in assembly language into machine language.

attention interrupt
An I/O interrupt caused by a terminal or workstation user pressing an attention key, or its equivalent.

attention key
A function key on terminals or workstations that, when pressed, causes an I/O interrupt in the processing unit.

attribute
A characteristic or trait you can specify.

| **B**

batch Pertaining to a predefined series of actions performed with little or no interaction between a user and the system. Contrast with *interactive*.

batch job
A job submitted for batch processing. See *batch*. Contrast with *interactive*.

block In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it.

Break window
The IDF window which lists the currently active breakpoints. These include those breakpoints set by the BREAK, DBREAK, and WATCH commands.

breakpoint
A place in a program, normally specified by a command or a condition, where execution can be interrupted and control given to the user or to the Interactive Debug Facility.

C

Command window
The IDF window which contains the command input area, the message display areas, and (optionally) the display of the PF key captions.

compile
To translate a program written in a high level language into a machine-language program.

compile unit
A sequence of statements that make a

portion of a program complete enough to compile (or assemble, as appropriate) correctly. Each language has different rules for what comprises a compile unit.

compiler
A program that translates instructions written in a high level programming language into machine language.

condition
Any synchronous event that may need to be brought to the attention of an executing program or the language routines supporting that program. Conditions fall into two major categories: conditions detected by the hardware or operating system, which result in an interrupt; and conditions defined by the programming language and detected by language-specific generated code or language library code. See also *exception*.

constant
A name used to represent a data item whose value cannot be changed while the program is running. Contrast with *variable*.

Control Register
Specialized control facility, 32 bits in size. A z/Archtecture processor has 16 GPRs, C0-C15. The CR contents are used to control processor modes and facilities

CR See *Control Register*.

Current Registers window
The IDF window which displays the current PSW and registers. By default, the current General Purpose Registers (GPRs) and Floating Point Registers (FPRs) are shown, but the current Access Registers (ARs) or Control Registers (CRs) may be shown instead.

currently qualified
See *qualification*.

D

data type
A characteristic that determines the kind of value that a field can assume.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

DBCS See *double-byte character set*.

DCSS DisContiguous Shared Segment (CMS).

debug To detect, diagnose, and eliminate errors in programs.

default
A value assumed for an omitted operand in a command. Contrast with *initial setting*.

disassemble
To translate machine language into assembly language instructions and data statements.

disassembler
A program that translates machine language into assembly language instructions and data statements.

disassembly
The assembly language instructions and data statements which result from using a disassembler to disassemble machine language.

The act of using a disassembler.

Disassembly window
The IDF windows which display the results of an IDF DISASM command. This storage display will normally take the form of disassembled machine instructions. If IDF Language extract data for these storage locations has been loaded with LANGUAGE LOAD, then program source statements will be shown interleaved with the machine instructions. You can control the display of source and disassembly with various IDF commands.

display
A visual presentation of information about a workstation, normally in a specific format. Sometimes called a screen or panel.

display attribute
A characteristic that determines how an item appears on the display. Display attributes can include the color of an item. See also *type style*.

display line
A viewable line of text in a window, whose exact appearance is determined by factors such as window size.

DLBL z/VSE only. Disk label information.

Further information is available in *z/VSE: System Control Statements*.

double-byte character set (DBCS)
A set of characters in which each character is represented by two bytes. Languages such as Japanese, which contain more symbols than can be represented by 256 code points, need double-byte character sets. Because each character needs two bytes, the typing, displaying, and printing of DBCS characters needs hardware and programs that support these characters.

dump The formatted display of storage contents. Typically formatted with a hexadecimal representation of the data on the left, and a character interpretation on the right.

A file or data set containing storage contents, intended for use in offline debug. See also *postmortem debug*.

Dump window
The IDF windows which display the results of an IDF DUMP command. This storage display will take the form of a dump, with the left side of the display containing the storage contents in hexadecimal, and the right side containing a character interpretation.

dynamic
In programming languages, pertaining to properties that can only be established during the execution of a program; for example, the length of a variable-length data object is dynamic. Contrast with *static*.

E

entry point
The address or label of the first instruction executed on entering a computer program, routine, or subroutine. A computer program may have a number of different entry points, each perhaps corresponding to a different function or purpose.

exception
An abnormal situation in the execution of a program which typically alters its normal flow. See also *condition*.

execute
To cause a program, utility, or other machine function to carry out the instructions contained within. See also *run*.

execution time
See *run time*.

execution-time environment
See *run-time environment*.

exit exec
The same as an *exit routine*.

exit routine
A customization feature which associates an IDF macro or another program with a particular event.

expression
A group of constants or variables separated by operators that yields a single value. An expression can be arithmetic, relational, logical, or a character string.

F

file A named set of records stored or processed as a unit.

A system object containing records: for example, a VM file, or an z/OS member or partitioned data set. See *data set*.

Floating Point Register
The CPU has 16 floating-point registers. The floating point registers are identified by the numbers 0-15 and are designated by a four-bit R field in floating point instructions. Each floating-point register is 64 bits long and can contain either a *Short* (32-bit) or a *Long* (64-bit) floating-point operand.

FPR See *Floating Point Register*.

font A set of characters or symbols of a given size, shape, and style.

frequency count
In the Interactive Debug Facility, a count of the number of times statements in the currently qualified program unit have been run.

Frequency
A choice located on the Compact Source or Compact Listing window action bar that allows you to monitor the frequency with which program statements are carried out.

full-screen mode
An interface mode for use with a

non-programmable terminal which displays a variety of information about the program you are debugging.

G

General Purpose Register
Each register contains 64 bit positions. The general registers are identified by the numbers 0-15.

GPR See *General Purpose Register*.

group A set of records that are associated together as a logical unit.

H

high level language (HLL)
A programming language such as C, PL/I, or COBOL.

HLL See *high level language*.

HLASM
Acronym for High Level Assembler.

I

Interactive Debug Facility
The IBM product informally known as IDF, an application development and maintenance facility for debugging assembly language programs.

Interactive Debug Facility macro
A REXX EXEC which contains Interactive Debug Facility commands.

inactive block
A block that is not currently executing, or is not in the CALL chain leading to the active block. See also *active block*, *block*.

initial setting
A value in effect when the user's Interactive Debug Facility session begins. Contrast with *default*.

interactive
Pertaining to a program or system that alternately accepts input and then responds. An interactive system is conversational; that is, a continuous dialog exists between a user and the system. Contrast with *batch*.

I/O Input/output.

L

Language Support Module
The IDF subsystem which provides Language Support extensions to the basic machine-level (object-level, disassembly-level) debug capabilities of IDF, originally packaged as a separate module.

LE/370 See *IBM Language Environment for z/OS and z/VM*.

library routine
A routine maintained in a program library.

line mode
An interface mode for use with a non-programmable terminal which uses a single command line to accept Interactive Debug Facility commands.

link-edit
To create a loadable computer program using a linkage editor.

linkage editor
A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single relocatable load module.

listing A printout that lists the source language statements of a program with all preprocessor statements, includes, and macros expanded.

load module
A program in a form suitable for loading into main storage for execution.

LUname
Defines the VTAM logical unit name of the terminal used by IDF in z/VSE.

LSM See *Language Support Module*.

LSM Information window
The IDF windows which contains information generated by IDF Language Support commands.

M

MainFrame Interface (MFI)
This refers to the use of a nonprogrammable terminal such as an IBM 3270.

Maximize
The action used to remove a window entry from the Minimized Windows Viewer, and restore it to its previous position on the display.

MFI See *MainFrame Interface*.

Minimize
The action used to remove an IDF window from the display and replaces it with an entry in the Minimized Windows Viewer.

Minimized Windows Viewer
An IDF window which contains entries which represent minimized IDF windows.

module
The "package" which contains the executable code and data for a program. This may be in the form of a file, or an area of storage.

multitasking
A mode of operation that enables the concurrent performance, or interleaved execution, of two or more tasks.

N

name pattern
A set of criteria used to display a list of variable names.

O

Old Registers window
The IDF window which displays the PSW and registers contents from the previous point when IDF had control. By default, the previous General Purpose Registers (GPRs) and Floating Point Registers (FPRs) are shown, but the previous Access Registers (ARs) or Control Registers (CRs) may be shown instead.

Options window
The IDF window which contains current values of the IDF options and settings.

P

panel In the MFI Interactive Debug Facility, an area of the screen used to display a specific type of information.

parameter
Data received by a program or procedure from another. Contrast with *argument*.

partitioned data set (PDS)
A data set in direct-access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

path point
A point in the program where control is about to be transferred to another location or a point in the program where control has just been given.

PDS See *partitioned data set*.

PER Program Event Recording

postmortem debug
To detect, diagnose, and eliminate errors in programs after the program has ABENDed. This is typically performed offline, using a dump file or data set.

prefix area
The eight columns to the left of the program source or listing containing line numbers. In the Interactive Debug Facility, statement breakpoints can be set in the prefix area.

primary entry point
See *entry point*.

Processor Status Word
This register describes the current processor execution state. Various fields contain the current:
- addressing mode
- execution address
- condition-code setting
- storage access key
- problem or supervisor state indicator
- other state indicators

procedure
In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call.

A set of related control statements. For example, a z/VM exec, or a z/OS CLIST.

profile
A group of customizable settings that govern how the user's session appears and operates.

Profile
A choice that allows you to change some characteristics of the working environment, such as the pace of statement execution in the Interactive Debug Facility.

program
A sequence of instructions suitable for processing by a computer. Processing can include the use of an assembler, a

compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it.

program part
A compile unit associated with an application program. All program parts known to the Interactive Debug Facility are displayed in the MAP window.

program unit
See *compile unit*.

programmable workstation (PWS)
A workstation that has some degree of processing capability and that allows you to change its functions (for example, a small computer such as an IBM Personal System/2* (PS/2*) as a terminal device along with appropriate 3270 emulation software).

PSW See *Processor Status Word*.

PWS See *programmable workstation*.

Q

qualification
A method used to specify to what procedure or load module a particular variable name, function name, label, or statement id belongs. The SET QUALIFY command changes the current implicit qualification.

R

record A group of related data, words, or fields treated as a unit, such as one name, address, and telephone number.

record format
The definition of how data is structured in the records contained in a file. The definition includes record name, field names, and field descriptions, such as length and data type. The record formats used in a file are contained in the file description.

reference
In programming languages, a language construct designating a declared language object.

A subset of an expression that resolves to an area of storage; that is, a possible target of an assignment statement. It can be any of the following: a variable, an array or array element, or a structure or structure element. Any of the above can be pointer-qualified where applicable.

Register
Specialized high speed storage location or control facility. See *General Purpose Register*, *Floating Point Register*, *Access Register*, and *Control Register*.

Register Alteration Stop
This is a special breakpoint available when IDF on CMS has exploitation of the virtual machine Program Event Recorder (PER) mode enabled. IDF will receive control when any of the specified registers is altered.

Register Stop
See *Register Alteration Stop*.

RegStop
See *Register Alteration Stop*.

run To cause a program, utility, or other machine function to execute.

An action that causes a program to begin execution and continue until a run-time exception occurs. If a run-time exception occurs, you can use debug windows to interact with the Interactive Debug Facility.

run time
Any instant at which a program is being executed.

run-time environment
A set of resources that are used to support the execution of a program.

run unit
A group of one or more object programs that are run together.

S

SBCS See *single-byte character set*.

semantic error
An error in the implementation of a program's specifications. The semantics of a program refer to the meaning of a program. Unlike syntax errors, semantic errors (since they are deviations from a program's specifications) can be detected only at run time.

sequence number
A number that identifies the records within a z/VM file, or a z/OS member or partitioned data set.

session
The events that take place between the time you start an application and the time you exit the application.

shortcut keys
A key or combination of keys that starts an application-defined function. The IDF user interface term for accelerator keys or *hot keys*.

single-byte character set (SBCS)
A character set in which each character is represented by a one-byte code.

Skipped Subroutines window
The IDF window which lists the currently active "Skipped Subroutine" breakpoints. These are set by the SKIPSTEP command.

source The statements in a file that make up a program.

In Interactive Debug Facility, the representation of a program's source statements displayed in the Disassembly window.

static In programming languages, pertaining to properties that can be established before execution of a program; for example, the length of a fixed length variable is static. Contrast with *dynamic*.

status area
An area appended to a window that shows the keyboard shift state for DBCS on a DBCS-enabled workstation.

step One statement in a computer routine.

To cause a computer to execute one or more statements.

storage
A unit into which recorded text can be entered, in which it can be retained, and from which it can be retrieved.

The action of placing data into a storage device.

A storage device.

Storage Alteration Stop
This is a special breakpoint available when IDF on CMS has exploitation of the virtual machine Program Event Recorder (PER) mode enabled. IDF will receive control when storage within the specified address ranges is altered.

subroutine
A sequenced set of instructions or statements that can be used in one or more computer programs at one or more points in a computer program.

suffix area
A variable-sized column to the right of the program source or listing statements, containing frequency counts for the first statement or verb on each line. In the Interactive Debug Facility, the MFI optionally displays the suffix area in the Disassembly window. See also *prefix area*.

synchronous
Pertaining to two or more processes that depend on the occurrence of specific events. Contrast with *batch, interactive*.

syntactic analysis
An analysis of a program done by a compiler to determine the structure of the program and the construction of its source statements to determine whether it is valid for a given programming language. See also *syntax error*.

syntax The rules governing the structure of a programming language and the construction of a statement in a programming language.

syntax error
Any deviation from the grammar (rules) of a given programming language appearing when a compiler performs a syntactic analysis of a source program. See also *syntactic analysis*.

T

Target Status window
The IDF window which lists information about the program modules which are currently defined to IDF.

token A character string in a specific format that has some defined significance in a programming language.

triglyph
A group of three characters which, taken together, are equivalent to a single special character.

type style
A form of highlighting of characters and symbols within a font set. For example, bold, italic, strikeout, or underscore. See also *display attribute*.

U

utility A computer program in general support of computer processes; for example, a diagnostic program, a trace program, or a sort program.

V

variable
A name used to represent a data item whose value can be changed while the program is running. Contrast with *constant*.

W

window
A division of the display screen in which one of several IDF commands can concurrently display information.

windowing
Dividing a display screen into distinct areas in which different display images can be viewed at the same time.

workstation
One or more programmable or nonprogrammable devices, normally connected to a host or a network, at which you can run applications. See also *programmable workstation*.

Index

GC26-8709-08