English

FUJITSU Software BS2000

AID V3.4B

Debugging of FORTRAN Programs

User Guide

Edition June 2018

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Documentation creation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

Contents

1 Preface

AID, the Advanced Interactive Debugger in BS2000, provides users with a powerful debugging tool. Thanks to AID, error diagnostics, debugging and short-term error recovery of all programs generated in BS2000 are considerably more rapid and more straightforward than other approaches, such as inserting debugging aid statements into a program, for example. AID is permanently available and is extremely adaptable to the particular programming language. Any program debugged using AID does not have to be recompiled but can be used in a production run immediately. The range of functions of AID and its debugging language (using AID commands) are primarily tailored to interactive applications. AID can, however, also be used in batch mode. AID provides the user with a wide range of options for monitoring and controlling execution, effecting output and modification of memory contents; furthermore it provides help information on program execution as well as information on the AID program itself.

With AID, the user can debug both on the symbolic level of the relevant programming language as well as on machine code level. If LSD records are generated, data, statement labels and program sections can be addressed for debugging purposes by using names the user has assigned in the course of programming. Statements can be addressed via the numbers or names created by the compiler. If no LSD records have been generated for a program or module, the user can address data and statements by using virtual addresses, CSECT names and keywords.
The BS2000 commands occurring in the AID documentation are described in the EXPERT form of the SDF (System Dialog Facility) format. SDF is the dialog interface to BS2000. The SDF command language supersedes the previous (ISP) command language.

1.1 Objectives and target groups of the AID documentation

AID is targeted to all software developers working in BS2000 with the programming languages COBOL, FORTRAN, C, PL/I or ASSEMBH or those who wish to debug or correct programs on machine code level.

1.2 Structure of the AID documentation

AID documentation is comprised of the AID Core Manual, the language-specific manuals for symbolic debugging, and the manual for debugging on machine code level. All the information the user requires for debugging can be found by referring to the manual for the particular language required and the core manual. The manual for debugging on machine code level can either be used as a substitute for or as a supplement to any of the language-specific manuals.

AID Core Manual [1]

This basic reference manual contains an overview of AID and a description of the contents and operands which are common to all the programming languages. As part of the overview, the BS2000 environment is described; basic concepts are explained and the AID repertoire of commands is presented. The other chapters describe prerequisites for debugging; command input; the operands *subcmd*, *compl-memref* and *medium-a-quantity*; AID literals and keywords. The manual also includes the BS2000 commands not permitted in command sequences and a comparison of AID and IDA.

AID - Debugging on Machine Code Level [2]

AID - Debugging of COBOL Programs [3]

AID - Debugging of FORTRAN Programs

AID - Debugging under POSIX[4]

AID - Debugging of ASSEMBH Programs [5]

AID - Debugging of C Programs

The manuals for the specific languages and the manual for debugging on machine code level list the commands in alphabetical order. All simple memory references are contained there.

In the language-specific manuals, the description of the operands is tailored to fit the programming language in question. A prerequisite for this is that the user knows the particular language scope and operation of the relevant compiler.

The manual for debugging on machine code level can be used for programs for which no LSD records exist or for which the information from symbolic testing does not suffice for error diagnosis. Debugging on machine code level means the user can issue AID commands regardless of the language in which the program was written.

Readme file

The functional changes to the current product version and revisions to this manual are described in the product-specific Readme file.

Readme files are available to you online in addition to the product manuals under the various products at *http://manuals.ts.fujitsu.com*. You will also find the Readme files on the Softbook DVD.

Information under BS2000

When a Readme file exists for a product version, you will find the following file on the BS2000 system:

`SYSRME.<product>.<version>.<lang>`

This file contains brief information on the Readme file in English or German (<lang>=E/D). You can view this information on screen using the `/SHOW-FILE` command or an editor. The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the user ID under which the product's files are stored.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at *http://manuals.ts.fujitsu.com*.

1.3 Changes since the last edition of this manual

AID V3.4B30 offers the following new functions compared to version V3.4B10:

- Extension of the%AID command: new *LEV* operand. This operand can expand the output of the AID command %SDUMP %NEST by the levels within the call hierarchy.
- New qualification *NESTLEV* in the %DISPLAY, %MOVE, %SDUMP and %SET commands designated to qualify all instances of recursive data.
- Enhancement of the %FIND command that enables searching the *find area* for characters from a coded character set (CCS) supported by XHCS.

1.4 Notational conventions

italics Within the text, operands are shown in *italic lowercase*.

This symbol marks points in the text to which particular attention should be paid.

2 Prerequisites for symbolic debugging

To perform symbolic debugging, AID requires a "List for Symbolic Debugging" (LSD) in which the symbolic names defined in a program are listed. This LSD information is generated by the compiler, and taken over by the linkage editor, the dynamic linking loader, or the static linkage editor and starter. The control statements needed for compiling are described briefly below.
In the AID Core Manual, chapter 3, you will find general information on LSD records, and on linking, loading, and starting.

2.1 Compilation

As of V2.1A, the FOR1 compiler can be controlled in two ways:

– via SDF options or

– via COMOPT statements.

Whether the compiler is to generate LSD records can thus be specified as described below, depending on the control option selected.

SDF control

```
                                                                    ⎡ NO  ⎤
/START-FOR1-COMPILER .....,TEST-SUPPORT = PARAMETER (TOOL-SUPPORT = ⎨     ⎬)
                                                                    ⎣ AID ⎦
```

NO
No LSD records are generated. AID can only be used to debug the program on machine code level.

AID The compiler generates LSD records. The program can be symbolically debugged using AID.

COMOPT control

```
/START-PROGRAM $FOR1
*...
                        ⎧  NO   ⎫
*COMOPT SYMTEST = ⎨  MAP  ⎬
                        ⎩  ALL  ⎭
```

NO No LSD records are generated.

MAP No LSD records are generated, but call hierarchies can be traced.

ALL The compiler generates LSD records. The program can be symbolically debugged using AID.

As of FOR1 V2.1A, LSD records may be generated for optimized programs as well. However, the optimized program then no longer matches the compiler listing because:

– the sequence of statements might be changed

– a statement might be split up

– statements may be dropped

– with %DISPLAY, the *previous* value of a variable is generally output, since storage of a value in a variable rarely occurs immediately after the corresponding assignment statement is processed.

To debug a highly-optimized FORTRAN program (SDF option `OPTIMIZATION=HIGH` or COMOPT statements `OPTIMIZE={3|4}` and `PROCEDURE-OPTIMIZATION=SPECIAL`) by means of AID, the user can have a decompiler listing generated (see "FOR1 User Guide" [8]). Such a listing facilitates debugging of an optimized program. Note, however, that the %JUMP command cannot be used for the debugging of optimized programs.

Generation of both shareable code and LSD information in the same compilation run is not possible. If both options are specified, FOR1 resets the SDF option `SHAREABLE-CODE=YES` or the COMOPT statement `OBJECT=(SHARE)` and issues an error message.

A complete representation of the operands which control compilation is provided in the FOR1 User Guide [8].

Example

```
/START-FOR1-COMPILER SOURCE = SOURCE.TEST,
         TEST-SUPPORT = PARAMETER (TOOL-SUPPORT = AID),
                     MODULE-LIBRARY = PROGRAMLIB
```

An object module is to be generated with LSD records when compiling the source program SOURCE.TEST. The object module is written directly to the PLAM library PROGRAMLIB.
If COMOPT control is used, the example reads as follows:

```
/DELETE-SYSTEM-FILE FILE-NAME = OMF
/START-PROGRAM $FOR1
*COMOPT SOURCE=SOURCE.TEST
*COMOPT SYMTEST=ALL
*COMOPT MODULE-LIBRARY=PROGRAMLIB
*END
```

2.2 Linking, loading and starting

For information on this topic, please refer to the AID Core Manual, section "Symbolic Debugging".

During the debugging phase, loading of the program via the LOAD-PROGRAM command is recommended so that the user can enter the AID commands required for debugging. START-PROGRAM is used to link, load and start the program. Both SDF commands are described in the AID Core Manual, chapter 3; they are same for all programming languages.

The FORTRAN program can also be linked, loaded and started by using the START-FOR1-PROGRAM command, in which case the SDF operand TESTOPT controls the way the LSD records are handled.
As of FOR1 V2.2A, START-FOR1-PROGRAM just loads the program without starting it if the SDF options TESTOPT=AID and RUNOPT=NO are specified. AID commands can thus be entered for debugging before the program is started using %RESUME.

If a FOR1 version < 2.2A is used, the FORTRAN statement PAUSE should be inserted at a suitable point in the program. This will interrupt the program run so that AID commands can be entered.

```
/START-FOR1-PROGRAM .....,TESTOPT = { NONE }
                                    { AID  }
```

NONE The program is loaded without LSD records.

If the LSD records with the object module are in a PLAM library, AID can dynamically load them whenever required. To do so, the library must be specified using %SYMLIB.

AID The program is loaded with LSD records.

The linkage editor does not check whether the processed object module actually includes LSD records.

Examples

1.
```
/START-FOR1-PROGRAM FROM-FILE = *MODULE (LIBRARY = *OMF),
                           TESTOPT = AID, RUNOPT = NO
```

   The dynamic linking loader links the program from the temporary object module file and loads it with the associated LSD records (as of FOR1 V2.2A).

2.
```
/START-FOR1-PROGRAM FROM-FILE = *PHASE (LIBRARY = PROGRAMLIB,
             ELEMENT = ROOTMOD),
                           TESTOPT = NONE
```

   From the PLAM library PROGRAMLIB the linked program ROOTMOD is loaded without LSD records and started.

3 FORTRAN-specific addressing

This chapter describes the memory references used for symbolic debugging of FORTRAN programs. For a general description of addressing methods please refer to the AID Core Manual, chapter 6.

Qualifications

Qualifications must always be specified in the order described below. They are delimited by periods. Likewise a period must be inserted between the final qualification and the following operand.

E={VM|Dn}

The base qualification specifies whether the AID work area is to be located in a loaded program (E=VM) or in a dump file (E=Dn). The base qualification is used in the same way both for symbolic debugging and for machine-oriented debugging, as described in the AID Core Manual, chapter 6, and under the %BASE command. A base qualification can be immediately followed by a data name, statement name, source reference or complex memory reference.

PROG=program-name

In FORTRAN, the user can employ the PROG qualification as the area qualification, where *program-name* designates a program unit from a FORTRAN program. *program-name* consists of up to 7 characters specified as part of the PROGRAM, SUBROUTINE or FUNCTION statement in the source program.
Operands specifying an address area (%CONTROL, %TRACE) or a name range (%SDUMP) can end with the PROG qualification. The address range or name range then encompasses the entire program unit.

PROG=program-name•program-name

If the name of a program unit is repeated directly after a PROG qualification, the user is thus designating the address of the first program unit statement which can be executed.
This specification can be used in %DISASSEMBLE and %INSERT.

NESTLEV=level-number

The NESTLEV qualification defines a level number.

Like the qualification S=*srcname*.PROC=*function*, the qualification NESTLEV=*level-number* is designed to manipulate data names that users declare in the source units. The environment qualification E={VM|Dn} is the only one NESTLEV=*level-number* can be combined with.

The qualification NESTLEV accepts a level number, in other words, a reference to the current call hierarchy. Based on this reference, AID identifies a complete list of available data names defined at the specified level.

Normally, you have to display and analyze the call hierarchy before using the NESTLEV qualification. The following AID commands output the current call hierarchy augmented with the levels:

```
%AID LEV=ON
%SDUMP %NEST
```

The NESTLEV qualification can be used in the commands %DISPLAY, %MOVE, %SDUMP and %SET. In these commands, the qualification NESTLEV=*level-number* can equally (with the same result) replace the qualification S=*srcname*.PROC=*function*, if *level-number* is correct.

For an example for the usage of the NESTLEVqualification, see AID Core Manual, section “Area qualifications“[1].

Memory references

Memory references may include all data names and statement labels from the program which are contained in the LSD records, as well as the statement numbers generated by the compiler, and may be subjected to all the operations described in the AID Core Manual, chapter 6.
In all operands in which *compl-memref* is possible, the user can arbitrarily switch between the memory references as described in this manual and those for debugging on machine code level (see [2]).

dataname

stands for all the names of constants, variables and arrays defined in the source program. *arrayname* must be indexed if an array element is to be addressed. As many indexes are required as need to be specified for access in a FORTRAN statement. Multiple indexes have to be separated by a comma.
index can be specified as follows:

```
{n                    }
{dataname             }
{arithmetic expression}
```

n

is an integer with a value $-2^{31} \leq n \leq 2^{31}-1$.

dataname

designates a numeric variable of type 'integer' which must be located in the same program unit as *arrayname*.

arithmetic-expression

AID calculates the value for *index*. Valid entries are the arithmetic operators (+,–,/,*) and the above-listed operands *n* and *dataname*.

dataname can be specified in all commands for output and modification of information (%DISPLAY, %MOVE, %SDUMP, %SET) and in the %FIND command (search for a string).

L'n'

is a statement name, designating the address of the first executable FORTRAN statement following a statement label.

n is a statement label (maximum of 5 digits) of the source program; the label is assigned by the programmer. Leading zeros are not to be specified.
L'n' may be specified in all operands either designating an address in the executable part of the program (%DISASSEMBLE, %FIND, %INSERT, %JUMP) or serving for the output and modification of memory locations (%DISPLAY, %MOVE, %SET).

S'n'

is a source reference designating the address of an executable FORTRAN statement.

n is the number of a source program statement; it is assigned by the compiler and can be found in column STMT of the compiler listing.

S'n' may be specified in all operands either designating an area (%CONTROLn, %TRACE) or address (%DISASSEMBLE, %FIND, %INSERT) in the executable part of the program or serving for the output and modification of memory locations (%DISPLAY, %MOVE, %SET).

index

Range of indexes

You can specify a range of indexes:

```
array (index{,...})
```

index1 : *index2*
This designates the range between *index1* and *index2*. Both must lie within the index limits, and *index1* must be less than or equal to *index2*.

*
This designates the entire index range of the dimension. In the case of single dimensional arrays, this is equivalent to using the array name without indexing.

You can only use range specification in the %DISPLAY command. Array names with range specifications must not be used in address calculations. Modifications of type or length are not permitted.

Examples

```
%D array (*,3)
```

In a two dimensional array, all elements belonging to the first dimension and whose second dimension index is 3 are output.

```
%D array (1 : 3,*,5 : 15)
```

The following elements are output from a three dimensional array:
- The index of the first dimension is 1, 2 or 3,
- The index of the second dimension ranges from the lower index limit to the upper index limit
- The index of the third dimension ranges from 5 to 15.

4 Metasyntax

The metasyntax shown below is the notational convention used to represent commands. The symbols used and their meanings are as follows:

UPPERCASE LETTERS

Mandatory string which the user must employ to select a particular function.

lowercase letters

String identifying a variable, in the place of which the user can insert any of the permissible operand values.

```
⌈ alternative ⌉
{    ...      }
⌊ alternative ⌋

{ alternative | ... | alternative }
```

Alternatives; one of these alternatives must be picked. The two formats have the same meaning.

```
[optional]
```

Specifications enclosed in square brackets indicate optional entries.

In the case of AID command names, only the entire part in square brackets can be omitted; any other abbreviations cause a syntactical error.

`[...]`

Reproducibility of an optional syntactical unit. If a delimiter, e.g. a comma, must be inserted before any repeated unit, it is shown before the periods.

`{...}`

Reproducibility of a syntactical unit which must be specified at least once. If a delimiter, e.g. a comma, must be inserted, it is shown before the periods.

Underscoring

Underscoring designates the default value which AID inserts if the user does not specify a value for the operand.

•

A bullet (period in bold print) delimits qualifications, stands for a *prequalification* (see also the %QUALIFY statement), is the operator for a byte offset or part of the execution counter or subcommand name. The bullet is entered from the keyboard using the key for a normal period. It is actually a normal period, but here it is shown in bold to make it stand out better.

All operands in the continuous text of the manual appear in *italics*.

5 AID commands

In the sections in this chapter, the AID commands are arranged in alphabetical order and described in detail. The table below provides an overview of the commands available:

Command	Function
%AID	Change global settings
%BASE	Define global base qualification
%CONTINUE	Start or continue program, continue any active %TRACE
%CONTROLn	Monitor selected statements
%DISASSEMBLE	Retranslate memory contents into symbolic Assembler notation
%DISPLAY	Output the contents of data elements, their addresses and lengths, system information and literals
%DUMPFILE	Open or close dump files and assign link names
%FIND	Search for a character string
%HELP	Help function for AID commands and AID messages
%INSERT	Set test points for monitoring program execution
%JUMP	Declare a continuation address
%MOVE	Change the contents of data elements without type checking and without converting numerical values
%ON	Monitor selected events
%OUT	Specify output media and additional information for output commands
%OUTFILE	Open or close AID output files and assign link names
%QUALIFY	Define a prequalification
%REMOVE	Delete monitoring declarations
%RESUME	Start or continue program, terminate any active %TRACE
%SDUMP	Symbolic dump; output data elements or the program names of the current call hierarchy

Command	Function
%SET	Change the contents of data elements with type checking and with conversion of numerical values
%STOP	Halt program and switch to command mode
%SYMLIB	Specify libraries for dynamic loading of LSD records
%TITLE	Define page headers and activate pagination for output to SYSLST
%TRACE	Start or continue program with tracing

%AID

The %AID command can be used to declare global settings or to revoke the settings valid up until then.

– By means of the *CHECK* operand you define whether an update dialog is to be initiated prior to execution of the %MOVE or %SET commands.

– By means of the *REP* operand you define whether memory updates of a %MOVE command are to be stored as REPs.

– By means of the *SYMCHARS* operand you define whether AID is to interpret a "-" in program, data and statement names as a hyphen or as a minus sign. If "-" should always be interpreted as a minus sign (in accordance with the FORTRAN conventions), SYMCHARS=NOSTD must be specified.

– By means of the *OV* operand you direct AID to take the overlay structure of a program into account.

– By means of the *LOW* operand you direct AID to convert lowercase letters of character literals and names to uppercase, or to interpret them as lowercase. The default value is OFF.

– By means of the *DELIM* operand you define the delimiters for AID output of alphanumeric data. The vertical bar is the default delimiter.

– By means of the *LANG* operand you define whether AID is to output %HELP information in English or German.

– With the operand *LEV*, you can activate the output of levels within the call hierarchy produced by the %SDUMP %NEST AID command.

```
Command       Operand
              ┌ CHECK [= {ALL|NO}]          ┐
              │ REP [= {YES|NO}]            │
              │ SYMCHARS [= {STD|NOSTD}]    │
              │ OV [= {YES|NO}]             │
%AID          { LOW [= {ON|OFF}]            }
              │                  ⎧C'x'|'x'C|'x'⎫ │
              │ DELIM [= ⎨                     ⎬] │
              │                  ⎩'|'          ⎭ │
              │                    ‾           │
              │ LANG [= {D | E}]            │
              │ LEV [= {ON|OFF}]            │
              └                             ┘
```

Declarations made using %AID remain valid until superseded by a new %AID command or until /LOGOFF.

%AID can only be issued as an individual command, it must never be part of a command sequence or a subcommand.

The %AID command does not alter the program state.

```
---------
| CHECK |
---------
```

ALL

Prior to execution of a %MOVE or %SET command, AID conducts the following update dialog:

```
OLD CONTENT:
AAAAAAAA
NEW CONTENT:
BBBBBBBB
%  IDA0129 CHANGE? (Y = YES; N = NO) ?

N

I342 NOTHING CHANGED
```

If **Y** is entered, the old contents of the array are overwritten and no further message is issued.
In procedures in batch mode, AID is not able to conduct a dialog and always assumes **Y**.

NO

%MOVE and %SET commands are executed without an update dialog.

If the *CHECK* operand is entered without specification of a value, AID assumes the default value (NO).

```
_______
| REP |
_______
```

YES

In the event of a memory update caused by a %MOVE command, LMS UPDR records (REPs) are created. If an object structure list is not available, AID does not create any REPs and issues an error message to this effect.

AID stores the corrections with the requisite LMS UPDR statements in a file with the link name F6, from which they can be fetched as a complete package. Care should therefore be taken that no other outputs are written to the file with link name F6. If no file with link name F6 is registered (cf. %OUTFILE), the REP record is stored in the file created by AID (AID.OUTFILE.F6).

User-specific REP files must be created with FCBTYPE=SAM. REP files created by AID are likewise defined with FCBTYPE=SAM, RECFORM=V and OPEN=EXTEND.
The file remains open until it is closed via %OUTFILE or until /LOGOFF.

NO

No REPs are generated.

If the *REP* operand is entered without a value specification, AID inserts the default (NO). The *REP* operand of the %MOVE command can supersede the declaration made with %AID, but only for this particular %MOVE command. For subsequent %MOVE commands without a REP operand, the declaration made with the %AID command is valid again.

```
------------
| SYMCHARS |
------------
```

STD

A hyphen "-" is interpreted as an alphanumeric character and can, as such, be used in program, data and statement names. A hyphen is only interpreted as a minus sign if a blank precedes it.

NOSTD

A hyphen "-" is always interpreted as a minus sign and cannot be used as a part of names.

If the *SYMCHARS* operand is entered without a value specification, AID inserts the default value (STD).
SYMCHARS=NOSTD must be set if the "-" character, in accordance with the FORTRAN conventions, is always to be interpreted as a minus sign.

```
------
| OV |
------
```

YES

Mandatory specification if the user is debugging a program with an overlay structure. AID checks each time whether the program unit which has been addressed originates from a dynamically loaded segment.

NO

AID assumes that the program to be debugged has been linked without an overlay structure. AID does not check whether the CSECT information or LSD records belong to the program unit which has been addressed.

If the *OV* operand is entered without a value specification, AID assumes the default (NO).

```
-------
| LOW |
-------
```

ON

Lowercase letters in character literals and in program, data and statement names are not converted to uppercase.

OFF

All lowercase letters from user entries are converted to uppercase.

ALL Entry of all BLS names is case sensitive.

In addition, upper and lower case entries in character literals and in program, data and instruction names are retained, as when %AID LOW=ON is specified.

The following BLS names are used by AID:
- Context names of the CTX qualification
- Load unit names of the L qualification
- Link module names of the O qualification
- CSECT names of the C qualification
- COMMON names of the COM qualification
- Names of compilation units of the S qualification

If no *LOW* operand has been entered in a debugging session, OFF applies.

If the *LOW* operand is input without a value specification, AID assumes the default (ON). In this case LOW=OFF must be entered if conversion to uppercase is to be reactivated.

```
_________
| DELIM |
_________
```

C'x'|'x'C|'x'

With this operand the user defines a character as the left-hand and right-hand delimiter for AID output of symbolic data of type 'character' (%DISPLAY and %SDUMP commands).

```
|
-
```

The standard delimiter is the vertical bar.

If the *DELIM* operand is entered without value specification, AID inserts the default value (|).

```
________
| LANG |
________
```

D

AID outputs information requested with %HELP in German.

E

AID outputs information requested with %HELP in English.

If the *LANG* operand is entered without a value specification, AID inserts the default (D).

```
--------
| LEV  |
--------
```

ON Enable level output.

When level output is enabled, %SDUMP %NEST additionally outputs two kinds of levels for each procedure (function or block in C/C++) in the call hierarchy:

– A general level (counter) with a backward numeration, i.e. from the current procedure to the main procedure. This level number is applicable in the new qualification *NESTLEV*.

– A recursive level (RLEV) or an individual counter for each procedure with a backward numeration starting from 0. The recursive level serves as informative element.

OFF Disable level output.

Examples

In the SYMCHAR program, the contents of array element IFELD(L+M) are to be replaced by the contents of IFELD(L-M) with the aid of the %SET command.

Source listing of the SYMCHAR program:

```
DO/IF SEG STMT  I/H LINE          SOURCE-TEXT
-------------------------------------------------------------------------------
  1/1    1         1 |        PROGRAM SYMCHAR
    1    2         2 |        PARAMETER (B=3, C=5)
    1    3         3 |        DIMENSION IFELD(B+C)
    1    4         4 |        INTEGER IFELD /1,2,3,4,5,6,7,8/
    1    5         5 |        L=5
    1    6         6 |        M=3
    1    7         7 |        WRITE *,IFELD
    1    8         8 |        WRITE *,IFELD(L+M)
    1    9         9 |        WRITE *, ' SYMCHAR TERMINATED!'
    1   10        10 |        END
```

1. Since %AID SYMCHARS = STD has been set, L-M is interpreted as the name of the element; AID issues the message "L-M NOT FOUND".

```
/LOAD-PROG FROM-FILE=*MOD(LIB=*OMF),TEST-OPT=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'SYMCHAR' LOADED
/%IN S'9' <%D IFELD;%SET IFELD(L-M) INTO IFELD(L+M);%D IFELD>
```

```
/%R
BS2000  F O R 1 : FORTRAN PROGRAM "SYMCHAR"
STARTED ON 91-02-18 AT 12:04:11
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
8
** ITN: #00000047'***TSN*8438************************************'
SCR_REF:     9 SOURCE: SYMCHAR     PROC: SYMCHAR *********************
IFELD( 1: 8)
( 1)           1  ( 2)             2  ( 3)           3  ( 4)      4
( 5)           5  ( 6)             6  ( 7)           7  ( 8)      8
I375  SYMBOL   L-M NOT FOUND

STOPPED AT SCR_REF: 9, SOURCE: SYMCHAR , PROC: SYMCHAR
```

2. After %AID SYMCHARS = NOSTD has been entered, the hyphen in L-M is interpreted as minus sign. AID then executes the %SET command correctly.

```
/LOAD-PROG FROM-FILE=*MOD(LIB=*OMF),TEST-OPT=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'SYMCHAR' LOADED
/%AID SYMCHARS=NOSTD
/%IN S'9' <%D IFELD;%SET IFELD(L-M) INTO IFELD(L+M);%D IFELD>
/%R
BS2000  F O R 1 : FORTRAN PROGRAM "SYMCHAR"
STARTED ON 91-02-18 AT 12:05:37
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
8
** ITN: #00000047'***TSN*8438************************************'
SCR_REF:     9 SOURCE: SYMCHAR     PROC: SYMCHAR *********************
IFELD( 1: 8)
( 1)           1  ( 2)             2  ( 3)           3  ( 4)      4
( 5)           5  ( 6)             6  ( 7)           7  ( 8)      8
IFELD( 1: 8)
( 1)           1  ( 2)             2  ( 3)           3  ( 4)      4
( 5)           5  ( 6)             6  ( 7)           7  ( 8)      2
 SYMCHAR TERMINATED !
BS2000  F O R 1 : FORTRAN PROGRAM "SYMCHAR " ENDED PROPERLY AT 12:05:41
CPU - TIME USED :       0.2124 SECONDS
ELAPSED TIME    :       4.6430 SECONDS
```

%BASE

The %BASE command is used to specify the base qualification. All subsequently entered memory references without their own base qualification assume the value declared via %BASE. The %BASE command also defines the AID work area.

– With the *base* operand the user designates either the virtual memory area of the program which has been loaded or a dump in a dump file.

Command	Operand
%BASE	[base]

With the %BASE command the user also defines the location of the AID work area. When debugging FORTRAN programs, the AID work area corresponds to the area which the current program unit occupies in virtual memory or in a dump file. If the user fails to enter a %BASE command during a debugging session or enters %BASE without any operands, the base qualification E=VM applies by default and the AID work area corresponds to that program unit in virtual memory which contains the current interrupt point (AID standard work area).

A %BASE command is valid until the next %BASE command is given, until /LOGOFF or until the dump file declared as the base qualification is closed (see %DUMPFILE).

Memory references within a subcommand are supplemented with current qualifications during input, i.e. a %BASE command has no effect on subcommands specified previously.

%BASE can only be entered as an individual command, it must never be part of a command sequence or subcommand.

%BASE does not alter the program state.

base

defines the base qualification. All subsequently entered memory references without a separate base qualification assume the value declared with the %BASE command.

```
base-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     ⎧ VM ⎫
E =  ⎨    ⎬
     ⎩ Dn ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

E=VM

The virtual memory area of the program which has been loaded is declared as the base qualification. VM is the default value.

E=Dn

A dump in a dump file with the link name Dn is declared as the base qualification. n is a number with a value $0 \leq n \leq 7$.

Before declaring a dump file as the base qualification, the user must assign the corresponding dump file a link name and open it, using the %DUMPFILE command.

%CONTINUE

The %CONTINUE command is used to start the program which has been loaded or to continue it at the interrupt point or at the location specified by %JUMP.
As opposed to %RESUME, an interrupted but still active %TRACE command is not terminated by %CONTINUE, rather it is continued depending on the declarations which have been made.

Command	Operand
%CONT[INUE]	

In the following cases a %TRACE command is regarded as interrupted and is resumed by any %CONTINUE command:

1. When a subcommand has been executed as the result of a monitoring condition from a %CONTROLn, %INSERT or %ON command having been satisfied, and the subcommand contained a %STOP.
2. When an %INSERT command terminates with a program interrupt because the *control* operand is K or S.
3. When the K2 key has been pressed.
4. When the program was halted by the FORTRAN statement PAUSE.

A subcommand containing only the %CONTINUE command merely increments the execution counter.

If the %CONTINUE command is given in a command sequence or subcommand, any subsequent commands are not executed.

%CONTINUE alters the program state.

%CONTROLn

By means of the %CONTROLn command you may declare up to seven monitoring functions one after the other, which then go into effect simultaneously. The seven commands are %CONTROL1 through %CONTROL7.

– By means of the *criterion* operand you may select different types of FORTRAN statements. If a statement of the selected type is waiting to be executed, AID interrupts the program and processes *subcmd*.

– By means of the *control-area* operand you may define the program area in which *criterion* is to be taken into consideration.

– By means of the *subcmd* operand you declare a command or a command sequence and possibly a condition (see AID Core Manual, "Subcommands"). *subcmd* is executed if *criterion* is satisfied and any specified condition has been met.

```
------------------------------------------------------------------------------
Command           Operand
------------------------------------------------------------------------------

%C[ONTROL]n       [criterion][,...]  [IN control-area] [<subcmd>]

------------------------------------------------------------------------------
```

Several %CONTROLn commands with different numbers do not affect one another. Therefore you may activate several commands with the same *criterion* for different areas, or with different *criteria* for the same area. If several %CONTROLn commands occur in one statement, the associated subcommands are executed successively, starting with %C1 and working through %C7.

The individual value of an operand for %CONTROLn is valid until overwritten by a new specification in a later %CONTROLn command with the same number, until the %CONTROLn command is deleted or until the end of the program.
A %REMOVE command can be used to delete either an individual %CONTROLn or all active %CONTROLn declarations.

%CONTROLn can only be used in a loaded program, i.e. the base qualification E=VM must have been set via %BASE or must be specified explicitly.

%CONTROLn does not alter the program state.

```
-------------
| criterion |
-------------
```

is the keyword defining the type of the FORTRAN statements prior to whose execution AID is to process *subcmd*.

You can specify several keywords at the same time, which are then valid at the same time. Any two keywords must be separated by a comma.
If no *criterion* is declared, AID works with the default value %STMT, unless a *criterion* declared in an earlier %CONTROLn command is still valid.

criterion	*subcmd* is processed prior to:
%STMT	Every executable FORTRAN statement
%ASSGN	Assignment statements
%CALL	SUBROUTINE calls (CALL statements)
%COND	IF(...) THEN, ELSE IF(...) THEN, ELSE and IF(...) statements
%GOTO	GOTO statements
%IO	Input/output statements
%LAB	Every statement with a label
%PROC	STOP, END, RETURN statements as well as the first executable statement following SUBROUTINE or FUNCTION

```
-----------------
| control-area  |
-----------------
```

specifies the program area in which the monitoring function will be valid. If the user exits from the specified program, the monitoring function becomes inactive until another statement within the program area to be monitored is executed. The default value is the current program area.

A *control-area* definition is valid until the next %CONTROLn command with the same number is issued with a new definition, until the corresponding %REMOVE %CONTROLn command is issued, or until the end of the program is reached. %CONTROLn without a

control-area operand of its own results in a valid area definition being taken over. To be valid, such a *control-area* operand must be defined in a %CONTROLn command with the same number, and the current interrupt point must be within this area. If no valid area definition exists, the *control-area* comprises the current program unit by default.

```
control-area OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - -

               ⌈PROG=program-name                    ⌉
IN  [•][E=VM•] {                                     }
               ⌊[PROG=program-name•]( S'n' : S'n' ) ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

E=VM

As *control-area* can only be in the virtual memory of the loaded program, *E=VM* need only be specified if a dump file has been declared as the current base qualification (see %BASE command).

PROG=program-name

program-name is the name of a program unit and may consist of up to 7 characters.

This program unit must have been loaded at the time the %CONTROL command is entered.

A PROG qualification is required only if a load module was created from several source modules and the %CONTROLn command does not refer to the current program unit, or if a previously valid *control-area* declaration is to be overwritten.

If *control-area* ends with a PROG qualification, the area covers the entire program unit specified.

(S'n' : S'n')

The *control-area* is defined by specifying a start address and an end address. The start and end addresses must be within the same program unit, where the following applies:
start address ≤ end address
where *n* is the number of a statement (see STMT column in compiler listing).

If *control-area* is to comprise only one statement, the start and end address must be the same.

subcmd is processed whenever a statement that satisfies the *criterion* is awaiting execution in the *control-area*. *subcmd* is processed before execution of the *criterion* statement.

If *subcmd* is not specified, AID inserts <%STOP> for %CONTROLn.

For a complete description of *subcmd* see the AID Core Manual, chapter 5.

```
subcmd-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                 ⌈AID-command    ⌉
<[subcmdname:] [(condition):] [{                 } {;...}]>
                                 ⌊BS2000-command ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A subcommand may contain a name, a condition and a command part.

Every subcommand has its own execution counter. The command portion can consist of an individual command or a command sequence; it may contain AID commands, BS2000 commands and comments.

If the subcommand consists of a name or a condition, but the command part is missing, AID merely increments the execution counter when a statement of type *criterion* has been reached.

In addition to the commands which are not permitted in any subcommand, the *subcmd* of a %CONTROLn must not contain the AID commands %CONTROLn, %INSERT, %JUMP or %ON.

The commands in *subcmd* are executed consecutively, after which the program is continued. The commands for runtime control also immediately change the program state when they are part of a subcommand. They abort *subcmd* and start the program (%CONTINUE, %RESUME, %TRACE) or halt it (%STOP). In practice, they are only useful as the last

command in *subcmd*, since any subsequent commands of the *subcmd* will not be executed. Likewise, deletion of the current subcommand via %REMOVE is only expedient as the last command in *subcmd*.

Examples

1. `%CONTROL1 %CALL, %PROCIN(S'123':S'250') <%DISPLAYCOUNTER;%STOP>`

   %C1 %CALL,%PROC IN(S'123':S'250') <%D COUNTER;%STOP>

   The two AID commands differ only in their notation.

   The first example is written in full and contains a varying number of blanks at the permissible positions; the second example is abbreviated.

   The %CONTROL1 command is valid for the criteria %CALL and %PROC and is to be effective between statements 123 and 250 (inclusive).

   If one of the FORTRAN statements identified via the criteria %CALL and %PROC occurs during program execution, the %DISPLAY command from *subcmd* is executed for the variable COUNTER. Then the program run is interrupted by means of %STOP, and AID or BS2000 commands may be entered.

2. `%CONTROL1 %CALL <%DISPLAY 'CALL' T=MAX; %STOP>`

   Prior to the execution of every CALL statement, AID executes the %DISPLAY command from *subcmd* and then interrupts the program by executing the %STOP command.

3. `%CONTROL2 %IO <%SDUMP %NEST P=MAX; %REMOVE C1>`

   Prior to the execution of an IO statement, AID outputs the current call hierarchy to the system file SYSLST and then executes the %REMOVE command, which deletes the declarations of %CONTROL1. Program execution continues.

4. `%C3 %PROC <%STOP>`

   The %C3 command declares that AID is to execute a %STOP command before a SUBROUTINE, FUNCTION, RETURN, STOP or END statement is executed.

5. `%C4 %PROC <(SLF LE 10): %D IFELD(1)>`

   %C4 is used to specify that AID is to output the first array element of IFELD before any SUBROUTINE, FUNCTION, RETURN, STOP or END statement is executed, provided that the SLF value is less than or equal to 10.

%DISASSEMBLE

%DISASSEMBLE enables memory contents to be "retranslated" into symbolic Assembler notation and displayed accordingly.
Output takes place to SYSOUT, SYSLST, or to a cataloged file (see the %OUT command).

The *number* operand enables you to determine how many instructions are to be disassembled and output.

– The *start* operand enables you to determine the address where AID is to begin disassembling.

```
------------------------------------------------------------------------------
Command           Operand
------------------------------------------------------------------------------

⎧ %DISASSEMBLE ⎫
⎨              ⎬  [number]    [FROM start]
⎩ %DA          ⎭

------------------------------------------------------------------------------
```

Disassembly of the memory contents starts with the first byte. For memory contents which cannot be interpreted as an instruction, an output line is generated which contains the hexadecimal representation of the memory contents and the message INVALID OPCODE. The search for a valid operation code then proceeds in steps of 2 bytes each.

%DISASSEMBLE without a *start* operand permits the user to continue a previously issued %DISASSEMBLE command until the test object is switched or a new operand value is defined by means of a BS2000 or AID command (/LOAD-PROGRAM, /EXEC-PROGRAM, %BASE). AID continues disassembly at the memory address following the address last processed by the previous %DISASSEMBLE command. If *number* is not specified either, AID generates the same number of output lines as declared before.

If the user has not entered a %DISASSEMBLE command during a test session or has changed the test object and does not specify current values for one or both operands in the %DISASSEMBLE command, AID works with the default value 10 for *number* and V'0' for *start*.

The %OUT command can be used to control how processed memory information is to be represented and to which output medium it is to be transferred. The format of the output lines is explained after the description of the *start* operand.

The %DISASSEMBLE command does not alter the program state.

```
----------
| number |
----------
```

Specifies how many Assembler commands are to be output.

If no value has been specified for *number* and no value from a previous %DISASSEMBLE command applies, AID inserts the default value (10).

number

is an integer with the value:

$1 \leq number \leq 2^{31}-1$

```
---------
| start |
---------
```

Defines the address at which disassembly of memory contents into Assembler commands is to begin. If the *start* value is not specified, AID assumes the default value V'0' for the first %DISASSEMBLE; on every further %DISASSEMBLE, AID continues after the Assembler command last disassembled.

```
start OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                ⎧program-name      ⎫
                ⎪L'n'              ⎪
FROM  [•][qua•] ⎨S'n'              ⎬
                ⎪                  ⎪
                ⎩compl-memref      ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined by a previous %QUALIFY command. Consecutive qualifications must be delimited by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

Specify a qualification only if the *start* value is not within the current AID work area.

E={VM | Dn}

Only required if the current base qualification is not to apply for *start* (see %BASE command).

PROG=program-name

Only required if *start* is not located in the current program unit (see chapter 3).

program-name

This specification is only possible following an explicit PROG qualification:
`PROG=program-name•program-name`
By repeating the *program-name* entry, *start* is set to the initial address of the designated program unit.

L'n'

is a statement name designating the address of the first executable FORTRAN statement following a statement label.

n is a statement label of up to 5 digits. Leading zeros must not be specified.

S'n'

is a source reference and designates the address of an executable FORTRAN statement.

n is a statement number; see STMT column of the compiler listing.

compl-memref

designates an address which is to be computed. It should be the start address of a machine instruction, otherwise the disassembly obtained will be meaningless. *compl-memref* may contain the following operations (see AID Core Manual, chapter 6):

- byte offset (•)
- indirect addressing (->)
- type modification (%A)
- length modification (%Ln)
- address selection (%@(...))

A statement name *L'n'* or a source reference *S'n'* can be used within *compl-memref*, but only in connection with the pointer operator, e.g. L'n' ->.4
A type modification makes sense only if the contents of a data element can be used as an address or if the address is taken from a register,
e.g. %1G.2 %AL2 ->

Output of the %DISASSEMBLE log

By default, the %DISASSEMBLE log is output with additional information to SYSOUT (T=MAX). With %OUT the user can select the output media and specify whether or not additional information is to be output by AID.

AID does not take into account XMAX and XFLAT modes for outputting the %DISASSEMBLE log. Instead, it generates the default value (T=MAX).

The following is contained in a %DA output line if the default value T=MAX is set:

– CSECT-relative memory address

– memory contents retranslated into symbolic Assembler notation, displacements being represented as hexadecimal numbers (as opposed to Assembler format)

– for memory contents which do not begin with a valid operation code: Assembler statement DC in hexadecimal format and with a length of 2 bytes, followed by the note INVALID OPCODE

– hexadecimal representation of the memory contents (machine code).

Example of line format with T=MAX

The statement number in the %DISASSEMBLE command refers to the sample application in section 6.1.

```
/LOAD-PROG FROM-FILE=*MOD(LIB=*OMF),TEST-OPT=AID
 %  BLS0001 DLL VER 823
 %  BLS0517 MODULE 'B1' LOADED
/%DISASSEMBLE 10 FROM PROG=SORT.S'22'
 SORT+90          L     R15,1B0(R0,R13)           58 F0 D1B0
 SORT+94          A     R15,B0(R0,R12)            5A F0 C0B0
 SORT+98          ST    R15,1B0(R0,R13)           50 F0 D1B0
 SORT+9C          BC    B'1111',76(R0,R11)        47 F0 B076
 SORT+A0          DC   X'0000'  INVALID OPCODE    00 00
 SORT+A2          BCR   B'1100',R8                07 C8
 SORT+A4          DC   X'0000'  INVALID OPCODE    00 00
 SORT+A6          ISK   R3,R8                     09 38
 SORT+A8          L     R15,1B4(R0,R13)           58 F0 D1B4
 SORT+AC          MH    R15,EE(R0,R12)            4C F0 C0EE
```

The %OUT operand value T=MIN causes AID to create shortened output lines in which the CSECT-relative address is replaced by the virtual address and the hexadecimal respresentation of the memory contents is omitted.

Example of line format with T=MIN

```
/%OUT %DA T=MIN
/%DISASSEMBLE 10 FROM PROG=SORT.S'22'
 000005F8  L     R15,1B0(R0,R13)
 000005FC  A     R15,B0(R0,R12)
 00000600  ST    R15,1B0(R0,R13)
 00000604  BC    B'1111',76(R0,R11)
 00000608  DC   X'0000'  INVALID OPCODE
 0000060A  BCR   B'1100',R8
 0000060C  DC   X'0000'  INVALID OPCODE
 0000060E  ISK   R3,R8
 00000610  L     R15,1B4(R0,R13)
 00000614  MH    R15,EE(R0,R12)
```

Examples

1. `%DISASSEMBLE FROM PROG=EXAMPLE.L'22'`

   This command initiates disassembly of 10 instructions (default), starting with the address of the first executable statement following statement label 22 in program unit EXAMPLE.

2. `%DA 2 FROM E=D1.PROG=EXAMPLE.EXAMPLE`

   Starting with the start address of program unit EXAMPLE in the dump file with link name D1, two instructions are to be disassembled.

3. `%DA FROM S'67'`

   Since no value is specified for *number*, AID either inserts the default value (in the case of the first %DISASSEMBLE for this program) or takes the value from the previous %DISASSEMBLE. Disassembly starts with the first instruction generated for the statement with the number 67.

%DISPLAY

The %DISPLAY command is used to output memory contents, addresses, lengths, system information and AID literals and to control feed to SYSLST. AID edits the data in accordance with the definition in the source program, unless you select another type of output by means of type modification.

Output is via SYSOUT, SYSLST or to a cataloged file.

– By means of the *data* operand you specify data elements, their addresses or lengths, statements, registers, execution counters of subcommands, and system information. Here you also define AID literals or you control feed to SYSLST.

– By means of the *medium-a-quantity* operand you specify the output medium AID uses and whether or not additional information is to be output. This operand disables a declaration made via the %OUT command, but only for the current %DISPLAY command.

Command	Operand
%D[ISPLAY]	data {,...} [medium-a-quantity][,...]

A %DISPLAY command which does not have a qualification for *data* addresses *data* of the current program unit.

If you do specify a qualification, you can access *data* in a dump file or in any other program unit which has been loaded, provided this program unit is part of the current call hierarchy.

If the *medium-a-quantity* operand is not specified, AID outputs the data in accordance with the declarations in the %OUT command or, by default, to SYSOUT, together with additional information (cf. AID Core Manual, chapter 7).

Immediate entry of the command right after loading the program is not recommended, as data and statements cannot be addressed without an explicit qualification until the program encounters the first executable statement. The first executable statement is reached by entering the command sequence:
%INSERT PROG=program-name.program-name
%RESUME

%DISPLAY %SORTEDMAP will produce a list of all program CSECTs, sorted by names and addresses.

In addition to the operand values described here, you can also use the operand values described for debugging on machine code level (see [2]).

This command can be used both in the loaded program and in a dump file.

%DISPLAY does not alter the program state.

```
_________
| data  |
_________
```

This operand defines the information AID is to output. You may output the contents, address and length of variables, arrays or array elements, the contents and length of constants, as well as the addresses of statements. The contents of registers and execution counters as well as the system information relevant to your program can be addressed via keywords. AID literals can be defined to improve the readability of debugging logs, and feed to SYSLST can be controlled for the same purpose.

AID edits data elements in accordance with the definitions in the source program, provided that you have not defined another type of output using a type modification (see also AID Core Manual, section 6.8). If the contents do not match the defined storage type, output is rejected and an error message is issued. Nevertheless the contents of the data element can be viewed, for instance by employing the type modification %X to edit the contents in hexadecimal form.

If you enter more than one *data* operand in a %DISPLAY command, you may switch from one operand to another between the symbolic entries described here and the non-symbolic entries described in the manual for debugging on machine code level (see [2]). Symbolic and machine-oriented specifications can also be combined within a complex memory reference.

For names which are not contained in the LSD records, AID issues an error message; the other *data* of the same command will be processed in the normal way.

```
data-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

⎛           ⎛dataname          ⎞       ⎞
|           |L'n'               |       |
| [•][qua•] ⎨S'n'               ⎬       |
|           |keyword            |       |
|           ⎝compl-memref       ⎠       |
|                                       |
| ⎛%@⎞            ⎛dataname          ⎞  |
⎨ ⎨  ⎬ ([•][qua•] ⎨                  ⎬ ⎬
| ⎝%L⎠            ⎝compl-memref      ⎠  |
|                                       |
| %L=(expression)                       |
|                                       |
| AID-literal                           |
|                                       |
⎝ feed-control                          ⎠

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

A qualification need only be specified for memory objects not located within the current AID work area.

E={VM | Dn}

Specified only if the current base qualification (see %BASE) is not to apply for a data/statement name, source reference or keyword.

PROG=program-name

Specified only if a data/statement name or source reference not contained in the current program unit is to be addressed (see chapter 3).

NESTLEV= level-number

level-number A level number in the current call hierarchy
level-number has to be followed by *dataname*.
The syntax indicates that the %DISPLAY command is to output the data item *dataname* defined at the level *level-number* of the current call hierarchy.

dataname

specifies the name of a constant, variable, array or array element as defined in the source program.

dataname is an alphanumeric string with up to 15 characters.

If *dataname* is the name of an array, it must be indexed as in a FORTRAN statement if an array element is to be addressed. If you specify the name of an array without the index list, all elements of the array are output.

array-name (index1[, index2][, ...])

index specifies the position within an array. The number of indexes required for access is the same as that which must be specified in a FORTRAN statement. When multiple indexes are specified, a comma must be used as a separator. *index* may be specified as follows:

```
{n                    }
{dataname             }
{arithmetic-expression}
```

The following FORTRAN data definitions are output differently, as shown below:

– INTEGER*8 as REAL*8

– REAL*16 as REAL*8

– COMPLEX*32 as COMPLEX*16

If *dataname* is a data element of type COMPLEX, the real and imaginary portions of the complex number will be output. You can also limit output either to the real portion or to the imaginary portion as follows:

%D *dataname* ._REAL	outputs the real part
%D *dataname* ._IMAG	outputs the imaginary part.

L'n'

Specifies a statement name and designates the address of the first executable FORTRAN statement after a statement label.

n is a statement label (maximum of 5 digits). Leading zeros must not be specified. If *L'n'* is entered without a pointer operator, the corresponding address is output in hexadecimal representation. With a pointer operator, i.e. with %DISPLAY L'n'->, AID outputs 4 bytes of the machine code contained at the relevant address.

S'n'

Specifies a source reference and designates the address of an executable FORTRAN statement.

n is the number of a statement (see STMT column of compiler listing).
If *S'n'* is entered without a pointer operator, the corresponding address is output in hexadecimal representation. With a pointer operator, i.e. with %DISPLAY S'n'->, AID outputs 4 bytes of the machine code contained at the relevant address.

keyword

Here you may specify all the keywords for program registers, AID registers, system tables and the one for the execution counter or the symbolic localization information (see AID Core Manual, chapter 9).
keyword can only be preceded by a base qualification.

%n	General register, $0 \leq n \leq 15$
%nD|E	Floating-point register, n = 0,2,4,6
%nQ	Floating-point register, n = 0,4
%nG	AID general register, $0 \leq n \leq 15$

%nDG	AID floating-point register, n = 0,2,4,6
%nAR	Access register, 0 ≤ n ≤ 15
%MR	All 16 general registers in tabular form
%FR	All 4 floating-point registers with double precision edited in tabular form
%AR	All 16 access register in tabular form
%PC	Program counter
%CC	Condition code
%PCB	Process control block
%PCBLST	List of all process control blocks
%SORTEDMAP	List of all CSECTs of the user program (sorted by name and address)
%IFR	Interrupt flag register
%IMR	Interrupt mask register
%ISR	Interrupt status register
%PM	Program mask
%AMODE	Addressing mode (24- or 31-bit addresses)
%ASC	ASC mode (with respect to AR mode: X'00' = off; X'01' = on)
%AUD1	P1 audit table, plus the SAVE table (if any)
%LINK	Name of the last dynamically loaded segment (see %ON, event %LPOV)
%HLLOC(speicherref)	Localization information on the symbolic level for a memory reference in the executable part of the program (high-level location)
%LOC(speicherref)	Localization information on machine code level for a memory reference in the executable part of the program (low-level location)
%DS[(ALET/SPID-qua)]	Information about SPIDs and/or ALETs of the active data spaces
%**.**subkdoname	Execution counter
%**.**	Execution counter of the currently active subcommando

compl-memref

The following operations may occur in a *compl-memref* (see AID Core Manual, chapter 6):

– byte offset (•)

– indirect addressing (->)

– type modification (%T(dataname), %X, %C, %E, %P, %D, %F, %A)

– length modification (%L(...), %L=(expression), %Ln)

– address selection (%@(...))

Following byte offset or indirect addressing, AID outputs the memory contents at the calculated address in dump format with a length of 4 (%XL4, default).
Using the type modification, *data* may be edited in any form, provided its contents match the specified storage type. %X can always be used to output a data element in hexadecimal format, regardless of its contents and definition in the source program.
With the length modification you can define the output length yourself, e.g. if you wish to output only parts of a data element or display a data element using the length of another data element.

%@(...)

With the address selector you can output the address of a data element or of *compl-memref*.
The address selector cannot be used for symbolic constants (including the statement names *L'n'* and the source references *S'n'*).

%L(...)

With the length selector you can output the length of a data element.

Example

```
%DISPLAY %L(AARRAY)
```

The length of AARRAY will be output.

%L=(expression)

With the length function you can have a value calculated (see AID Core Manual, sections 6.9 and 6.10).
expression you can link the contents of memory references and constants of type 'integer' with arithmetic operators (+, -, *, /).

Example

```
%DISPLAY %L=(AARRAY)
```

If AARRAY is of type 'integer', its contents will be output. Otherwise AID issues an error message.

AID-literal

All AID literals described in the AID Core Manual, chapter 8, may be specified:

```
{C'x...x' | 'x...x'C | 'x...x'}      Character literal
{X'f...f' | 'f...f'X}                Hexadecimal literal
{B'b...b' | 'b...b'B}                Binary literal
[{±}]n                               Integer
#f...f'Hexadecimalnumber'
[{±}]n.m                             Fixed-point number
[{±}]mantissaE[{±}]exponent          Floating-point number
```

feed-control

For output to SYSLST, print editing can be controlled by the following two keywords, where:

%NP results in a page feed

%NL[(n)] results in a line feed by *n* blank lines.

$1 \le n \le 255$. The default for *n* is 1.

```
---------------------
| medium-a-quantity |
---------------------
```

Defines the medium or media via which output is to take place, and whether additional information is to be output by AID. If this operand is omitted and no declaration has been made using the %OUT command, AID uses the presetting T = MAX.

```
medium-a-quantity-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - -

,{I  }   {MIN }
 {H  } = {MAX }
 {Fn }   {XMAX}
 {P  }   {XFLAT}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-a-quantity is described in full detail in the AID Core Manual, chapter 7.

T Terminal output

H Hardcopy output

Fn File output

P Output to SYSLST

MAX	Output with additional information
MIN	Output without additional information
XMAX	The following applies for AID V3.4B: In the %DISPLAY command the operand value XMAX is not taken into account, as a result of which the behavior is identical to the default value MAX.
XFLAT	The following applies for AID V3.4B: In the %DISPLAY command the operand value XFLAT is not taken into account, as a result of which the behavior is identical to the default value MAX.

Examples

1. `%DISPLAY E=D1.PROG=EXAMPLE.INTVAR,'CONTENTS OF DUMP'`

   Here the contents of a dump are evaluated.

```
** D1: DUMP.EXAMPLE
************************************************************
INTVAR         =                        -89
CONTENTS OF DUMP
```

2. `%DISPLAY %L=(S'13'-S'12')`

   AID outputs the length of the machine code sequence generated for statement 12.

```
   +52
```

3. `%BASE`

   `%DISPLAY L'200'`

   %BASE switches back to the AID standard work area. AID then outputs the address of the first executable statement following label 200 as a hexadecimal number.

```
** ITN:
#00010053'***TSN:6567*********************************************'
SRC_REF:    26 SOURCE: B1        PROC: B1
************************************
200             = 0000051C
```

4. `%DISPLAY L'200'->`

   AID outputs 4 bytes of the machine code generated at the address of label 200. The pointer operator switches to the machine code level, which causes AID to display an additional header.

```
CURRENT PC: 0000CEFA    CSECT: IF@STOP
***************************************
V'0000051C' = B1       + #0000051C''
0000051C (0000051C) 9500D17C                                  n.J@
```

5. `%DISPLAY %HLLOC(L'200'->)`

   AID outputs symbolic localization information for label 200.

```
V'0000051C' = SMOD     :  B1
      PROC     :  B1
      SRC-REF  :  82
      LABEL    :  200
```

6. `%DISPLAY %LOC(L'200'->)`

   AID outputs localization information on machine code level for label 200.

```
V'0000051C' = PROG :  QSORT
      LMOD :  %ROOT
      SMOD :  B1
      OMOD :  B1
      CSECT :  B1       (00000000) + 0000051C
```

7. `%DISPLAY CHARARRAY`

   The array CHARARRAY comprises 26 array elements and is defined in the program as follows:
   `CHARACTER CHARARRAY (26)/'A','B','C','D',...,'X','Y','Z'/`
   As no index list is specified in the %DISPLAY command, AID outputs all the elements of the array:

```
** ITN:
#00010053'***TSN:6567*********************************************'
SRC_REF:    66 SOURCE: EXAMPLE   PROC: EXAMPLE
*********************************
CHARARRAY(1:26)
(  1) |A|  (  2) |B|  (  3) |C|  (  4) |D|  (  5) |E|  (  6) |F|  (  7)
|G|
(  8) |H|  (  9) |I|  ( 10) |J|  ( 11) |K|  ( 12) |L|  ( 13) |M|  ( 14)
|N|
( 15) |O|  ( 16) |P|  ( 17) |Q|  ( 18) |R|  ( 19) |S|  ( 20) |T|  ( 21)
|U|
( 22) |V|  ( 23) |W|  ( 24) |X|  ( 25) |Y|  ( 26) |Z|
```

8. The FORTRAN program OUTPUT displays all data types which can be defined in FOR1.

```
 DO/IF SEG  STMT  I/H LINE      SOURCE-TEXT
-------------------------------------------------------------------------------
   1/1     1         1 |    PROGRAM OUTPUT
                     2 | *
     1     2         3 |    INTEGER * 1       INT1    /-12/
     1     3         4 |    INTEGER * 2       INT2    /234/
     1     4         5 |    INTEGER * 4       INT4    /997/
     1     5         6 |    INTEGER * 8       INT8    /757/
                     7 | *
     1     6         8 |    REAL    *   4      REAL4   /123.456/
     1     7         9 |    REAL    *   8      REAL8   /128.996/
     1     8        10 |    REAL    *  16      REAL16  /-987.772/
                    11 | *
     1     9        12 |    COMPLEX *   8     CPLX8    /(2.5,4.7)/
     1    10        13 |    COMPLEX *  16     CPLX16  /(1.6,9.6)/
     1    11        14 |    COMPLEX *  32     CPLX32  /(3.7,8.9)/
                    15 | *
     1    12        16 |    CHARACTER *   5      CHARC(3)   /'AAAAA','BBBBB','CCCCC'/
     1    13        17 |    CHARACTER * (45,V)    CHARV      /'44778'/
                    18 | *
     1    14        19 |    LOGICAL * 1   LOG1     /.TRUE./
     1    15        20 |    LOGICAL * 4   LOG4     /.FALSE./
                    21 | *
     1    16        22 |    CHARACTER CTEXT*30 /'CORRESPONDING FORTRAN OUTPUT:'/
                    23 | *
                    24 | *
                    25 | *
     1    17        26 |    WRITE(2,*) CTEXT
     1    18        27 |    WRITE(2,*) INT1
     1    19        28 |    WRITE(2,*) INT2
     1    20        29 |    WRITE(2,*) INT4
     1    21        30 |    WRITE(2,*) INT8
     1    22        31 |    WRITE(2,*)
                    32 | *
     1    23        33 |    WRITE(2,*) CTEXT
```

```
     1    24        34 |    WRITE(2,*) REAL4
     1    25        35 |    WRITE(2,*) REAL8
     1    26        36 |    WRITE(2,*) REAL16
     1    27        37 |    WRITE(2,*)
                    38 | *
     1    28        39 |    WRITE(2,*) CTEXT
     1    29        40 |    WRITE(2,*) CPLX8
     1    30        41 |    WRITE(2,*) CPLX16
     1    31        42 |    WRITE(2,*) CPLX32
     1    32        43 |    WRITE(2,*) REAL(CPLX8)
     1    33        44 |    WRITE(2,*) IMAG(CPLX8)
     1    34        45 |    WRITE(2,*)
                    46 | *
     1    35        47 |    WRITE(2,*) CTEXT
     1    36        48 |    WRITE(2,*) CHARC
     1    37        49 |    WRITE(2,*) CHARV
     1    38        50 |    WRITE(2,*)
                    51 | *
     1    39        52 |    WRITE(2,*) CTEXT
     1    40        53 |    WRITE(2,*) LOG1
     1    41        54 |    WRITE(2,*) LOG4
     1    42        55 |    WRITE(2,*)
                    56 | *
     1    43        57 |    END
```

```
/START-FOR1-COMPILER SOURCE=Q.OUTPUT,OPTIMIZATION=NO,-
/TEST-SUPPORT=PARAMETER(TOOL-SUPPORT=AID),-
/LISTING=PARAMETER(OUTPUT=LF.OUTPUT)
%  BLS0500 PROGRAM 'FOR1', VERSION '2.1A00' OF '91-03-06' LOADED.
 FOR1:   V2.1A00 READY, GIVE COMPILER OPTION
 FOR1: LIST FILE REPLACED = LF.OUTPUT
 FOR1: NO ERRORS DURING COMPILATION OF P.U. OUTPUT
 END OF  F O R 1  COMPILATION;  CPU TIME USED: 1.675 SEC.
/LOAD-PROG FROM-FILE=*MOD(LIB=*OMF),TEST-OPT=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'OUTPUT' LOADED
/%INSERT S'17' <%DISPLAY C'AID OUTPUT I*1, I*2, I*4, I*8',INT1, -
/INT2, INT4, INT8>
/%INSERT S'23' <%DISPLAY C'AID OUTPUT R*4, R*8, R*16',REAL4,REAL8,REAL16>
/%INSERT S'28' <%DISPLAY C'AID OUTPUT C*8, C*16, C*32, REAL(C*8),
IMAG(C*8)',-
/CPLX8, CPLX16, CPLX32, CPLX8._REAL, CPLX8._IMAG>
/%INSERT S'35' <%DISPLAY C'AID OUTPUT CHAR*L, CHAR*(MAXL,V)',CHARC, CHARV>
/%INSERT S'39' <%DISPLAY C'AID OUTPUT LOG*1, LOG*4', LOG1, LOG4>
/%RESUME
 BS2000  F O R 1 : FORTRAN PROGRAM "OUTPUT"
 STARTED ON 91-06-28 AT 11:33:32
```

Program OUTPUT was compiled without errors and was linked and loaded with LSD records. Test points were defined via %INSERTs so that each %DISPLAY command is followed by the corresponding FOR1 output. The text lines "AID OUTPUT" and "CORRESPONDING FORTRAN OUTPUT" are printed in bold for greater clarity.

```
AID OUTPUT I*1, I*2, I*4, I*8
** ITN: #000000DF'***TSN:1627************************************************'
SRC REF:    17 SOURCE: OUTPUT   PROC: OUTPUT
**********************************
INT1             =    -12
INT2             =         234
INT4             =         997
INT8             = +.7570000000000000 E+003
CORRESPONDING FORTRAN OUTPUT:
-12
234
997
757
```

First, all integer variables are output. Unlike FOR1, AID outputs data elements of type INTEGER*8 as REAL*8 elements.

```
AID OUTPUT R*4, R*8, R*16
SRC REF:    23 SOURCE: OUTPUT    PROC: OUTPUT
**********************************
REAL4            = +.1234559 E+003
REAL8            = +.1289960021972656 E+003
REAL16           = -.9877719726562500 E+003
CORRESPONDING FORTRAN OUTPUT:
0.12345599E+03
0.12899600219726625E+03
-0.98777197265625000000000000000000E+03
```

The data elements of type REAL*4, REAL*8 and REAL*16 are output. REAL*16 variables are output as REAL*8 variables by AID.

```
AID OUTPUT C*8, C*16, C*32, REAL(C*8), IMAG(C*8)
** ITN: #000000DF'***TSN:1627************************************************'
SRC REF:    28 SOURCE: OUTPUT   PROC: OUTPUT
**********************************
CPLX8
 REAL            = +.2500000 E+001
 IMAG            = +.4699999 E+001
CPLX16
```

```
 REAL           = +.1600000381469726 E+001
 IMAG           = +.9600000381469726 E+001
CPLX32
 REAL           = +.3699999809265136 E+001
 IMAG           = +.8899999618530273 E+001
CPLX8. REAL     = +.2500000 E+001
CPLX8. IMAG     = +.4699999 E+001
CORRESPONDING FORTRAN OUTPUT:
(0.2500000E+01,0.46999998E+01)
(0.16000003814697265E+01,0.96000003814697265E+01)
(0.3699999809265136718750000000000000E+01,0.8899999618530273437500000000000000E
+01)
0.25000000E+01
0.46999998E+01
```

The real and imaginary parts of complex numbers are always output separately by AID. This corresponds to the bracketed representation of complex numbers in FOR1.
Like FOR1, AID permits the real and imaginary parts to be addressed individually: *dataname*._REAL designates the real part, *dataname*._IMAG the imaginary part of a complex variable.

```
AID OUTPUT CHAR*L, CHAR*(MAXL,V)
SRC REF:    35 SOURCE: OUTPUT   PROC: OUTPUT
**********************************
CHARC( 1: 3)
( 1) |AAAAA|   ( 2) |BBBBB|   ( 3) |CCCCC|
CHARV           = |44778|
CORRESPONDING FORTRAN OUTPUT:
AAAAABBBBBCCCCC
44778

AID OUTPUT LOG*1, LOG*4
SRC REF:    39 SOURCE: OUTPUT   PROC: OUTPUT
**********************************
LOG1            = %TRUE
LOG4            = %FALSE
CORRESPONDING FORTRAN OUTPUT:
T
F
```

Arrays of type ’character’ are likewise output according to array elements by AID. Comparison of the various data types is concluded with logical variables as edited in AID and FOR1 respectively.

%DUMPFILE

With %DUMPFILE you assign a dump file to a link name and cause AID to open or close this file.

– With *link* you select the link name for the dump file to be opened or closed.

– With *file* you designate the dump file to be opened.

```
------------------------------------------------------------------------------
Command           Operand
------------------------------------------------------------------------------

⌈%DUMPFILE⌉
{          }       [link  [=datei]]
⌊%DF      ⌋

------------------------------------------------------------------------------
```

If you omit the *file* operand AID will close the file assigned to the specified link name.

With a %DUMPFILE command without operands, you cause AID to close all open dump files. If the AID work area was, up until this point, contained in a dump file now closed, the AID standard work area then reapplies (see also %BASE command).

%DUMPFILE may only be specified as an individual command, i.e. it may not be

part of a command sequence and may not be included in a subcommand.

%DUMPFILE does not alter the program state.

```
--------
| link |
--------
```

Designates one of the AID link names for input files and has the format Dn, where n is a number with a value $0 \leq n \leq 7$.

```
--------
| file |
--------
```

Specifies the fully-qualified file name under which the dump file AID is to open is cataloged. If this operand is omitted, the dump file with the link name *link* is closed.
An open dump file must first be closed with a separate %DUMPFILE command before another file can be assigned the same link name.

Examples

1. `%DUMPFILED3=DUMP.1234.00001`

   The file DUMP.1234.00001 with link name D3 is opened.

2. `%DF D3`

   The file assigned to link name D3 is closed.

3. `%DF`

   All open dump files are closed.

%FIND

With %FIND you can search for a literal in a data element or in the executable part of a program, and output hits to the terminal (via SYSOUT). In addition, the address of the hit and the continuation address are stored in AID registers %0G and %1G. %FIND can be used to search both virtual memory and a dump file.

– *search-criterion* is the character literal or hexadecimal literal to be searched.

– With *find-area* you specify which data element or which section of the executable part of the program AID is to search for *search-criterion*. AID can search the virtual address space of the task as well as dump files. If the *find-area* value is omitted, AID searches the entire memory area in accordance with the base qualification currently set (see %BASE).

– With *alignment* you specify whether the search for *search-criterion* is to be effected at a doubleword, word, halfword or byte boundary. When a value for *alignment* is not given, searching takes place at the byte boundary.

– With *ALL* you specify that the search is not to be terminated after output of the first hit, rather the entire *find-area* is to be searched and all hits are to be output. The search can only be aborted by pressing the K2 key.

Command	Operands
%F[IND]	[[ALL] search-criterion [IN find-area] [alignment]]

If the *ALL* operand is omitted from a %FIND command, the user may continue after the address of the last hit and up to the end of the *find-area* by specifying a new %FIND command without any operand values.

A %FIND command with a separate *search-criterion* and without any further operands takes declarations for *find-area* and *alignment* from a preceding %FIND command. If there has not been any preceding %FIND command, AID inserts the default values.

Output of hits is always in dump format (hexadecimal and character representation) with a length of 12 bytes to the terminal (SYSOUT). In addition to the hit itself, its address and (insofar as possible) the name of the program unit in which the hit was found, and the relative address of the hit with respect to the beginning of the program unit, are output.

In the event of a hit, the hit address is stored in AID register %0G and the continuation address (hit address + search string length) in AID register %1G. With the *ALL* specification, the address of the last hit is stored in %OG and the continuation address of the last hit is stored in %1G. If the *search-criterion* has not been found, AID sets %0G to -1; %1G remains

unchanged.
The two register contents permit you to use the %FIND command in procedures as well as in subcommands and to further process the results.

The %FIND command does not alter the program state.

```
--------------------
| search-criterion |
--------------------
```

is a character literal or hexadecimal literal. *search-criterion* may contain wildcard symbols. These symbols are always hits. They are represented by '%'.

```
search-criterion-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - -

{C'x...x' | 'x...x'C | 'x...x' }
{X'f...f' | 'f...f'X           }

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

`{C'x...x'|'x...x'C|'x...x'}`

Character literal with a maximum length of 80 characters. Lowercase letters can only be located as character literals after specifying %AID LOW[=ON].

x can be any representable character, in particular the wildcard symbol '%', which always represents a hit. The character '%' itself cannot be located when it is in this form, since C'%' in a character literal must always result in a hit. For this reason it must be represented as the hexadecimal literal X'6C'.

Please note that in order to properly locate character data, the CCS of *find-area* has to agree with the CCS of the input media (SYSCMD). Be sure to specify the CCS of *find-area* before looking for some character data in *find-area*:

`%AID CCS=` *`CCS-name`*

A complete list of *CCS-name* supported by XHCS and the current CCS of SYSCMD can be displayed with the following AID command:

`%SHOW %CCSN`

The CCS of SYSCMD can be changed with the following SDF command:

`MODIFY-TERMINAL-OPTION CODED-CHARACTER-SET= {`*`EBCDIC-CCS-name`* | *`UTFE`*`}`

The current CCS of *find-area* can be displayed with the following AID command:

`%SHOW %AID`

Be aware that since V3.4B11 the %DISPLAY command refers to the CCS value of %AID as to the default (implicit) CCS of character data to be displayed:

```
%D char-data ['CCS-name']
```

See the section “Character literal” in the AID Core Manual [1] for an example on how to search for character literals in different coded character sets.

```
{X'f...f' | 'f...f'X}
```

Hexadecimal literal with a maximum length of 80 hexadecimal digits or 40 characters. A literal with an odd number of digits is padded with X’0’ on the right.

f can assume any value between 0 and F, as well as the wildcard symbol X’%’. The wildcard symbol represents a hit for every hexadecimal digit between 0 and F.

```
--------------
| find-area  |
--------------
```

defines the memory area to be searched for *search-criterion*. *find-area* can be a data element or a section of the executable part of the loaded program or of a dump file. *find-area* must not exceed 65535 bytes in length.

If no *find-area* has been specified, AID inserts the default value %CLASS6 (see AID Core Manual), i.e. %FIND starts its search at address V'0'.

```
find-area-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

              ⎧dataname          ⎫
              ⎪L'n'->            ⎪
IN [•][qua•] ⎨S'n'->            ⎬
              ⎪                  ⎪
              ⎩compl-memref      ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

A qualification need be specified only if *find-area* is not within the current AID work area.

E={VM | Dn}

Need only be specified if the current base qualification is not to apply for *find-area* (see also %BASE command).

PROG=program-name

Need only be specified if *find-area* is not within the current program unit (see chapter 3).

dataname

is the name of a variable, array or array element defined in the source program.

dataname is an alphanumeric string with up to 15 characters.

If *dataname* is the name of an array, it must be indexed as in a FORTRAN statement if an array element is to be addressed. If you specify the name of an array without an index list, the entire array is searched for *search-criterion*.

array-name (index1[, index2][, ...])

index specifies the position within an array. The number of indexes required for access is the same as that necessary in a comparable FORTRAN statement. Multiple indexes must be separated by commas.

index may have the following appearance:

```
{n                    }
{dataname             }
{arithmetic-expression}
```

L'n'->

designates the memory location at the address of the first executable FORTRAN statement following a statement label.

n is a statement label of up to 5 digits. Leading zeros must not be specified.

If no length modification value is specified, 4 bytes are searched, starting with the address stored in the address constant *L'n'*.

S'n'->

designates the memory location at the address of the FORTRAN statement with the specified number.

n is the number of a statement (see STMT column in compiler listing).

If no length modification value is specified, 4 bytes are searched, starting with the address stored in the address constant *S'n'*.

compl-memref

designates an area of 4 bytes, starting with the calculated address. If a different number of bytes is to be searched, *compl-memref* must terminate with the appropriate length modification. When modifying the length of data elements, you must pay attention to area boundaries or switch to machine code level using %@(dataname)->.
The following operations may occur in *compl-memref* (see also AID Core Manual, chapter 6):

- byte offset (•)
- indirect addressing (->)
- type modification (%A)
- length modification (%L(...), %L=(expression), %Ln)
- address selection (%@(...))

```
-------------
| alignment |
-------------
```

defines that the search for *search-criterion* is to be effected at certain aligned addresses only.

```
alignment-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          ⌈1⌉
          |2|
ALIGN [=] { }
          |4|
          ⌊8⌋
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

search-criterion is searched for at:

1 byte boundary (default)

2 halfword boundary

4 word boundary

8 doubleword boundary

Examples

1. `%FIND X'F0' IN DATA`

   The hexadecimal literal X’F0’ is searched for in the variable DATA. Any hit is output to SYSOUT.

2. `%F X'D2' IN S'12'->%L=(S'13'-S'12') ALIGN=2`

   The hexadecimal literal X’D2’ is searched for at a halfword boundary in the machine code generated for statement 12.

3. `%F`

   The search is continued with the parameters of the last %FIND command behind the last hit.

%HELP

By means of %HELP you can request information on the operation of AID. The following information is output to the selected medium: either all the AID commands or the selected command and its operands, or the selected error message with its meaning and possible responses.

– By means of the *info-target* operand you specify the command on which you need further information or the AID message for which you want an explanation of its meaning and actions to be taken.

– By means of the *medium-a-quantity* operand you specify to which output media AID is to output the required information. By means of this operand you temporarily disable a declaration made via %OUT.

```
------------------------------------------------------------------------------
Command          Operand
------------------------------------------------------------------------------

%H[ELP]          [info-target]            [medium-a-quantity][,...]

------------------------------------------------------------------------------
```

%HELP provides information on all the operands of the selected command, i.e. all language-specific operands for symbolic debugging as well as all operands for machine-oriented debugging. Refer to the relevant manual to see what is permitted for the language in which your program is written.

Messages from AIDSYS have the message code format IDA0n and are queried using /HELP.

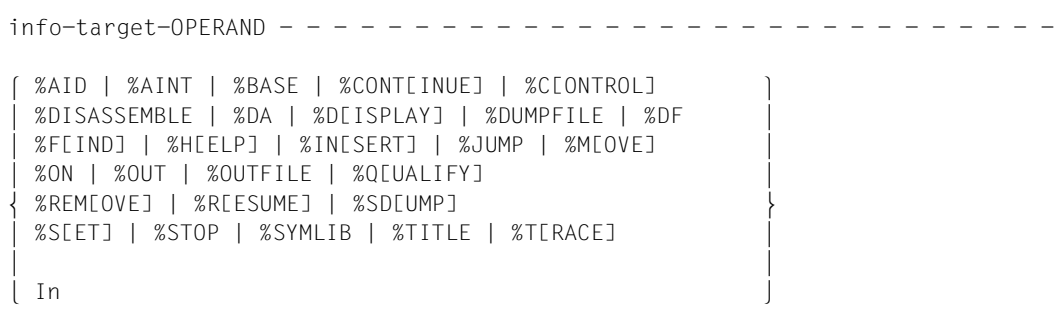
%HELP can only be entered as an individual command, i.e. it must not be contained in a command sequence or subcommand.

The %HELP command does not alter the program state.

```
---------------
| info-target |
---------------
```

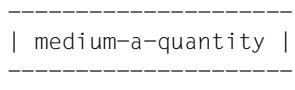
designates a command or a message number about which information is to be output. If the *info-target* operand is omitted, the command initiates output of an overview of the AID commands with a brief description of each command, and of the AID message number range.

AID responds to a %HELP command containing an invalid *info-target* operand by issuing an error message. This is followed by the same overview as for a %HELP command without *info-target*. This overview can also be requested via the %?, %H? or %H %? entries.



The AID command names may be abbreviated as shown above.

In designates the message number for which the meaning and possible responses are to be output.
n is a 3-digit message number.



defines the media via which information on the *info-target* is to be output.

The {MAX | MIN | XMAX | XFLAT} specification is not relevant for %HELP, but the syntax requires that one of these two options must be specified.

If this operand is omitted and no declaration has been made using the %OUT command, AID works with the default value T=MAX.

medium-a-quantity is described in detail in the AID Core Manual, chapter 7.

T Terminal output

H Hardcopy output

Fn File output

P Output to SYSLST

%INSERT

By means of %INSERT you can specify a test point and define a subcommand. Once the program sequence reaches the test point, AID processes the associated subcommand. In addition, the user can also specify whether AID is to delete the test point once a specific number of executions has been counted and halt the program afterwards.

– By means of the *test-point* operand you may define the address of a command in the program prior to whose execution AID interrupts the program run and to process *subcmd*.

– By means of the *subcmd* operand you may define a command or a command sequence and perhaps a condition. Once *test-point* has been reached and the condition has been satisfied, *subcmd* is executed.

– By means of the *control* operand, you can declare whether *test-point* is to be deleted after a specified number of passes and whether the program is then to be halted.

```
------------------------------------------------------------------------------
Command           Operand
------------------------------------------------------------------------------

%IN[SERT]         test-point  [<subcmd>]    [control]

------------------------------------------------------------------------------
```

A *test-point* is deleted in the following cases:

1. When the end of the program is reached.
2. When the number of passes specified via *control* has been reached and deletion of *test-point* has been specified.
3. If a %REMOVE command deleting the *test-point* has been issued.

If no *subcmd* operand is specified, AID inserts the *subcmd* <%STOP>.

The *subcmd* in an %INSERT command for a *test-point* which has already been set does not overwrite the existing *subcmd*; instead, the new *subcmd* is prefixed to the existing one. The chained subcommands are thus processed according to the LIFO rule (last in, first out).

%REMOVE can be used to delete a subcommand, a test point or all test points entered.

test-point can only be an address in the program which has been loaded, therefore the base qualification E=VM must have been set (see %BASE) or must be specified explicitly.

%INSERT does not alter the program state.

```
--------------
| test-point |
--------------
```

must be the address of an executable machine instruction generated for a FORTRAN statement. *test-point* is immediately entered by targeted overwriting of the memory position addressed and must therefore be loaded in virtual memory at the time the %INSERT command is input. Since, by entering *test-point*, the program code is modified, a test point which has been incorrectly set may lead to errors in program execution (e.g. data/addressing errors).

When the program reaches the *test-point*, AID interrupts the program and starts the *subcmd*.

```
test-point-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - -

           ⎧program-name      ⎫
           ⎪L'n'              ⎪
 [•][qua•] ⎨S'n'              ⎬
           ⎪                  ⎪
           ⎩compl-memref      ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

A qualification is only required if *test-point* is not located in the current AID work area.

E=VM

Since *test-point* can only be entered in the virtual memory of the program which has been loaded, specify *E=VM* only if a dump file has been declared as the current base qualification (see %BASE command).

PROG=program-name

is specified only if *test-point* is not in the current program unit (see chapter 3).

program-name

This specification is only possible after an explicit PROG qualification:

```
PROG=program-name•program-name
```

By repeating *program-name* you set *test-point* to the first statement of the designated program unit.

L'n'

is a statement name, designating the address of the first executable FORTRAN statement following a statement label, i.e. the address of the first machine instruction of the code sequence generated for this statement.
n is a statement label and comprises up to 5 digits. Leading zeros must not be specified.

S'n'

is a source reference and designates the address of an executable FORTRAN statement, i.e. the address of the first machine instruction of the code sequence generated for this statement.
n is the number of a statement (see STMT column in compiler listing).

compl-memref

The result of *compl-memref* must be the start address of an executable machine instruction.
compl-memref may contain the following operations (see AID Core Manual, chapter 6):

- byte offset (•)
- indirect addressing (->)
- type modification (%A)
- length modification (%Ln)
- address selection (%@(...))

A statement name *L'n'* or a source reference *S'n'* can be used within *compl-memref*, but only in connection with the pointer operator (e.g. L'200' ->.4).
Type modification makes sense only if the contents of a data element can be used as an address or if you take the address from a register,
e.g. %1G.2 %AL2 ->.

```
----------
| subcmd |
----------
```

subcmd is processed whenever program execution reaches the address designated by *test-point*.
If the *subcmd* operand is omitted, AID inserts a <%STOP>.

A complete description of *subcmd* can be found in the AID Core Manual, chapter 5.

```
subcmd-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                 ⌈AID-command    ⌉
<[subcmdname:] [(condition):] [{                 } {;...}]>
                                 ⌊BS2000-command ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A subcommand may contain a name, a condition and a command part. Every subcommand has its own execution counter. The command portion can comprise a single command or a command sequence and may contain AID and BS2000 commands as well as comments.

If the subcommand consists of a name or a condition but the command part is missing, AID merely increments the execution counter when the test point is reached.

subcmd does not overwrite an existing subcommand for the same *test-point*, rather the new subcommand is prefixed to the existing one. *subcmd* may contain the commands %CONTROLn, %INSERT, %JUMP and %ON. Nesting over a maximum of 5 levels is possible.

The commands in a *subcmd* are executed one after the other; program execution is then continued. The commands for runtime control immediately alter the program state, even in a subcommand. They abort the *subcmd* and start the program (%CONTINUE, %RESUME, %TRACE) or halt it (%STOP). They are thus only effective as the last command in a *subcmd*, since any subsequent commands in the *subcmd* would fail to be executed. Likewise, deletion of the current subcommand via %REMOVE makes sense as the last command in *subcmd* only.

```
------------
| control  |
------------
```

specifies whether *test-point* is to be deleted after the n-th pass and whether the program is to be halted with the purpose of inserting new commands.
If no *control* operand has been specified, AID assumes the defaults 2^{31}-1 (for *n*) and K.

```
control-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          [ K ]
ONLY  n [{ S }]
          [ C ]

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

n

is a number with the value $1 \leq n \leq 2^{31}$-1, specifying after how many *test-point* passes the further declarations for this *control* operand are to go into effect.

K

test-point is not deleted (KEEP).

Program execution is interrupted, and AID expects input of commands.

S

test-point is deleted (STOP).

Program execution is interrupted, and AID expects input of commands.

C

test-point is deleted (CONTINUE).

No interruption of the program.

Examples

1. `%IN S'48'`

   The statement with the number 48 is specified as *test-point*.

2. `%IN L'0' <%DISPLAY X>`

   The statement with the statement label 0 is specified as *test-point*.

3. `%IN S'3' <%DISPLAY PERSNO> ONLY 10 S`

   The statement with the number 3 is specified as the *test-point*. Whenever the program sequence arrives at the third statement, the %DISPLAY command of the *subcmd* is executed. When *test-point* is reached for the 10th time, AID sets the program to STOP and deletes the test point, at which time you may enter new commands.

4. `%IN L'2' <%DISPLAY TEXTDAT, 'L2'>`

   ```
   %IN S'3' <%DISPLAY 'INSERT1', TEXTDAT; %IN L'20' <%D 'INSERT2', -
   I,J,K, NUMBER; %IN S'172' <%D 'INSERT3' ,I,J; %REMOVE L'20'>>>
   ```

   With the first %INSERT command, the *test-point* set is the statement with the label 2. If, after the end of command input, the program execution reaches L'2', the subcommand is executed. It consists of a %DISPLAY command (for data name TEXTDAT) and the literal L'2'. Afterwards the program is continued.

   By means of the second %INSERT command, *test-point* S'3' is declared. This %INSERT command contains two other nested %INSERT commands. Their *test-point* values are still inactive for AID. They do not become active until the *test-point* of the %INSERT command in whose *subcmd* they are defined is reached.

   When program execution reaches statement S'3', the corresponding *subcmd* is executed, i.e. the %DISPLAY command for the literal 'INSERT1' and the variable TEXTDAT is executed and the *test-point* L'20' is set.

   The *subcmd* for *test-point* L'20' is still inactive. Thus, in the program to be tested, the following three *test-points* have been set at this stage in the program run: L'2', S'3' and L'20'.

   As the *subcmd* for *test-point* S'3' does not contain any %STOP command, the program is continued after execution of *subcmd*. If program execution is not interrupted for some other reason, e.g. an error or the occurrence of an event declared

by %ON, and finally reaches the symbolic address L’20’, then the %D command ’INSERT2’, I, J, K, NUMBER is executed. Furthermore, *subcmd* contains a further %INSERT command, whose *test-point* this time is specified via S’172’, i.e. statement 172.

If the position marked S’172’ is reached during further program execution, AID executes the %DISPLAY command for the literal ’INSERT3’ and the contents of variables I and J. By way of the second command in this *subcmd*, the %REMOVE L’20’ command, *test-point* L’20’ is deleted. This is necessary, for instance, if a *test-point* is located in a loop and this would lead to an undesirable chaining of nested subcommands. Without the %REMOVE command, the following *subcmd* would be created for *test-point* S’172’ during the second pass of L’20’:

```
<%D 'INSERT3', I,J; %D 'INSERT3',I,J>
```

5. `%OUT %DISPLAY P=MAX`

   ```
   %IN L'100' <%D 'I GE 10',I,X(I),K,Y(I,K)>
   ```

   ```
   %IN L'100' <(I LT 10): %D 'I LT 10',I,X(I); %CONT>
   ```

   First, all outputs of the %DISPLAY command are directed to SYSLST.
   The two subsequent %INSERTs create the following subcommand at *test-point* L’100’:

   ```
   <(I LT 10): %D 'I LT 10',I,X(I); %CONT; %D 'I GE
   10',I,X(I),K,Y(I,K)>
   ```

   Every time the program sequence reaches the statement with label 100, a check is made whether index I contains a value < 10. If the condition is satisfied, AID writes the comment ’I LT 10’ and the contents of I und X(I) to SYSLST and, as a result of %CONTINUE, continues the program (with tracing, if the subcommand interrupted a %TRACE).
   If the value of I is ≥ 10, AID writes the comment ’I GE 10’ and, in addition to I and X(I), also the values of index K and array element Y(I,K) to SYSLST and likewise continues the program. In this case, too, any active %TRACE is continued.

%JUMP

With the %JUMP command you define a continuation address at which the program is to continue with %CONTINUE, %RESUME or %TRACE. With this address you deviate from the coded program sequence. The command is acknowledged with a message reporting execution of the branch.

– With the *continuation* operand you designate the position in the program where AID is to continue following termination of command input. *continuation* can only be the address of a FORTRAN statement.

Command	Operand
%JUMP	continuation

%JUMP can only be used for program units which were compiled with FOR1 as of V2.1A and which have not been optimized (SDF option OPTIMIZATION=NO or COMOPT statement OPTIMIZE=NO).

The continuation address must be located in the same program unit in which the program was interrupted, otherwise AID outputs an error message. AID does not make any other checks. The user must ensure that the prerequisites (e.g. index or counter states, file status) for error-free execution of program as of *continuation* have been fulfilled. This is especially important if you use the %JUMP command to reach an address which comes logically before the interrupt point in the course of program execution.

You may not enter the %JUMP command in the following cases:

– immediately after the LOAD-PROGRAM command

– if the program has been interrupted by the system, e.g. because a file to be opened has not yet been assigned

– if the K2 key has been used to interrupt the program

– if the program has been halted by the FORTRAN statement PAUSE.

The %JUMP command does not alter the program state.

continuation

defines the position at which the program is to be continued. *continuation* must be the address of an executable statement within the current program unit. If the %JUMP command is part of a subcommand, *continuation* must designate a statement in the program unit which also contains the test point or in which the event defined with %ON has occurred.

L'n'

is a statement name designating the address of the first executable FORTRAN statement after a statement label.

n is a statement label and consists of up to 5 digits. Leading zeros must not be specified.

S'n'

is a source reference designating the address of an executable FORTRAN statement.
n is the number of a statement (see STMT column of compiler listing).

Examples

1. `%JUMP S'24'`

   The statement with number 24 is declared as the continuation address.

2. `%JUMP L'100'`

   The first executable statement following statement label 100 is declared as the continuation address.

%MOVE

With the %MOVE command you transfer memory contents or AID literals to memory positions within the program which has been loaded. Transfer is effected without checking and without matching of sender and receiver storage types.

– With the *sender* operand you designate a variable, an array or an array element, a length, an execution count, an AID register or an AID literal. *sender* can be located in virtual memory of the loaded program or in a dump file.

– With the *receiver* operand you designate a variable, an array or an array element, an execution counter or an AID register which is to be overwritten. *receiver* can only be located in virtual memory of the loaded program.

– With the *REP* operand you specify whether AID is to generate a REP record in conjunction with a modification which has taken place. This operand has a higher priority than a default specified in the %AID command but affects only the current %MOVE.

```
_____________________________________________________________________________
Command             Operand
_____________________________________________________________________________

%M[OVE]             sender  INTO  receiver      [REP]

_____________________________________________________________________________
```

In contrast to the %SET command, AID does not check for compatibility between the storage types *sender* and *receiver* when the %MOVE command is involved, and does not match these two storage types.

AID passes the information left-justified, with the length of *sender*. If the length of *sender* is greater than that of *receiver*, AID rejects the attempt to transfer and issues an error message.

Also in contrast to the %SET command, the %MOVE command can be used to transfer or overwrite complete COMPLEX data elements. However, you also have the option (as with %SET) of using the operands *dataname*._REAL or *dataname*._IMAG to apply the modification only to the real or imaginary portion of the complex number.

Input of the command immediately following loading is not recommended, as you cannot address data and statements without an explicit qualification until the program is about to process the first executable statement. The following command sequence must be entered:
%INSERT PROG=program-name.program-name
%RESUME

In addition to the operand values described here, the values described in the manual for debugging on machine code level can also be employed.

Using `%AID CHECK=ALL` you can also activate an update dialog, which first provides you with a display of the old and new contents of *receiver* and offers you the option of aborting the %MOVE command.

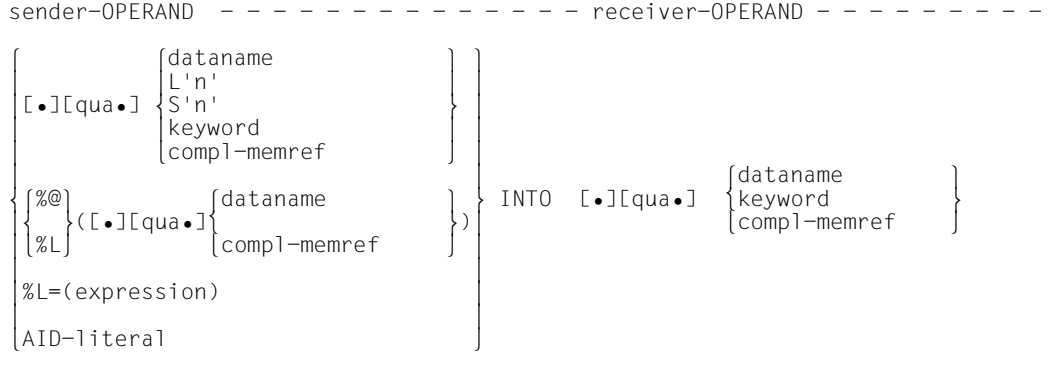
The %MOVE command does not alter the program state.

```
----------          ------------
| sender |   INTO   | receiver |
----------          ------------
```

For *sender* or *receiver* you can specify a variable, an array or an array element, a complex memory reference, an execution counter, or a register. Symbolic constants, addresses and lengths of data elements as well as AID literals can only be employed as *sender*.

sender may be either in the virtual memory area of the program which has been loaded or in a dump file; *receiver*, on the other hand, can only be within the virtual memory of the loaded program.

No more than 3900 bytes can be transferred with a %MOVE command. If the area to be transferred is larger, you must issue multiple %MOVE commands.



- If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

A qualification is necessary only if *sender* or *receiver* is not within the current AID work area.

E={VM | Dn} for *sender*

E=VM for *receiver*

You specify a base qualification only if the current base qualification is not to apply for a data/statement name, source reference or keyword (see %BASE).
sender may be either in virtual memory or in a dump file; *receiver*, on the other hand, can only be in virtual memory.

PROG=program-name

is to be specified only if you address a data/statement name or source reference that is not in the current program unit (see chapter 3).

NESTLEV= level-number

level-number A level number in the current call hierarchy
level-nummer has to be followed by *dataname*.
Specify NESTLEV= *level-number* when you want to address a data name on a certain level in the current call hierarchy. This qualification can only be combined with E=, and not with any other qualification.

dataname

specifies the name of a constant, variable, array or array element as defined in the source program. Constants can only be used as *sender*.

dataname is an alphanumeric string consisting of up to 15 characters.

If *dataname* is the name of an array, then you must index it as in a FORTRAN statement if you want to address an array element. If you specify the name of an array without an index list, this means that all array elements will be transferred (in the case of *sender*). If you specify the name of an array without an index list in the case of *receiver*, the array will be overwritten beginning at the start address and using the length of *sender*, without taking into account the subdivision into array elements.

array-name (index1[, index2][, ...])

index specifies the position within an array. The number of indexes required for access is the same as in a FORTRAN statement. Multiple indexes must be separated by a comma.
index may be specified as follows:

```
{n        }
{dataname }
```

```
[arithmetic-expression  ]
```

L’n’

is a statement name designating the address of the first executable FORTRAN statement following a statement label.

n is a statement label and consists of up to 5 digits. Leading zeros must not be specified.

S’n’

is a source reference designating the address of an executable FORTRAN statement.

n is the number of a statement (see STMT column of compiler listing).

Statement names and source references are address constants and can therefore only be specified for *sender*. The address designated using *L’n’* or *S’n’* is then transferred.

Example

```
%MOVE S'5' INTO %0G
```

The address of the statement with number 5 is written to AID register %0G.

With *L’n’->* or *S’n’->* you designate 4 bytes of the machine code at the corresponding address (see AID Core Manual, section 6.4).
%DISASSEMBLE can be used to output the machine instructions in order to perform any length modification.
In the case of *receiver*, you may use statement names and source references only in connection with the pointer operator (->).

Example

```
%MOVE S'12'->%L=(S'13'-S'12') INTO S'24'->
```

By means of this %MOVE command you modify the code of your program. The machine code for statement 24 is overwritten by that of statement 12. The specification %L=(S’13’-S’12’) yields the length of the machine code generated for statement 12.

keyword

specifies an execution counter, the program counter, or a register. *keyword* may only be preceded by a base qualification.

```
%.subcmdname       Execution counter
%.                 Execution counter of the current subcommand
%PC                 Program counter
%n                  General register, 0 ≤ n ≤ 15
%nD|E               Floating-point register, n = 0,2,4,6
%nQ                 Floating-point register, n = 0,4
%nG                 AID general register, 0 ≤ n ≤ 15
%nDG                AID floating-point register, n = 0,2,4,6
```

compl-memref

may contain the following operations (see AID Core Manual, chapter 6):

– byte offset (•)

– indirect addressing (->)

– type modification %E

– length modification (%L(...), %L=(expression), %Ln)

– address selection (%@(...))

A subsequent type modification for *compl-memref* is pointless, since transfer is always in binary form, regardless of the storage type of *sender* and *receiver*. However, a type modification may be necessary before a pointer operation (->).

Example

```
%0G.2%AL2->
```

The last two bytes of AID register %0G are to be used as the address.

After byte offset (•) or pointer operation (->), the implicit storage type and implicit length of the original address are lost. At the calculated address, storage type %X with length 4 applies, if no value for type and length has been explicitly specified by the user.
For each operand in a complex memory reference the assigned memory area must not be exceeded as the result of byte offset or length modification, otherwise AID does not execute the command and writes an error message. By combining the address selection (%@) with the pointer operator (->) you can exit from the symbolic level. You may then use the address of a data element without having to take note of its area boundaries.

Example

The variables CARRAY and CFIELD1 each occupy 5 bytes. The last 2 bytes of CARRAY as well as the 3 following bytes are to be transferred to CARRAY1. AID would reject the following command as a violation of the CARRAY area:

```
%MOVE CARRAY.3%L5 INTO CFIELD1
```

The correct command reads:

```
%MOVE %@(CARRAY)->.3%L5 INTO CFIELD1
```

%@(...)

With the address selector you can use the address of a data element or complex memory reference as *sender* (see AID Core Manual, section 6.11). The address selector produces an address constant as a result.

%L(...)

With the length selector you can use the length of a data element or complex memory reference as *sender* (see AID Core Manual, section 6.11). The length selector produces an integer as a result.

Example

```
%MOVE %L(ARRAY1) INTO %OG
```

The length of ARRAY1 will be transferred.

%L=(expression)

With the length function you can calculate the value of *expression* and have it stored in *receiver* (see AID Core Manual, sections 6.9 and 6.10). In *expression* you may combine the contents of memory references, constants of type ’integer’ and integers with the arithmetic operators (+,–,*,/). The length function produces an integer as a result.

Example

```
%MOVE %L=(ARRAY1) INTO %OG
```

The contents of ARRAY1 are transferred. FIELD1 must be of type ’integer’, otherwise AID issues an error message.

AID literal

The following AID literals (see AID Core Manual, chapter 8) can be transferred using %MOVE:

```
{C'x...x' | 'x...x'C | 'x...x'}      Character literal
{X'f...f' | 'f...f'X}                Hexadecimal literal
{B'b...b' | 'b...b'B}                Binary literal
[{±}]n                               Integer
#f...f'Hexadecimalnumber'
```

```
_______
| REP |
_______
```

Specifies whether AID is to generate a REP record after a modification has been performed. With *REP* you temporarily deactivate a declaration made with the %AID command. If *REP* is not specified and there is no valid declaration in the %AID command, no REP record is created.

```
REP-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

REP = {Y[ES] | NO}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

REP=Y[ES]

LMS UPDR records (REPs) are created for the update caused by the current %MOVE. If the object structure list is not available, no REP records are generated and AID will output an error message.
Also, if *receiver* is not located completely within one CSECT, AID will output an error message and not write a REP record. To obtain REP records despite this, the user may distribute transfer operations over several %MOVE commands in which the CSECT limits are observed (see [2]).

AID stores the REPs with the requisite LMS UPDR statements in a file with the link name F6, from which they can be fetched as a complete package. Therefore no other output should be written to the file with link name F6.

If no file with link name F6 is registered (see %OUTFILE), the REP is stored in the file AID.OUTFILE.F6 created by AID.

REP=NO

No REPs are created for the current %MOVE command.

Examples

The following variables and arrays are defined in a FORTRAN program:

```
INTEGER*2 IARRAY(10)
INTEGER*4 JARRAY(10)
REAL*4 RNUMBER
CHARACTER*4 CVAR
```

1. `%MOVE IARRAY INTO JFIELD`

   No index has been specified for the two arrays: AID therefore transfers the contents of IARRAY to the symbolic address JARRAY in hexadecimal format and left-justified, without taking into account any subdivision into array elements.

2. `%MOVE 20 INTO JARRAY(2)`

   AID writes a word containing an integer with the value 20 to the array element JARRAY(2) of type INTEGER*4.

3. `%MOVE 20 INTO RNUMBER`

   As in example 2, a word with the contents X’00000014’ is written to RNUMBER, which of course makes no sense when a REAL number is involved. To transfer value 20 to RNUMBER, you will have to enter a %SET command (see %SET), which performs conversion prior to the transfer.

4. `%MOVE X'58F0C160' INTO CVAR REP=YES`

   The contents of the CVAR variable are overwritten with the hexadecimal literal X’58F0C160’. A REP record is created for the correction and is stored in the file AID.OUTFILE.F6 or the file assigned to link name F6.

%ON

With the %ON command you define events and subcommands. When a selected *event* occurs, AID processes the associated *subcmd*.

– With *write-event* you define the event of write access to a memory area. Whenever the program changes the memory area specified, AID is to interrupt the program in order to process the *subcmd*.

– With *write-event*

– With *event* you define normal or abnormal program termination, a supervisor call (SVC), a program error or any event for which AID is to interrupt the program in order to process the *subcmd*.

– With *subcmd* you define a command or a command sequence and perhaps a condition. When *event* occurs and this condition is satisfied, *subcmd* is executed.

```
Command          Operand
```

```
                 ⎡write-event ⎤
%ON              ⎨            ⎬        [<subcmd>]
                 ⎣event       ⎦
```

If an *event* is not deleted, it remains valid until the program ends.

If the *subcmd* operand is omitted, AID inserts the *subcmd* <%STOP>.

The *subcmd* of an %ON command for an *event* which has already been defined does not overwrite the existing *subcmd*, rather the new *subcmd* is prefixed to the existing subcommand. This means that chained subcommands are processed in accordance with the LIFO principle. This does not apply for *write-event*. Entering a new *write-event* overwrites any which already exists.

An entered event applies until it is deleted with %REMOVE or until the program terminates.

The base qualification E=VM must apply for %ON (see %BASE).

The %ON command does not alter the program state.

write-event

You activate write access monitoring by means of the keyword %WRITE(...). The memory area to be monitored is specified in brackets. Whenever a byte within the specified memory area is write-accessed, AID executes the subcommand defined in %ON. The interrupt point is after the hardware instruction that caused the interruption. This hardware instruction may me situated in the program or in the runtime system. The area to be monitored must not be larger than 65535 bytes.

Subcommands for *write-event* cannot be chained. It is not possible to monitor two or more areas at the same time. However, 'write-event' can be defined at the same time as other events.

The storage location is unchanged during program execution. If this area is overloaded in case of a program the meanig of this area can be changed. After the program interrupt this meaning must be verified. (%WRITE (VAR) defines the storage location of VAR at the time of AID command execution. This storage is monitored during program execution but not the variable VAR by itself).

Only one *write-event* can be defined. Entering a new *write-event* overwrite any which already exists.

When an *event* arrives at the same time as a *write-event*, AID processes the subcommand for the *write-event* first.
You delete the *write-event* using %REMOVE %WRITE without specifying the memory reference.

The following interdependencies exist between %ON *write-event* and other AID commands:

– When a %CONTROLn or a %TRACE with a machine-oriented *criterion* is registered, the entry of %ON *write-event* is rejected with an error message.

– When a machine instruction has been overwritten with the AID-internal marking (X'0A81') by a %CONTROLn or %TRACE with symbolic *criterion*, AID does not notice the write access of this instruction.

– When a machine instruction has been overwritten with the AID-internal marking by the test point defined with %INSERT, AID does not recognize the write access of this instruction here, either.

To ensure gap-free write monitoring, it is recommended that all %CONTROLn and %INSERT commands be deleted with %REMOVE and that any %TRACE entered be deleted by continuing with %RESUME after %ON.

The memory area to be monitored can be any memory object addressed in any way. It is defined by the start address and implicit or explicit length. The length may not exceed 1024 bytes, otherwise an error message is issued.

When, in the case of a program with overlay structure, the address of the specified memory object is overloaded, the corresponding area of the newly loaded program unit is monitored.

Further information on write-access monitoring can be found in sections 10.13 and 12.1 of the "AID Core Manual".

```
write-event OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                                  ⎧dataname          ⎫
                                                  ⎪L'n'->            ⎪
%WRITE ( [⎧•                      ⎫ ] ⎧S'n'->                      ⎪)
          ⎩PROC=program-name.     ⎭   ⎪                            ⎪
                                      ⎩compl-memref                 ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•
: If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY.
In addition, there must be a period between the PROG qualification and the following operand part.

PROG=program-name
: Specify only when you address a data or statement name or a source reference which is not within the current program unit (see chapter 3).

dataname
: is the name of a variable, array or array element as defined in the source program. *dataname* is an alphanumeric string consisting of up to 15 characters.

  If *dataname* is the name of an array, it can be indexed as in a FORTRAN statement if an array element is to be addressed. If you specify the name of an array without the index list, the entire array is monitored.

  array-name (index1[, index2][, ...])
  : *index* specifies the position within an array. The number of indexes required for access is the same as that which must be specified in a FORTRAN statement. When multiple indexes are specified, a comma must be used as a separator.

    index may be specified as follows:

    ⎡n ⎤

```
{dataname                }
{arithmetic-expression   }
```

L'n'->
: designates the memory location at the address of the first executable FORTRAN statement following a statement label.
n is a statement label of up to 5 digits. Leading zeros must not be specified.

  If no length modification value is specified, 4 bytes are searched, starting with the address stored in the address constant *L'n'*.

S'n'->
: designates the memory location at the address of the FORTRAN statement with the specified number.
n is the number of a statement (see the STMT column in compiler listing).

  If no length modification value is specified, 4 bytes are monitored, starting with the address stored in the address constant *S'n'*.

compl-memref
: designates an area of 4 bytes, starting with the calculated address. If a different number of bytes is to be monitored, *compl-memref* must terminate with the appropriate length modification. When modifying the length of data elements, you must pay attention to area boundaries or switch to machine code level using %@(dataname)->. The following operations may occur in *compl-memref* (see AID Core Manual, chapter 6):

  – byte offset (.)
  – indirect addressing (->)
  – type modification (%A)
  – length modification (%L(...), %L=(expression), %Ln)
  – address selection (%@(...))

```
_________
| event |
_________
```

A keyword is used to specify an event (program error, abnormal termination of the program, supervisor call, etc.) upon which AID is to process the *subcmd* specified.

If several %ON commands with different *event* declarations are simultaneously active and satisfied, AID processes the associated subcommands in the order in which the keywords are listed in the table below. If various %TERM events are applicable, the associated subcommands are processed in the opposite order in which the %TERM events have been declared (LIFO rule as for chaining of subcommands). If a *write-event* and another *event* are occuring simultanously, then the subcommand for the *write-event* is processed first.

For selection of the SVC numbers see the "Executive Macros" manual [6].

In an %ON command, it is not advisable to define events which are already covered by FOR1 error recovery routines. Such events include the following interrupt conditions:
%ERRFLG(zzz), %INSTCHK, %ARTHCHK, %ABNORM and %ERRFLG
These events can only be addressed in an %ON command if the FOR1 error recovery routines have been suppressed (possible only for FOR1 programs without standard linkage).
To do so, enter the following commands:

```
  /PARAMETER CARD = YES
  .
  .
⎧/LOAD-PROGRAM  ⎫
⎨               ⎬ ..., TEST-OPT = AID
⎩/START-PROGRAM⎭
  .
  .
  GIVE 'RUNOPT' OR 'END' OR '?'
  *RUNOPT STXIT = NO
  .
  .
```

The following restriction applies to the events %ERRFLG[(zzz)], %INSTCK, %ARTHCHK and %ABNORM: AID cannot process these events if STXIT routines of, for example, the runtime system or ILCS have been defined. More detailed information on %ON and STXIT can be found in section 11.1 of the "AID Core Manual".

event	*subcmd* is processed:	
%ERRFLG (zzz)	after	the occurrence of an error with error weight zzz and
	before	abortion of the program
%INSTCHK	after	the occurrence of an addressing error, an impermissible supervisor call (SVC), an operation code which cannot be decoded, a paging error or a privileged operation and
	before	abortion of the program
%ARTHCHK	after	the occurrence of a data error, divide error, exponent overflow or a zero mantissa and
	before	abortion of the program
%ABNORM	after	the occurrence of one of the errors covered by the previously described events
%ERRFLG	after	the occurrence of an error with any error weight

%SVC(zzz)	before execution of the supervisor call (SVC) with the specified number
%LPOV(xxxxxxxx)	after loading of the segment with the specified name xxxxxxxx
%LPOV	after loading of any arbitrary segment (the name is output with %D %LINK)
%TERM(N[ORMAL])	before normal termination of a program
%TERM(A[BNORMAL])	before abnormal termination of a program, but after output of a memory dump
%TERM	before termination of a program by any of the %TERM events described above
%ANY	before termination of a program with %TERM because of program error or as a result of the %TERM events described above
%SVC	before execution of any supervisor call

zzz may be specified in one of two formats:

n unsigned decimal number of up to three digits

#'ff' two-digit hexadecimal number

The following applies for the value *zzz*: $1 \leq zzz \leq 255$

No check is made whether the specified number of the error weight or the SVC number is meaningful or permissible.

```
----------
| subcmd |
----------
```

is processed whenever the specified *event* occurs in the course of program execution. If the *subcmd* operand is omitted, AID inserts a <%STOP>.

For a complete description of *subcmd* refer to the AID Core Manual, chapter 5.

```
subcmd-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                 ⌈AID-command    ⌉
<[subcmdname:] [(condition):] [{                 } {;...}]>
                                 ⌊BS2000-command ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A subcommand may comprise a name, a condition and a command part. Every subcommand has its own execution counter. The command portion can consist of either an individual command or a command sequence; it may contain AID and BS2000 commands as well as comments.

If the subcommand contains a name or condition but no command part, AID merely increments the execution counter when the declared event occurs.

subcmd does not overwrite an existing subcommand for the same *event*. Instead, the new subcommand is prefixed to the existing one. The %CONTROLn, %INSERT, %JUMP and %ON commands are permitted in *subcmd*. The user can form up to 5 nesting levels. An example can be found under the description of the %INSERT command.

The commands in a *subcmd* are executed one after the other; then the program is continued. The commands for runtime control immediately alter the program state, even in a subcommand. They abort *subcmd* and continue the program (%CONTINUE, %RESUME, %TRACE) or halt it (%STOP). They should only be placed as the last command in a *subcmd*, since any subsequent commands of the *subcmd* will not be executed. Likewise, deletion of the current subcommand via %REMOVE makes sense only as the last command in *subcmd*.

Examples

1. `%ON %LPOV (MON12) <%D '%LPOV (MON12)'; %STOP>`

   After MON12 has been loaded, AID outputs the literal ’%LPOV (MON12)’ and interrupts the program.

2. `%ON %ERRFLG (108)`

   `%ON %ERRFLG (#'6C')`

   Both specifications designate the same program error (mantissa equals zero).

3. `%ON %ERRFLG (107) <%D 'ERROR'>`

   This error weight does not exist, therefore the *subcmd* defined for this *event* will never be started.

%OUT

With %OUT you define the media via which data is to be output and whether output is to contain additional information, in conjunction with the output commands %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE.

– With *target-cmd* you specify the output command for which you want to define *medium-a-quantity*.

– With *medium-a-quantity* you specify which output media are to be used and whether or not additional information is to be output.

```
_____________________________________________________________________________
Command             Operand
_____________________________________________________________________________

%OUT                [target-cmd              [medium-a-quantity][,...]  ]

_____________________________________________________________________________
```

In the case of %DISPLAY, %HELP and %SDUMP commands, you may specify a *medium-a-quantity* operand which for these commands temporarily deactivates the declarations of the %OUT command. %DISASSEMBLE and %TRACE include no *medium-a-quantity* operand of their own; their output can only be controlled with the aid of the %OUT command.

Before selecting a file as the output medium via %OUT, you must issue the %OUTFILE command to assign the file to a link name and open it; otherwise AID creates a default output file with the name AID.OUTFILE.Fn.

The declarations made with the %OUT command are valid until overwritten by a new %OUT command, or until /LOGOFF.

An %OUT command without operands assumes the default value T=MAX for all *target-commands*.

%OUT may only be specified as an individual command, i.e. it may not be part of a command sequence or subcommand.

%OUT does not alter the program state.

```
______________
| target-cmd |
______________
```

designates the command for which the declarations are to apply. Any of the commands listed below may be specified.

```
┌%D[IS]A[SSEMBLE]┐
│%D[ISPLAY]      │
{%H[ELP]         }
│%SD[UMP]        │
└%T[RACE]        ┘
```

```
---------------------
| medium-a-quantity |
---------------------
```

In conjunction with *target-cmd* this specifies the medium or media via which output is to take place, as well as whether or not AID is to output additional information pertaining to the AID work area, the current interrupt point and the data to be output.

If the *medium-a-quantity* operand has been omitted, the default value T=MAX applies for *target-cmd*.

```
medium-a-quantity-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - -

{T   } {MIN  }
{H   } {MAX  }
{Fn  } {XMAX }
{P   } {XFLAT}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-a-quantity is described in detail in the AID Core Manual, chapter 7.

T Terminal output

H Hardcopy output

Fn File output

P Output to SYSLST

AID does not take into account XMAX and XFLAT modes for outputting the %OUT log. Instead, it generates the default value (T=MAX).

MAX	Output with additional information
MIN	Output without additional information
XMAX	Definition of XMAX mode for the corresponding command %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP or %TRACE.
XFLAT	Definition of XFLAT mode for the corresponding command %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP or %TRACE.

Examples

1. `%OUT %SDUMP T=MIN,F1=MAX`

   Data output of the %SDUMP command should be output on the terminal in abbreviated form, and in parallel to this also to the file with link name F1, along with additional information.

2. `%OUT %TRACE F1=MAX`

   The TRACE log with additional information is output only to the file with link name F1.

3. `%OUT %TRACE`

   For the %TRACE command, this specifies that previous declarations for output of data are erased, and that the default value T=MAX applies.

%OUTFILE

%OUTFILE assigns output files to AID link names F0 through F7 or closes output files. You can write output of the commands %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE to these files by specifying the corresponding link name in the *medium-a-quantity* operand of %OUT, %DISPLAY, %HELP or %SDUMP. If a file does not yet exist, AID will make an entry for it in the catalog and then open it.

– With *link* you select a link name for the file to be cataloged and opened or closed.

– With *file* you assign a file name to the link name.

```
_____________________________________________________________________________
Command          Operand
_____________________________________________________________________________

%OUTFILE         [link    [ = file]]

_____________________________________________________________________________
```

If you do not specify the *file* operand, this causes AID to close the file designated using *link*. In this way an intermediate status of the file can be printed during debugging.

An %OUTFILE without operands closes all open AID output files. If you have not explicitly closed an AID output file using the %OUTFILE command, the file will remain open until /LOGOFF.

Without %OUTFILE, you have two options of creating and assigning AID output files:

1. Enter a /SET-FILE-LINK command for a link name Fn which has not yet been reserved. Then AID opens this file when the first output command for this link name is issued.
2. Leave the creation, assignment and opening of files to AID. AID then uses default file names with the format AID.OUTFILE.Fn corresponding to link name *Fn*.

%OUTFILE does not alter the program state.

```
________
| link |
________
```

Designates one of the AID link names for output files and has the format Fn, where n is a number with a value $0 \leq n \leq 7$.

The REP records for the %MOVE command are written to the output file with link name F6 (see also the %AID and %MOVE commands).

```
--------
| file |
--------
```

specifies the fully-qualified file name with which AID catalogs and opens the output file. Use of an %OUTFILE command without the *file* operand closes the file assigned to link name Fn.

%QUALIFY

With %QUALIFY you define qualifications. In the address operand of another command you may refer to these qualifications by prefixing a period.

Use of this abbreviated format for a qualification is practical whenever you want to repeatedly reference addresses which are not located in the current AID work area.

– By means of the *prequalification* operand you define qualifications which you would like to incorporate in other commands by referencing them via a prefixed period.

```
_____________________________________________________________________________
Command           Operand
_____________________________________________________________________________

%Q[UALIFY]        [prequalification]

_____________________________________________________________________________
```

A *prequalification* specified with the aid of the %QUALIFY command applies until it is overwritten by a %QUALIFY with a new *prequalification* or revoked by a %QUALIFY without operands, or until /LOGOFF.

On input of a %QUALIFY command, only a syntax check is made. Whether the specified link name has been assigned a dump file or whether the specified program unit has been loaded or included in the LSD records is not checked until subsequent commands are executed and the information from *prequalification* is actually used in addressing.

The declarations of the %QUALIFY command are only used by commands which are input subsequently. %QUALIFY has no effect on any subcommands in %CONTROL, %INSERT and %ON commands entered prior to this %QUALIFY command, even if they are executed after it.

The same %AID LOW={ON|OFF} setting must apply for input of the %QUALIFY and for replacement in an address operand.

%QUALIFY may only be specified as an individual command, i.e. it may not be part of a command sequence or subcommand.

The %QUALIFY command does not alter the program state.

```
____________________
| prequalification |
____________________
```

designates a base qualification or a PROG qualification or both qualifications, which must then be separated by a period.

The reference to a *prequalification* defined in the %QUALIFY command is effected by prefixing a period to the address operands of subsequent AID commands.

```
prequalification operand  - - - - - - - - - - - - - - - - - - - - - - - - -

   ⌈VM ⌉
[E=⎨   ⎬[.]][PROG=program-name]
   ⌊Dn ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

E={VM|Dn}

must be specified if you want to use a base qualification which is different from the current one (see %BASE command).

PROG=program-name

designates a program unit.

Examples

1. `%QUALIFYE=D1.PROG=SORT`

   `%D .CARRAY(1)`

   Because of the *prequalification*, the %DISPLAY command has the same effect as the following %DISPLAY command in full format:

   `%D E=D1.PROG=SORT.CARRAY(1)`

2. `%QUALIFYPROG=SUB`

   `%SET .A INTO .B`

   Because of the *prequalification*, the %SET command has the same effect as the following %SET command in full format:

   `%SET PROG=SUB.A INTO PROG=SUB.B`

%REMOVE

With the %REMOVE command you revoke the test declarations for the %CONTROLn, %INSERT and %ON commands.

– With *target* you specify whether AID is to revoke all effective declarations for a particular command or whether only a specific test point or event or a subcommand is to be deleted.

```
_____________________________________________________________________________
Command            Operand
_____________________________________________________________________________

%REM[OVE]          target

_____________________________________________________________________________
```

If a subcommand contains a %REMOVE which deletes this subcommand or the associated monitoring condition (*test-point*, *event* or *criterion*), any subsequent *subcmd* commands will not be executed. Such an entry is therefore only meaningful as the last command in a subcommand.

The %REMOVE command does not alter the program state.

```
__________
| target |
__________
```

Designates a command for which all the valid declarations are to be deleted, or a *test-point* to be deleted, or an *event* which is no longer to be monitored, or the subcommand to be deleted. If *target* is within a nested subcommand and therefore has not yet been entered, it cannot be deleted either.

```
target-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

⎧%C[ONTROL] | %C[ONTROL]n    ⎫
⎪                           ⎪
⎪%IN[SERT] | test-point     ⎪
⎨                           ⎬
⎪%ON | %WRITE | event       ⎪
⎪                           ⎪
⎩%•[subcmdname]             ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

%C[ONTROL]

The declarations for all %CONTROLn commands entered are deleted.

%C[ONTROL]n

The %CONTROLn command with the specified number ($1 \leq n \leq 7$) is deleted.

%IN[SERT]

All test points which have been entered are deleted.

test-point

The specified *test-point* is deleted. *test-point* is specified as under the %INSERT command.
Within the current subcommand, *test-point* can also be deleted with the aid of %REMOVE %PC->, as the program counter (%PC) contains, at this point in time, the address of the *test-point*.

%ON

All events which have been entered are deleted.

%WRITE

The write event is deleted.

event

The specified *event* is deleted. *event* is specified with a keyword, as under the %ON command. The *event* table with the keywords and explanations of the individual events can be found under the description of the %ON command.

The following applies for the events %ERRFLG(zzz), %SVC(zzz) and %LPOV(zzz):

%REMOVE *event(zzz)* deletes only the event with the specified number. %REMOVE *event* without specification of a number deletes all events of the corresponding group.

%•[subcmdname]

deletes the subcommand with the name *subcmdname* in a %CONTROLn or %INSERT command.

%• is the abbreviated form of a subcommand name and can only be used within the subcommand. %REMOVE %. deletes the current subcommand and is thus only practical as the last command in a subcommand, since any commands following it within a *subcmd* will not be executed.

As %CONTROLn cannot be chained, the associated %CONTROLn will be deleted as well. Deleting the subcommand therefore has the same effect as deleting the %CONTROLn by specifying the appropriate number.

On the other hand, several subcommands may be chained at a *test-point* of the %INSERT command. With the aid of %REMOVE %.[subcmdname] you can delete an individual subcommand from the chain, while further subcommands for the same *test-point* will still continue to exist (see AID Core Manual, chapter 5). If only the subcommand designated *subcmdname* was entered for the *test-point*, the *test-point* will be deleted along with the subcommand.

%REMOVE %.[subcmdname] is not permitted for %ON.

Examples

1.
```
%C1 %CALL <CTL1: %D %.>
   %REM %C1
   %REM %.CTL1
```

Both %REMOVE commands have the same effect: %C1 is deleted.

2.
```
%IN L'100' <SUB1: %D I,J,IARRAY(I,J)>
   %IN L'100' <SUB2: %D %PC; %REM %.>
   .
   .
   %REM L'100'
```

When the test point L’100’ is reached, the program counter is output. Then subcommand SUB2 is deleted, i.e. this subcommand is executed only once. Subsequently the indexes I and J and the associated array element IARRAY(I,J) are output, and the program continues. Whenever test point L’100’ is reached in the program sequence, subcommand SUB1 is executed. %REM L’100’ deletes the test point later on. %REM %.SUB1 would have the same effect, as this subcommand is the only remaining entry for test point L’100’.

%RESUME

With %RESUME you start the loaded program or continue it at the interrupt point or the point specified in the %JUMP command. The program executes without tracing.

If the program has been halted during execution of a %TRACE command, the %TRACE command will be aborted. If an interrupted %TRACE is to be continued, the %CONTINUE command must be issued instead of %RESUME.

Command	Operand
%R[ESUME]	

If a %RESUME command is contained within a command sequence or subcommand, any commands which follow it will not be executed.

If the %RESUME command is the only command in a subcommand, the execution counter is incremented and any active %TRACE deleted.

The %RESUME command alters the program state.

%SDUMP

With %SDUMP you can output a symbolic dump: individual data elements, all data elements of the current call hierarchy, or the program names of the current call hierarchy. The current call hierarchy extends from the subprogram level on which the the program was interrupted to the subprograms invoked by CALL statements to the main program. Output is via SYSOUT, SYSLST or to a cataloged file.

– With *dump-area* you designate the variables or arrays which AID is to output, or you specify that AID is to output the program names of the current call hierarchy.

– With *medium-a-quantity* you specify which output media AID is to use, and whether or not additional information is to be output. This operand is used to deactivate a declaration made by the %OUT command, as far as the current %SDUMP command is concerned.

```
_____________________________________________________________________________
Command           Operand
_____________________________________________________________________________

%SD[UMP]          [[dump-area][,...]    [medium-a-quantity][,...]]

_____________________________________________________________________________
```

With the %SDUMP command, data can only be addressed after initialization, i.e. when the first executable statement of a program unit has been reached. achieve this, enter the following two commands:

%INSERT PROG=program-name.program-name

%RESUME

If program units for which there are no LSD records, not even in a PLAM library, are included in the hierarchy, the user can only issue the %SDUMP command individually for program units for which LSD records have been loaded or can be loaded from a PLAM library (see %SYMLIB command).

%SDUMP without operands outputs all data elements of the current call hierarchy. Multiply defined data is also output multiply.

%SDUMP %NEST outputs the names of all program units of the current call hierarchy.

dump-area can be repeated up to 7 times.

With this command the user can work either in the loaded program or in a dump file.

The %SDUMP command does not alter the program state.

```
-------------
| dump-area |
-------------
```

describes which information AID is to output.

AID can output the program names of the current call hierarchy, all data of the current call hierarchy, all data of a program unit or individual data elements. AID edits the data elements in accordance with the definition in the source program. If the contents do not match the defined storage type, output is rejected and an error message is issued.

If *dataname* is defined in multiple program units of the current call hierarchy it is also output repeatedly, unless *dump-area* has been restricted by a qualification.
If *dataname* is not contained in the LSD records, AID issues an error message; subsequent *dump-areas* of the same command are output, however.

```
dump-area-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

⎧      ⎧VM ⎫     ⎧[PROG=program-name[•]] [dataname] ⎫
⎨[•][E=⎨   ⎬[•]] ⎨                                  ⎬
⎩      ⎩Dn ⎭     ⎩%NEST                            ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

E ={VM | Dn}
: An explicit base qualification is to be entered only if the current base qualification is not to apply for the *dump-area*. If you specify only a base qualification, all data of the corresponding call hierarchy will be output.

PROG=program-name
: A PROG qualification is mandatory if *dump-area* is to apply only for the specified program unit. If the definition of *dump-area* terminates with a PROG qualification, AID will output all data elements of this program unit.

dataname

is the name of a constant, variable, array or array element as defined in the source program.

dataname is an alphanumeric string consisting of up to 15 characters. If *dataname* is the name of an array, it can be indexed in the same way as in a FORTRAN statement.

If *array-name* is specified without an index list, all array elements will be output.

array-name (index1[, index2][, ...])

index specifies the position within an array. The number of indexes required for access is the same as in a FORTRAN statement. When multiple indexes are involved, a comma must be used to separate them.
index may be specified as follows:

```
⎧n                     ⎫
⎨dataname              ⎬
⎩arithmetic-expression ⎭
```

%NEST

Is an AID keyword which effects output of the current call hierarchy.

For the lowest hierarchical level AID outputs the name of the program unit and the number of the statement where the program was interrupted. For higher hierarchical levels AID outputs the name of the calling program and the number of the CALL statement.

```
---------------------
| medium-a-quantity |
---------------------
```

Defines the medium or media via which output is to take place and whether or not AID is to output additional information. If this operand is omitted and no declaration has been made in the %OUT command, AID assumes the default value T = MAX.

```
medium-a-quantity-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - -

⎧T  ⎫   ⎧MIN  ⎫
⎪H  ⎪   ⎪MAX  ⎪
⎪Fn ⎪ = ⎪XMAX ⎪
⎩P  ⎭   ⎩XFLAT⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-a-quantity is described in detail in the AID Core Manual, chapter 7.

T Terminal output

H Hardcopy output

Fn File output

P Output to SYSLST

MAX Output with additional information

MIN Output without additional information

XMAX Output as with MAX, but extended by the type information:
In addition, each data element is preceded by a type tag which defines the type, size and output format of this data element. Syntax of the type tag:
`<data-type(memory-size-in-bytes),output-format>`

XFLAT Output as with XMAX, but with the following restrictions:
Only the topmost structure level is output for structured data types. In the case of long data (e.g. long strings or arrays), the first elements are output.

Data types

If you have specified the operand value XMAX or XFLAT, AID generates the output as with MAX, extended by the following type tags:

```
<INT(size),D>
int-name = int-value
```

size Storage length in bytes.

int-name Specifies an element of the type integer.

int-value Decimal value (D); value of *int-name*.

```
<POINTER(size),X>
pointer-name = pointer-value
```

size Storage length in bytes.

pointer-name Specifies an element of the type pointer.

pointer-value Hexadecimal number (X); value of *pointer-name*.

```
<FLOAT(size),E>
float-name = float-value
```

size Storage length in bytes.

float-name Specifies an element of the type floating point number.

float-value Floating point number displayed as a decimal fraction with exponent (E); value of *float-name.*

```
<CHARS(size),C>
chars-name = |string|
```

size Storage length in bytes.

chars-name Specifies an element of the type string, in other words an array of the type character.

string String of printable characters (C); value of *chars-name;*
Non-printable characters are displayed as a hexadecimal value.

If *string* is longer than 80 characters, with XFLAT only the first 72 characters are output, followed by three periods ... in order to display the incompleteness of the output. See also note 1 at the end of the list.

```
<UNSIGN(size),D>
unsign-name = unsign-value
```

size Storage length in bytes.

unsign-name Specifies an element of the type integer without a sign (unsigned).

unsign-value Decimal value (D); value of *unsign-name*.

```
<ADDR(size),X>
addr-name = addr-value
```

size Storage length in bytes.

addr-name Specifies an element of a relative or absolute storage address.

addr-value Hexadecimal number (X); value of *addr-name*.

```
<AREA(size),X>
area-name = area-value
```

size Storage length in bytes.

area-name Specifies a primary memory area.

area-value Memory dump in dump format, value of *area-name*. The dump format consists of a hexadecimal (X) and alphanumeric display, non-printable characters are displayed in the alphanumeric display as `|.|`.

If the output is longer than 80 characters, with XFLAT only the first 4 hexadecimal words are output (possibly also fewer). The alphanumeric display contains a maximum of 16 characters (with UTF16: 8 characters) followed by the string ETC.
See also note 1 at the end of the list.

```
<ARRAY(size),type | STRUCT>
array-name (dimension)
(a1) value1 (a2) value2 (a3) value3 ...
```

size Primary memory length in bytes.

type Data type (CHARS, INT, FLOAT,...) if the array consists of a particular data type.

STRUCT The array has a complex structure consisting of various data types.

array-name	Specifies an element of the type array.
dimension	The dimensions of the array.
(*a1*) *value1* (*a2*) *value2* (*a3*) *value3* ...	*a1*, *a2*, *a3*, ... specifies the subelements of the array, *value1*, *value2*, *value3*, ... and their values. The display of the values depends on the particular data type. With XMAX, all subelements are output. With XFLAT, no subelements are output, see also note 1. For details on array areas, see note 2.

```
<STRUCT(size)>
level struct-name
sub-elements
```

size	Storage length in bytes.
level	Level of embedding of the structure or of a structure element (01, 02, 03, etc.). 01 stands for the topmost level.
struct-name	Specifies an element of the type structure.
sub-elements	Further elements which are contained in the structure. With XMAX, all elements are output. With XFLAT, only some of the elements, see section „Structures with XFLAT“. See also note 1 at the end of the list.

Notes

1. Use the following syntax to query the entire content of a string, structure or array distributed over several lines:

   ```
   %SDUMP name {T | H | Fn | P} = {XMAX | MAX}
   ```

2. Use the following syntax to query the content of the array elements within the particular area:

   ```
   %SDUMP name [from:to] {T | H | Fn | P} = {XMAX | XFLAT | MAX}
   ```

Structures with XFLAT

For structures, AID generates various XFLAT data outputs depending on whether or not the %SDUMP command contains data operands.

- %SDUMP without data operand

  ```
  %SDUMP {T | H | Fn | P} = XFLAT
  ```

  Only the type tag and the name are output (level 01). The output of the structure elements is omitted.

- %SDUMP with a structure as operand

  ```
  %SDUMP structure-name {T | H | Fn | P} = XFLAT
  ```

The structure name and the structure elements are output (level 02). Elements with elementary types are normally output, elements with array type with their name, and elements with structure type only with their name. Each element is preceded by a type tag. The name is extended by a number, the level of embedding.

- %SDUMP with a substructure as operand

  ```
  %SDUMP structure-name.substruct-name {T | H | Fn |P} = XFLAT
  ```

  Also outputs the structure elements of the substructure (level 03)

  Further levels of embedding can also be specified by the other substructure names being chained by a period:

  ```
  structure-name.substruct1-name.substruct2-name.substruct3-name. ....
  ```

In order to query the entire content of a structure and of its substructures, use XMAX instead of XFLAT.

Examples

The compiler listing for both examples is given in section 6.1.

1. Using the %SD command as a subcommand of the %INSERT command, a symbolic dump of all program units of the current call hierarchy is requested. All data elements of the program units EXCHANGE, SORT and B1 are output. The default value for *medium-a-quantity* (T=MAX) is used. The subcommand also includes a %STOP. The program therefore remains interrupted after output of all the data, and AID writes a STOP message with the number of the statement and the name of the program unit for the current interrupt point.

```
/LOAD-PROG FROM-FILE=*MOD(LIB=*OMF),TEST-OPT=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'B1' LOADED
/%IN PROG=EXCHANGE.S'5' <%SD; %STOP>
/%R
BS2000 F O R 1 : FORTRAN PROGRAM "B1"
STARTED ON 91-06-28 AT 15:18:33
   CARRAY UNSORTED
Jimmy
Maria
Jamie
Lesly
Jonny
Donna
Marie
Carol
Frank
```

```
** ITN: #000000CB'***TSN:1114************************************'
SRC_REF:     5  SOURCE: EXCHANGE   PROC: EXCHANGE **********************
CHAR            = |Jonny|

SRC_REF:    33  SOURCE: SORT       PROC: SORT  *************************
CARRAY( 1: 9)
( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|

IMIDPT          = |Jonny|

L( 1: 5)
( 1)           1  ( 2)           0  ( 3)          0  ( 4)           0
( 5)           0

R( 1: 5)
( 1)           9  ( 2)           0  ( 3)          0  ( 4)           0
( 5)           0

Z               =           0
LI              =           1
RI              =           9
I               =           5
J               =           9


SRC_REF:    11  SOURCE: B1         PROC: B1  ****************************
CFELD( 1: 9)
( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|

K               =          10
STOPPED AT SRC_REF: 5  SOURCE: EXCHANGE   PROC: EXCHANGE
```

2. The %SD %NEST command is used to request the current call hierarchy:

   ```
   %SD %NEST
   ```

   AID first outputs the number of the statement in the program unit on the lowest hierarchical level on which the program was interrupted. This is followed by output of the numbers of the CALL statements used to exit from the program units on higher hierarchical levels (in this case: the SORT subprogram and main program B1).

   ```
   SRC_REF:     5  SOURCE: EXCHANGE  PROC: EXCHANGE
   *************************

   SRC_REF:    33  SOURCE: SORT      PROC: SORT
   ******************************

   SRC_REF:    11  SOURCE: B1        PROC: B1
   ********************************
   ```

3. `%SDUMP [..] {T | H | Fn | P} = XMAX`

   AID generates the entire output, which is similar to the output with XFLAT. In addition, the output with XMAX contains the entire contents of long strings, arrays and structures.

4. `%SD[UMP] T=XFLAT` generates the following output, for example. When large quantities of data or complex data types are output, the command truncates certain data values or data elements or removes these.

   ```
   <INT(4),D> int_var = 5

   <FLOAT(8),E> float_var = -0.123456789

   <CHARS(1),C> dollar = |$|

   <CHARS(72),C> char_array =
   |This is a string of 72 characters in length ..........................|

   <CHARS(100),C> char_string =
   |These are the first 75 characters of a string wíth 100 characters in
   length.....................|

   <STRUCT(128)> person

   <ARRAY(1280),STRUCT> person_array(1:10)

   <BITINT(1),D> bit_integer = 5
   ```

   For instance, for the selective selection of components of complex data structures:

```
struct { struct {int x; char y;} inner[100];
             struct {float u, v;}  *pointer;
             } outer [10];

%AID low=on
%OUT %SDUMP T=XFLAT

%SD
<ARRAY(8040),STRUCT> outer( 0: 9)

%SD outer
<ARRAY(8040),STRUCT> outer( 0: 9)

%SD outer[5]
<STRUCT(804)> 01 outer( 5)
<ARRAY(800),STRUCT>  02 inner( 0:99)
<POINTER(4),X>       02 pointer = 0103B700

%SD outer[5].inner[15]
<STRUCT(5)> 02 outer.inner( 5, 15)
<INT(4),D>    03  x = -2130463928
<CHARS(1),C>  03  y = |.|

%SD outer[5].inner[15] T=XMAX
<STRUCT(5)> 02 outer.inner( 5, 15)
<INT(4),D>    03  x = -2130463928
<CHARS(1),C>  03  y = |.|

%SD outer[5].inner[15:16]
<ARRAY(16),STRUCT> outer.inner ( 5) ( 15: 16)

%SD outer[5].inner[15:16] T=XMAX
<ARRAY(16),STRUCT> outer.inner ( 5) ( 15: 16)
<STRUCT(5)> 02 inner(15)
<INT(4),D>    03  x = -2130463928
<CHARS(1),C>  03  y = |.|
<STRUCT(5)> 02 inner(16)
<INT(4),D>    03  x = 0
<CHARS(1),C>  03  y = |%|
```

%SET

With the %SET command you transfer the memory contents or AID literals to memory positions in the program which has been loaded. Before transfer, the storage types *sender* and *receiver* are checked for compatibility. The contents of *sender* are matched to the storage type of *receiver*.

– With *sender* you designate a variable or an array element, a logical value, a length, an address, an execution counter, an AID register or an AID literal. *sender* may be either within the virtual memory of the loaded program or in a dump file.

– With *receiver* you designate a variable or an array element, an execution counter or an AID register to be overwritten. *receiver* may only be located within the virtual memory of the program which has been loaded.

Command	Operand
%S[ET]	sender INTO receiver

In contrast to the %MOVE command, AID checks for the %SET command (prior to transfer) whether the storage type of *receiver* is compatible with that of *sender* and whether the contents of *sender* match its storage type. In the event of incompatibility, AID rejects the transfer and outputs an error message.

If *sender* is longer than *receiver*, it is truncated on the left or right, depending on its storage type, and AID issues a warning message. *sender* and *receiver* may overlap. In the case of numeric transfer, *sender* is converted to the storage type of *receiver* if required, and the contents of *sender* are stored in *receiver* with the value being retained. If the value does not fully fit into *receiver*, a warning is issued.

Transfer with the %SET command thus corresponds to the conventions for the FORTRAN assignment statement. The following special rules must, however, be adhered to:
Data elements of the COMPLEX type can only be modified by the %SET command by targeted modification via *dataname* ._REAL and *dataname* ._IMAG, i.e. by altering the real and imaginary parts of the complex number. If a REAL*4 data element is specified as *sender* in the %SET command and a REAL*8 data element as *receiver*, the rightmost 4 bytes of the REAL*8 data element are padded with binary zeros. This may result in inaccuracies as in all cases where conversion of numeric values is involved.
The %SET command is not suitable for emulating the FOR1 statement ASSIGN. %SET always has the effect of an assignment statement, even if *receiver* is a label variable.

Which storage types are compatible and how transfer takes place is shown in the table at the end of the description of the %SET command.

Entry of the command immediately after loading the program is not advisable, as the user cannot address data and statements without an explicit qualification until the program encounters the first executable statement. This is achieved by entering the command sequence:

%INSERT PROG=program-name.program-name

%RESUME

In addition to the operand values described here, you can also use those described in the manual for debugging on machine code level (see [2]).

With `%AID CHECK=ALL` you can activate an update dialog; this dialog shows you the old and new contents of *receiver* prior to transfer and offers the option of aborting the %SET command.

The %SET command does not alter the program state.

```
----------          ------------
| sender |   INTO   | receiver |
----------          ------------
```

For *sender* or *receiver* you may specify a variable, an array element, a complex memory reference, an execution counter or a register. Symbolic constants, addresses and lengths of data elements, logic values and AID literals can only be used as *sender*.

sender may be located either in the virtual memory area of the loaded program (E=VM) or in a dump file; *receiver*, on the other hand, may only be located in the virtual memory area of the loaded program.

```
sender-OPERAND  - - - - - - - - - - - - - - - receiver-OPERAND - - - - - - - - -

⎡          ⎡dataname         ⎤     ⎤
⎢          ⎢L'n'             ⎥     ⎥
⎢[•][qua•] ⎨S'n'             ⎬     ⎥
⎢          ⎢keyword          ⎥     ⎥
⎢          ⎣compl-memref     ⎦     ⎥
⎢                                  ⎢                    ⎡dataname        ⎤
⎨⎡%@⎤          ⎡dataname         ⎤ ⎬ INTO  [•][qua•]  ⎨keyword         ⎬
⎢⎨  ⎬([•][qua•]⎨                 ⎬)⎢                    ⎣compl-memref    ⎦
⎢⎣%L⎦          ⎣compl-memref     ⎦ ⎢
⎢                                  ⎥
⎢%L=(expression)                   ⎥
⎢                                  ⎥
⎣AID-literal                       ⎦
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

qua

A qualification need only be specified if a memory object is not within the current AID work area.

E={VM | Dn} for *sender*

E=VM for *receiver*

need only be specified if the current base qualification (see %BASE command) is not to apply for a data/statement name, source reference or keyword.
sender can be located either in virtual memory or in a dump file, whereas *receiver* must be located in virtual memory.

PROG=program-name

Specified only when addressing a data/statement name or source reference which is not located in the current program unit (see chapter 3).

NESTLEV= level-number

level-number A level number in the current call hierarchy
level-nummer has to be followed by *dataname*.
Specify NESTLEV= *level-number* when you want to address a data name on a certain level in the current call hierarchy. This qualification can only be combined with E=, and not with any other qualification.

dataname

specifies the name of a constant, variable or array element as defined in the source program. Constants can only be used as *sender*.

dataname is an alphanumeric string with up to 15 characters.

You can neither transfer nor overwrite an entire array. You may only transfer or overwrite individual array elements. In order to address an array element, index the name of the array in the same way as in a FORTRAN statement.

array-name (index1[, index2][, ...])

index specifies the position within an array. The number of indexes required for access is the same as in a FORTRAN statement. When multiple indexes are involved, a comma must be used to separate them.
index may be specified as follows:

```
{n                    }
{dataname             }
{arithmetic-expression}
```

L'n'

Specifies a statement name and designates the address of the first executable FORTRAN statement after a statement label.

n is a statement label and has up to 5 digits. Leading zeros must not be specified.

S'n'

Specifies a source reference and designates the address of an executable FORTRAN statement.

n is the number of a statement; see STMT column of compiler listing.

Statement names and source references are address constants and can thus only be specified as *sender*. The address designated with *L'n'* or *S'n'* is transferred.

Example

```
%SET S'5' INTO %0G
```

The address of the statement with number 5 is written to AID register %0G.

By means of *L'n'*-> or *S'n'*-> you designate 4 bytes of machine code at the corresponding address (see AID Core Manual, section 6.4).
Machine instructions can be output by issuing the %DISASSEMBLE command in order to make any length modification that may be required.
With *receiver*, you may use statement names and source references only in connection with the pointer operator (->).

keyword

is a logic value, an execution counter, the program counter or a register. The AID Core Manual, chapter 9, lists the implicit storage types of the keywords.

The two keywords for .TRUE and .FALSE can only be used as *sender*. They can be transferred to any logical variable in the source program.

keyword may only be preceded by a base qualification.

```
%TRUE               Logic value for .TRUE
%FALSE              Logic value for .FALSE

%•subcmdname        Execution counter
```

```
%•                Execution counter of the current subcommand

%PC                 Program counter

%n                  General register, 0 ≤ n ≤ 15
%nD|E               Floating-point register, n = 0,2,4,6
%nQ                 Floating-point register, n = 0,4
%nG                 AID general register, 0 ≤ n ≤ 15
%nDG                AID floating-point register, n = 0,2,4,6
```

compl-memref

The following operations may occur in *compl-memref* (see AID Core Manual, chapter 6):

– byte offset (•)

– indirect addressing (->)

– type modification (%T(dataname), %X, %C, %E, %D, %P, %F, %A)

– length modification (%L(...), %L=(expression), %Ln)

– address selection (%@(...))

With an explicit type or length modification you can match the storage type for *sender* to that of *receiver*. Memory contents which are incompatible with the storage type will nevertheless be rejected by AID even if a type modification is performed (see also AID Core Manual, section 6.8).
Following a byte offset (•) or pointer operation (->), the implicit storage type and original address length are lost. At the calculated address, storage type %X with a length of 4 applies unless the user has made an explicit specification for type and length.
For each operand in a complex memory reference, the assigned memory area must not be exceeded by a byte offset or length modification, otherwise AID will reject the command and issue an error message. By combining address selection (%@) and pointer operator (->) you may exit from the symbolic level. You can then use the address of a data element without regarding its area boundaries.

Example

The CARRAY and CFIELD1 variables are of type ’character’ and occupy 5 bytes each. The last 2 bytes of CARRAY as well as the next 3 bytes are to be transferred to CARRAY1.
AID would reject the command shown below, since it represents a violation of the CARRAY area:

```
%SET CARRAY.3%CL5 INTO CFIELD1
```

The correct command reads:

```
%SET %@(CARRAY)->.3%CL5 INTO CFIELD1
```

%@(...)

The address selector can be used to specify the address of a data element or complex memory reference as *sender* (see also AID Core Manual, section 6.11). The address selector produces an address constant as a result.

%L(...)

The length selector can be used to specify the length of a data element or complex memory reference as *sender* (see also AID Core Manual, section 6.11). The length selector produces an integer as a result.

Example

```
%SET %L(ARRAY1) INTO %OG
```

The length of ARRAY1 will be transferred.

%L=(expression)

With the aid of the length function, you can direct AID to calculate the value of *expression* and store it in *receiver* (see also AID Core Manual, sections 6.9 and 6.10). In *expression* you can link memory references and integers via the arithmetic operators (+,–,*,/). The length function produces an integer as a result.

Example

```
%SET %L=(ARRAY1) INTO %OG
```

The contents of ARRAY1 are transferred. FIELD1 must be of type ’integer’, otherwise AID issues an error message.

AID literal

All AID literals described in the AID Core Manual, chapter 8, may be specified. Note well the conversion options for matching AID literals to the respective *receivers* as described in that chapter:

```
{C'x...x' | 'x...x'C | 'x...x'}      Character literal
{X'f...f' | 'f...f'X}                Hexadecimal literal
{B'b...b' | 'b...b'B}                Binary literal
[{±}]n                               Integer
```

```
#f...f'Hexadecimalnumber'
[{±}]n.m                                Decimal number
[{±}]mantissaE[{±}]exponent             Floating-point number
```

%SET table

The following table provides an overview on permissible combinations of the sender and receiver types in conjunction with the %SET command.

Sender	**Receiver**				
	INTEGER REAL *complex*._REAL *complex*._IMAG %F %P %A %D	COMPLEX	CHARACTER %C	LOGICAL	%X
INTEGER REAL *complex*._REAL *complex*._IMAG %F %P %A %D ±n #'f...f'	num	*	–	–	bin
±n.m ±mantE±exp	num	*	–	–	–
COMPLEX	*	*	–	–	–
CHARACTER %C C'x...x'	num(1)	–	char	–	bin
LOGICAL %TRUE %FALSE	–	–	–	bin	–
%X X'f...f' B'b...b'	bin	–	bin	bin	bin

bin Binary transfer

(left-justified)

sender < *receiver*: padding with binary zeros on the right.

sender > *receiver*: truncation on the right.

When a transfer is made to storage type %X, a numeric literal (only integers are permitted) corresponds to a signed integer value with a length of 4 bytes (%FL4), which are transferred in binary form.

char Character transfer

(left-justified)

sender < *receiver*: padding with blanks (X'40') on the right.

sender > *receiver*: truncation on the right.

num Numeric transfer

(value retained)

sender is matched to the storage type of *receiver* if required.

num(1) If a sender of type 'character' contains digits only and is no

more than 18 digits in length, transfer is in numeric form, provided that the receiver is of the numeric type. Any other character type senders cannot be transferred to numeric receivers.

– no transfer

AID reports that the storage types are incompatible.

* no transfer

AID does not effect transfer, in contrast to FOR1.

Complex values can only be transferred separately as real (*dataname*._REAL) and imaginary (*dataname*._IMAG) portions.

Examples

For the following examples the update dialog was activated via %AID CHECK=ALL. This displays the contents of the receive field before and after the execution of %SET:

1. %SET #'061' INTO COUNTER

```
OLD CONTENT:
      1
NEW CONTENT:
     97
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
```

   The following command produces the same result:

```
%SET 97 INTO COUNTER
```

2. %QUALIFY PROG=SORT
   %SET .LI INTO .L(Z)

```
OLD CONTENT:
      0
NEW CONTENT:
     10
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
```

3. %SET 'ABCDEFG' INTO CHARVAR

```
OLD CONTENT:
|1234567890|
NEW CONTENT:
|ABCDEFG   |
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
```

4. %SET 0.12345E-03 INTO COMPLVAR ._REAL

```
I390 WARNING: SOURCE TRUNCATED
OLD CONTENT:
+.0000000 E+000
NEW CONTENT:
+.1234499 E-003
```

```
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
```

5.
```
%AID SYMCHARS=NOSTD
%SET ARRAY(I*J-K,L) INTO CARRAY ._IMAG(M+3)
```

   The %AID command causes AID to calculate the index I*J-K correctly. Otherwise AID would interpret the expression J-K as the name of a variable.

```
OLD CONTENT:
+.7000000000000000 E+003
NEW CONTENT:
+.8765429999999999 E-003
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
```

%STOP

With the %STOP command you direct AID to halt the program, to switch to command mode and to issue a STOP message. This message indicates the statement and the program unit where the program was interrupted.

If the command is entered at the terminal or from a procedure file, the program state is not altered, since the program is already in the STOP state. In this case you may employ the command to obtain localization information on the program interrupt point by referring to the STOP message.

```
_____________________________________________________________________________
Command          Operand
_____________________________________________________________________________

%STOP

_____________________________________________________________________________
```

If the %STOP command is contained in a command sequence or subcommand, any commands following it will not be executed.

If you set a dump file as a basic qualification with %BASE and then enter a %STOP command, AID outputs a STOP message containing localization information for the address at which the program was interrupted when the dump file was written.

If the program has been interrupted by pressing the K2 key, the program interrupt point need not necessarily be within the user program, it may also be located in the runtime system routines.

The %STOP command alters the program state.

A %STOP in a subcommand always refers to the loaded program.

Example

```
/%IN PROG=SORT.S'20' <%D CARRAY; %STOP>
/%RESUME

CARRAY( 1: 9)
( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
STOPPED AT SRC_REF: 20 , SOURCE: SORT , PROC: SORT
```

%INSERT sets a test point for statement 20. The subcommand comprises the %DISPLAY and %STOP commands. After CARRAY has been output, AID halts the program and writes a STOP message indicating the statement number and program unit of the current interrupt point.

%SYMLIB

With the %SYMLIB command you direct AID to open or close PLAM libraries. AID accesses open PLAM libraries if symbolic memory references located in a program unit for which no LSD records have been loaded are addressed in a command.

– By means of *qualification-a-lib* you open or close one or more libraries in which object modules and their associated LSD records are stored. In order to dynamically load LSD records, any library can be assigned to the current program or to a dump file by specifying the appropriate base qualification.

```
_____________________________________________________________________________
Command           Operand
_____________________________________________________________________________

%SYMLIB           [qualification-a-lib][,...]

_____________________________________________________________________________
```

When this command is executed AID checks only whether the specified library can be opened; it does not check whether the contents of the library match the program being processed. Thus it is possible to initially open all libraries which you might need later during a test run. AID does not check whether the object module of the program which has been addressed matches that of the PLAM library until the dynamically loaded LSD records are accessed.

If several libraries have been opened for a base qualification, AID scans them in the order in which they were specified in the %SYMLIB command.
If the AID search is not successful or if no library is open, you may assign the correct library by way of a new %SYMLIB command after the corresponding message has been issued. You then repeat the command for whose execution the LSD records were lacking.

A library remains open until a new %SYMLIB command is issued for the same base qualification or until it is closed by a %SYMLIB command without operand, or until /LOGOFF. If a new command contains new file names, these libraries are assigned and opened.

The %SYMLIB command does not alter the program state.

```
_______________________
| qualification-a-lib |
_______________________
```

is a base qualification and/or the file name of a PLAM library.

– If you enter a base qualification and a file name, AID assigns the specified library for this base qualification and opens it. Previously assigned libraries for the same base qualification are closed.

– If you specify a file name only, AID assigns the library for the base qualification which is currently applicable (see %BASE command) and opens it. All libraries previously assigned for the current base qualification will be closed.
– If you specify a base qualification only, all open libraries for this qualification will be closed.

AID can handle up to 15 library assignments. A library which is concurrently assigned for several base qualifications is counted as often as it is specified.

```
qualification-a-lib-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - -

      ⌈VM ⌉
[•][E=⎨   ⎬•][filename]
      ⌊Dn ⌋

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command and can only stand for a base qualification.

E=VM

%SYMLIB applies for the loaded program (see also %BASE command).

E=Dn

%SYMLIB applies for a memory dump in a dump file with the link name *Dn* (see %BASE command).

filename

is the BS2000 catalog name of a PLAM library which is assigned for the base qualification specified with *prequalification* or entered explicitly. If the qualification is omitted, the library is assigned for the base qualification which currently applies.

Example

```
%SYMLIB  E=D5.PLAMLIB,FOR1OUTPUT
```

If AID requires LSD records for processing a memory dump in the dump file with the link name D5, AID attempts to load these records from the PLAMLIB library.
The FOR1OUTPUT library is assigned for the currently set base qualification. If no %BASE command has been issued, AID uses this library to dynamically load LSD records for the program being executed.

%TITLE

With the %TITLE command you define the text of your own page header. AID uses this text when the %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE commands write to the system file SYSLST.

– By means of the *page-header* operand you specify the text of the header and direct AID to set the page counter to 1 and to position SYSLST to the top of the page before the next line to be printed.

```
_____________________________________________________________________________
Command           Operand
_____________________________________________________________________________

%TITLE            [page-header]

_____________________________________________________________________________
```

With a %TITLE command without a *page-header* operand you switch back to the AID standard header. AID resets the page counter to 1 and positions SYSLST to the top of the page before the next line to be printed.

A page header defined with %TITLE remains valid until a new %TITLE command is issued or until the program ends.

The %TITLE command does not alter the program state.

```
_______________
| page-header |
_______________
```

Specifies the variable part of the page title. AID completes this specification by adding the time, date and page counter.

page-header

is a character literal in the format {C'x...x' | 'x...x'C | 'x...x'} and may have a maximum length of 80 characters. A longer literal is rejected with an error message outputting only the first 52 positions of the literal.

Up to 58 lines are printed on one page, not counting the title of the page.

%TRACE

With the %TRACE command you switch on the AID tracing function and start the program or continue it at the interrupt point or the point specified in the %JUMP command.

– By means of the *number* operand you can specify the maximum number of FORTRAN statements to be traced, i.e. executed and logged.

– By means of the *continue* operand you control whether the program halts after the %TRACE terminates (default) or continues running without logging.

– By means of the *criterion* operand you select different types of FORTRAN statements which AID is to log. Logging takes place prior to execution of the statements selected.

– By means of the *trace-area* operand you define the program area in which the *criterion* is to be taken into consideration.

```
------------------------------------------------------------------------------
Command           Operand
------------------------------------------------------------------------------

%T[RACE]          [number] [continue] [criterion][,...]  [IN trace-area]

------------------------------------------------------------------------------
```

A %TRACE command is terminated if any of the following five events occurs during the test run:

1. The maximum number of statements to be traced has been reached.
2. A subcommand has been executed because a monitoring condition from a %CONTROLn, %INSERT or %ON command was satisfied, and this subcommand contains a %RESUME, %STOP or %TRACE command.
3. An %INSERT command terminates with a program interrupt, as the *control* operand is K or S.
4. The K2 key has been used. At the terminal, the SDF option `OVERFLOW-CONTROL = USER-ACKNOWLEDGE` (`/MODIFY-TERMINAL-OPTIONS` command) must have been set.
5. The program has been halted by the FORTRAN statement PAUSE.

A %TRACE command which is still active after being interrupted by an event described under points 2 through 5 above may be continued by issuing the %CONTINUE command.

The operand values of a %TRACE command apply until they are overwritten by the entries in a subsequent %TRACE command, or until the program is terminated. In a new %TRACE command, AID therefore assumes the value from the previous %TRACE command if an operand has not been specified. In the case of the *trace-area* operand, this only happens if

the current interrupt point is within the *trace-area* to be assumed. If there are no values to be taken over, AID assumes the default values 10 (for *number*) and the program unit containing the current interrupt point (for *trace-area*).

With the aid of the %OUT command, you can control the information to be contained in a line of the log and the output medium to which the log is to be written.

If the %TRACE is contained in a command sequence or subcommand, any commands which follow will not be executed.

trace-area can only be located in the loaded program, therefore the base qualification E=VM must have been set (see %BASE) or must be specified explicitly.

The %TRACE command alters the program state.

```
----------
| number |
----------
```

specifies the maximum number of FORTRAN statements of type *criterion* which are to be executed and logged.

number

is an integer $1 \leq number \leq 2^{31}-1$. The default value is 10. If there is no value from a previous %TRACE command, AID inserts the default value in a %TRACE command without the *number* operand.

After the specified *number* of statements has been traced, AID outputs a message via SYSOUT, the program is halted and the user can enter AID or BS2000 commands. The message tells you at which statement and in which program unit the program was halted.

```
------------
| continue |
------------
```

Defines whether AID is to halt or continue program execution after the %TRACE terminates. 'continue' applies until a different operand value for it is entered in a new %TRACE or until the program terminates.

```
continue-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{S | R}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

S The program is halted. AID issues a STOP message containing the localization information about the interrupt point. S is the default value.

R The program is continued without a message being issued.

```
-------------
| criterion |
-------------
```

is a keyword which defines the type of statements to be traced during program execution. Several keywords can be specified at a time; they take effect simultaneously. A comma must be used to separate any two keywords.
If no *criterion* is declared, AID uses the default value %STMT unless a *criterion* declaration from an earlier %TRACE command is still valid.

criterion	Logging takes place prior to execution of:
%STMT	Every executable FORTRAN statement
%ASSGN	Assignment statements
%CALL	SUBROUTINE calls (CALL statements)
%COND	IF(...) THEN, ELSE IF(...) THEN, ELSE and IF(...) statements
%GOTO	GOTO statements
%IO	Input/output statements
%LAB	Every statement with a label
%PROC	STOP, END, RETURN statements and the first executable statement following SUBROUTINE and FUNCTION

```
--------------
| trace-area |
--------------
```

defines the program area in which tracing is to take place, i.e. only within this area can monitoring and logging of the statements selected by means of the *criterion* operand be effected. The %TRACE command is inactive outside of this area and is activated again only on returning to this area.

A *trace-area* remains effective until a new %TRACE command with its own *trace-area* operand is entered, until a %TRACE command is issued outside of this area or until the program ends. If the *trace-area* operand has been omitted, the area definition from an earlier %TRACE command is assumed if the current interrupt point is located in this area. Otherwise AID uses the default value, i.e. the program unit containing the current interrupt point.

The continuation address for program execution cannot be influenced by the %TRACE command; such is only possible by means of the %JUMP command.

```
trace-area-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                ⎧PROG=program-name                    ⎫
IN  [•][E=VM•] ⎨                                      ⎬
                ⎩[PROG=program-name•]( S'n' : S'n' ) ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command. Consecutive qualifications must be separated by a period. In addition, there must be a period between the final qualification and the following operand part.

E=VM

As *trace-area* may only be located in the virtual memory of the program which has been loaded, enter *E=VM* only if a dump file has been declared as the current base qualification (see also %BASE command).

PROG=program-name

program-name is the name of a program unit and consists of up to 7 characters.

This program unit must already be loaded at the time the %TRACE command is input.

A PROG qualification is required only if a load module has been created from several program units and the %TRACE command does not refer to the current program unit or if a previously applicable *trace-area* declaration is to be overwritten.

If *trace-area* ends with a PROG qualification, it covers the entire program unit specified.

(S'n' : S'n')

The *trace-area* is defined by specifying a start address and an end address. The start and end addresses must both be within the same program unit and the following must apply:
start address ≤ end address.
n is the number of a statement; see STMT column in compiler listing.

If the *trace-area* is to cover only one statement, the start address and the end address must be identical.

Output of the %TRACE listing

The %TRACE listing is output in full format via SYSOUT as a standard procedure (%OUT operand value T=MAX). With the %OUT command, you can define the output media and the scope of information to be output (see AID Core Manual, chapter 7).

A %TRACE listing with additional information (T=MAX) contains the number and type of the statement that was executed. If a statement label exists, it will be output as well.

A %TRACE listing without additional information (T=MIN) does not show the statement type.

AID does not take into account XMAX and XFLAT modes for outputting the %TRACE log. Instead, it generates the default value (T=MAX).

Examples

1.
```
/%OUT %TRACE    T=MAX
/%T 3
   49         33       STMT
   50                  ASSIGN
   51                  ASSIGN
STOPPED AT SRC_REF:  51,  SOURCE: EXAMPLE, PROC: EXAMPLE
```

With the aid of the %OUT command, output is switched back to the terminal and the maximum range of information is defined for output.
The %TRACE command is to trace three FORTRAN statements. After the third statement the termination message for this %TRACE command follows, to the effect that program execution was interrupted at statement 51, that statement 51 is in the program unit EXAMPLE and that the load module has the same name.

2.
```
/%OUT %T T=MIN
/%T 3
   49         33
   50
   51
STOPPED AT SRC_REF:  51,  SOURCE: EXAMPLE, PROC: EXAMPLE
```

With the %OUT command the range of information for the %TRACE command is reduced. A subsequently entered %TRACE command outputs the log without additional information.

3. `%TRACE 5 R %INSTR`

   5 program commands are executed and logged. After this, the program continues without logging.

4. `%C1 %CALL IN S=TESTPROG <%TRACE 1 R>`

   All subroutine calls by the TESTPROG module are logged. The program continues after each respective CALL instruction is executed and logged.

6 Sample application

This chapter illustrates an AID debugging session for a short FORTRAN program. This sample test is intended to help you understand the application and effect of various AID commands; for the sake of clarity, a relatively uncomplicated approach has been taken. The FORTRAN program is shown first, the test run follows afterwards.

6.1 Source listing

```
PROGRAM UNIT: B1

DO/IF SEG  STMT  I/H LINE      SOURCE-TEXT
-------------------------------------------------------------------------------
  1/1    1          1 |    PROGRAM B1
                    2 |*
                    3 |*       SORTING A CHARACTER ARRAY
                    4 |*
   1     2          5 |    IMPLICIT INTEGER (A-Z)
   1     3          6 |    PARAMETER (DIM=9)
   1     4          7 |    COMMON  /CB/ CARRAY
   1     5          8 |    CHARACTER * 5 CARRAY(DIM)
   1     6          9 |    DATA CARRAY /'Jimmy','Maria','Jamie','Lesly','Jonny',
   1               10 |   &             'Donna','Marie','Carol','Frank'/
   1     7         11 |    WRITE (2,*) '   CARRAY UNSORTED'
   1     8         12 |    DO 10 K=1,DIM
1  2     9         13 |       WRITE (2,*) CARRAY(K)
1  3    10         14 | 10 CONTINUE
   4    11         15 |    CALL SORT
                    16 |*
                    17 |*  OUTPUT FOR CHECKING PURPOSES
                    18 |*
   4    12         19 |    WRITE (2,*) '   CARRAY SORTED'
   4    13         20 |    DO 20 K=1,DIM
1  5    14         21 |       WRITE (2,*) CARRAY(K)
1  6    15         22 | 20 CONTINUE
   7    16         23 |    END
```

```
PROGRAM UNIT: SORT

DO/IF SEG  STMT  I/H LINE      SOURCE-TEXT
-------------------------------------------------------------------------------
                    1 |*
  1/1    1          2 |    SUBROUTINE SORT
                    3 |*
                    4 |*  CARRAY IS SORTED
                    5 |*
   1     2          6 |    IMPLICIT INTEGER (A-Z)
   1     3          7 |    PARAMETER (DIM=9)
   1     4          8 |    COMMON /CB/CARRAY
   1     5          9 |    CHARACTER * 5 CARRAY(DIM), IMIDPT
   1     6         10 |    DIMENSION L(5) ! LEFT INTERVAL END POINTS
   1     7         11 |    DIMENSION R(5) ! RIGHT INTERVAL END POINTS
                   12 |*
   1     8         13 |    Z=1            ! NUMBER OF SUBINTERVALS
   1              14 |                   ! TO BE SORTED
   1     9         15 |    L(1)=1         ! START = TOTAL INTERVAL
   1    10         16 |    R(1)=DIM
                   17 |*
   1    11         18 |  1 CONTINUE   ! SIMULATION OF A "REPEAT LOOP"
   1              19 |               ! THE LOOP IS EXECUTED UNTIL ALL
   1              20 |               ! SUBINTERVALS HAVE BEEN SORTED
   1              21 |               ! WITH RESPECT TO THE INTERVAL MIDPOINT
                   22 |*
                   23 |*
                   24 |*   SORTING A SINGLE SUBINTERVAL WITH RESPECT TO
                   25 |*   THE INTERVAL MIDPOINT
                   26 |*
   2    12         27 |    LI=L(Z)    ! LEFT AND RIGHT INTERVAL END POINTS     (10)
   2    13         28 |    RI=R(Z)    ! OF THE CURRENT SORTING INTERVAL
   2    14         29 |    Z=Z-1      ! DECREASE NUMBER OF INTERVALS
   2               30 |               ! YET TO BE SORTED
   2    15         31 |    IMIDPT=CARRAY(INT(LI+RI)/2)    ! INTERVAL MIDPOINT
                   32 |*
                   33 |* ALGORITHM:
                   34 |*   THE ARRAY TO BE SORTED IS REGARDED AS AN INTERVAL
                   35 |*   AND THE INTERVAL MIDPOINT IS DETERMINED.
                   36 |*   THEN AN ELEMENT LARGER THAN THE INTERVAL MIDPOINT
                   37 |*   VALUE IS SOUGHT IN THE LEFT HALF-INTERVAL,
                   38 |*   STARTING FROM THE LEFT. SIMILARLY, AN ELEMENT
                   39 |*   SMALLER THAN THE INTERVAL MIDPOINT VALUE IS SOUGHT
                   40 |*   FROM THE RIGHT. THESE TWO VALUES ARE INTERCHANGED.
                   41 |*   IF THE INTERVAL MIDPOINT HAS NOT YET BEEN
                   42 |*   REACHED FROM THE LEFT AND RIGHT RESPECTIVELY,
                   43 |*   THIS PROCEDURE CONTINUES.
                   44 |*
                   45 |*   WHEN AN INTERVAL END POINT HAS BEEN REACHED,
                   46 |*   A CHECK IS MADE WHETHER THE ELEMENT OF THE
                   47 |*   INTERVAL MIDPOINT CAN BE SUBSTITUTED FOR AN
                   48 |*   ELEMENT OF THE LEFT OR RIGHT HALF-INTERVAL.
                   49 |*
                   50 |*   THIS PROCEDURE IS NOW FOLLOWED SEPARATELY FOR
                   51 |*   THE LEFT AND RIGHT SUBINTERVALS
                   52 |*   ETC.
                   53 |*   THIS PROCEDURE ENDS WHEN ALL INTERVALS HAVE
                   54 |*   BEEN PROCESSED.
                   55 |*
   2    16         56 |    I=LI  ! SET SEQUENTIAL INDEX FOR LEFT HALF-INTERVAL
   2    17         57 |    J=RI  ! SET SEQUENTIAL INDEX FOR RIGHT HALF-INTERVAL
                   58 |*
   2    18         59 |  2 CONTINUE   ! SIMULATION OF A "REPEAT LOOP"
   2               60 |               ! PROCESSING THE CURRENT INTERVAL
                   61 |*
   2    19         62 |  3  CONTINUE  ! SIMULATION OF A "DO LOOP"
   2               63 |               ! SEEK ELEMENT TO BE SUBSTITUTED ON THE LEFT
   3    20         64 |       IF (CARRAY(I) .GE. IMIDPT) GOTO 31
```

```
                    65 |*
                    66 |*   GOTO 31: ELEMENT MUST BE SUBSTITUTED OR
                    67 |*             INTERVAL MIDPOINT REACHED
                    68 |*
     4     22       69 |         I=I+1
                    70 |*          CHECK NEXT ELEMENT
     4     23       71 |         GOTO 3
     4     24       72 |  31 CONTINUE
                    73 |*
     4     25       74 |    4 CONTINUE  ! SIMULATION OF A "DO LOOP"
     5     26       75 |         IF (CARRAY(J).LE.IMIDPT) GOTO 41                  (1)
                    76 |*
                    77 |*   GOTO 41: ELEMENT MUST BE SUBSTITUTED OR
                    78 |*             INTERVAL MIDPOINT REACHED
                    79 |*
     6     28       80 |         J=J-1
     6     29       81 |         GOTO 4
     6     30       82 |  41 CONTINUE
                    83 |*
     7     31       84 |      IF (I .GE. J)GOTO 21  ! EXIT LOOP                    (2)
                    85 |*
                    86 |*   GOTO 21: NO ELEMENTS ARE TO BE INTERCHANGED
                    87 |*             OR ALL NECESSARY SUBSTITUTIONS HAVE
                    88 |*             BEEN EFFECTED.
                    89 |*
                    90 |*
                    91 |*   SUBSTITUTE ELEMENTS FROM LOWER INTERVAL AREA
                    92 |*   FOR ELEMENT FROM UPPER INTERVAL AREA
                    93 |*
     8     33       94 |         CALL EXCHANGE(CARRAY(I),CARRAY(J))
     8     34       95 |         I=I+1
     8     35       96 |         J=J-1
     8     36       97 |         IF (I .LT. J) GOTO 2  ! UNTIL LOOP
                    98 |*
                    99 |*   GOTO 2 * CHECK FOR FURTHER SUBSTITUTIONS
                   100 |*
                   101 |*   ALL ELEMENTS SUBSTITUTED WITHIN THE INTERVAL
                   102 |*
     8     38      103 |  21 CONTINUE
     9     39      104 |      IF (LI .LT. J) THEN  ! ALWAYS SATISFIED              (3)
                   105 |*       DETERMINE SUBINTERVAL AND STORE IF APPLICABLE
   1 10    40      106 |         IF (LI .EQ. J-1) THEN                             (4)
                   107 |*   SUBINTERVAL CONSISTS OF TWO ELEMENTS ONLY
   1 10            108 |*
   2 11    41      109 |         IF (CARRAY(LI).GT.CARRAY(J)) THEN
                   110 |*
                   111 |*   SUBSTITUTE ELEMENT OF INTERVAL MARGIN
                   112 |*
   3 12    42      113 |           CALL EXCHANGE (CARRAY(J),CARRAY(LI))
   3 12    43      114 |         ENDIF
   2 12    44      115 |      ELSE
                   116 |*
                   117 |*   STORE SUBINTERVAL NOT YET PROCESSED
                   118 |*
   2 13    45      119 |           Z=Z+1                                           (5)
   2 13    46      120 |           L(Z)=I                                          (6)
   2 13    47      121 |           R(Z)=J                                          (7)
   2 13    48      122 |        ENDIF
   1 13    49      123 |      ENDIF
     14    50      124 |      IF (I .LT. RI) THEN                                  (8)
                   125 |*   DETERMINE SUBINTERVAL AND STORE IF APPLICABLE
   1 15    51      126 |         IF (I .EQ. RI-1) THEN
                   127 |*   SUBINTERVAL CONSISTS OF TWO ELEMENTS ONLY
   2 16    52      128 |           IF (CARRAY(I) .GT. CARRAY(RI)) THEN
                   129 |*
                   130 |*   SUBSTITUTE ELEMENT OF INTERVAL MARGIN
                   131 |*
   3 17    53      132 |               CALL EXCHANGE (CARRAY(I),CARRAY(RI))
```

```
3 17    54        133 |          ENDIF
2 17    55        134 |        ELSE
                  135 |*
                  136 |*   STORE SUBINTERVAL NOT YET PROCESSED
                  137 |*
2 18    56        138 |           Z=Z+1
2 18    57        139 |           L(Z)=I
2 18    58        140 |           R(Z)=RI
2 18    59        141 |        ENDIF
1 18    60        142 |      ENDIF
  19    61        143 |    IF (Z .NE. 0) GOTO 1  ! UNTIL SIMULATION          (9)
                  144 |*
                  145 |*   INTERVAL ARRAY (=STACK SUBSTITUTE) PROCESSED
                  146 |*
  20    63        147 |    RETURN
  20    64        148 |    END
```

6.2 Test run

Step 1

The FORTRAN source program B1 in the file QSORT is compiled using FOR1. Specification of the SDF option TOOL-SUPPORT = AID causes FOR1 to generate LSD information as a prerequisite for symbolic testing. To facilitate testing with AID, the first compilation is undertaken without optimization (SDF option OPTIMIZATION = NO; see chapter 2). The program is compiled without errors.
In the examples below, input is printed in bold for better legibility.

```
/START-FOR1-COMPILER SOURCE=QSORT,OPTIMIZATION=NO,-
             TEST-SUPPORT=PARAMETER(TOOL-SUPPORT=AID),-
             LISTING=PARAMETER(OUTPUT=LF.QSORT),-
             SOURCE-PROPERTIES=PARAMETER(LINE-END-COMMENTS='!')
%  BLS0500 PROGRAM 'FOR1', VERSION '2.1A00' OF '91-04-29' LOADED.
  FOR1:   V2.1A00 READY, GIVE COMPILER OPTION
  FOR1: LIST FILE REPLACED = LF.QSORT
  FOR1: NO ERRORS DURING COMPILATION OF P.U. B1
  FOR1: NO ERRORS DURING COMPILATION OF P.U. SORT
  FOR1: NO ERRORS DURING COMPILATION OF P.U. EXCHANGE

 END OF  F O R 1  COMPILATION; CPU TIME USED: 3.904 SEC.
```

Step 2

The program likewise runs without errors. However, the result of the sort algorithm is not correct: the list of names is not output in alphabetical order.

```
/SET-TASKLIB LIBRARY=$FOR1MODLIBS
/START-FOR1-PROGRAM FROM-FILE=*MODULE(LIBRARY=*OMF)
 %  BLS0001 DLL VER 823
 %  BLS0517 MODULE 'B1' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "B1"
 STARTED ON 91-04-29 AT 16:10:53
    CARRAY UNSORTED
 Jimmy
 Maria
 Jamie
 Lesly
 Jonny
 Donna
 Marie
 Carol
 Frank
    CARRAY SORTED
 Jimmy
 Maria
 Jamie
 Lesly
 Frank
 Donna
 Marie
```

```
 Carol
 Jonny
 BS2000  F O R 1 : FORTRAN PROGRAM "B1           " ENDED PROPERLY AT 16:11:04
 CPU - TIME USED:      0.0937 SECONDS
 ELAPSED TIME   :     11.4430 SECONDS
```

Step 3

To symbolically test the program with AID, it is loaded using the SDF option TEST-OPTIONS=AID. After the LOAD-PROGRAM command, AID commands may be entered.

```
/LOAD-PROGRAM FROM-FILE=*MODULE(LIBRARY=*OMF),-
      TEST-OPTIONS=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'B1' LOADED
```

Step 4

```
/%C1 %CALL IN PROG=SORT <%D I,J,IMIDPT,CARRAY,RI; %STOP>
```

The %CONTROL command declares the CALL statement as the *criterion*, which is to be monitored only in subprogram SORT. The subcommand is to be executed prior to each execution of the CALL statement. The subcommand is to output the sequential indexes I and J for the left and right interval end points respectively, the value of the middle array element IMIDPT, the array CARRAY to be sorted, and the right interval end point RI of the current sorting interval. Following output, the program run is to be interrupted so that AID commands can be entered.

```
/%IN PROG=SORT.S'18' <%D I,J,IMIDPT,CARRAY,LI,RI; %STOP>
```

The %INSERT command is used to set a test point for statement number 18 which initiates processing of the current sorting interval. Data elements I, J, IMIDPT, CARRAY, LI and RI are to be output prior to each execution of the CONTINUE statement.

Step 5

The %RESUME command starts the loaded program. AID reports the IF statement with number 20 as the interrupt point instead of the statement with the number 18 specified in the %INSERT command, since the CONTINUE statement is used merely as a dummy statement here.

```
/%R
BS2000  F O R 1 : FORTRAN PROGRAM "B1"
STARTED ON 91-04-29 AT 09:43:28
   CARRAY UNSORTED
Jimmy
Maria
Jamie
Lesly
Jonny
Donna
Marie
Carol
Frank
** ITN: #000000DF'***TSN:1627*****************************************'
SRC REF:     20 SOURCE: SORT      PROC: SORT ******************************
I               =           1
J               =           9
IMIDPT          = |Jonny|
CARRAY ( 1: 9)
( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
LI              =           1
RI              =           9

STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT
```

Step 6

The AID commands %R "1" through %R "4" respectively are used to resume the program. The quotes stand for start/end of comment. The program checks each time whether the array element CARRAY(I) is greater than or equal to the value of the interval midpoint IMIDPT. Following the comparison, sequential index I for the left half-interval is incremented by 1 in each case. After execution of the AID command %R "4", I has the value 5.

```
/%R "1"
 I               =           2
 J               =           9
 IMIDPT          = |Jonny|
 CARRAY ( 1: 9)
```

```
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT

/%R "2"
 I                =            3
 J                =            9
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT
/%R "3"
 I                =            4
 J                =            9
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT

/%R "4"
 I                =            5
 J                =            9
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT
```

Step 7

The program run is resumed with %R "5". The comparison (CARRAY(I).GE.IMIDPT) leads to the statement with label 31, the comparison (CARRAY(J).LE.IMIDPT) leads to the statement with label 41 and to the invocation of subprogram EXCHANGE with statement number 33. Prior to execution of this CALL statement, the program is interrupted due to the %C1 command and the associated subcommand is executed.

```
/%R "5"
SRC REF:     33 SOURCE: SORT      PROC: SORT ********************************
 I                =            5
 J                =            9
```

```
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 RI               =            9

 STOPPED AT SRC REF: 33 , SOURCE: SORT, PROC: SORT
```

Step 8

After the program has been resumed with %R "6", I is incremented by 1 and J is decremented by 1. These values are then used for a new comparison with the interval midpoint. Upon the subsequent %RESUME commands %R "7" through %R "9", I is incremented by 1 in each case.

```
/%R "6"
SRC_REF:     20 SOURCE: SORT      PROC: SORT ********************************
 I                =            6
 J                =            8
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT

/%R "7"
 I                =            7
 J                =            8
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT

/%R "8"
 I                =            8
 J                =            8
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =            1
 RI               =            9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT

/%R "9"
 I                =            9
 J                =            8
```

```
 IMIDPT           = |Jonny|
 CARRAY ( 1: 9)
 ( 1) |Jimmy|  ( 2) |Maria|  ( 3) |Jamie|  ( 4) |Lesly|  ( 5) |Jonny|
 ( 6) |Donna|  ( 7) |Marie|  ( 8) |Carol|  ( 9) |Frank|
 LI               =           1
 RI               =           9

 STOPPED AT SRC REF: 20 , SOURCE: SORT, PROC: SORT
```

Step 9

CARRAY(I)=IMIDPT is true for I=9. The %TRACE command is now to be used to execute and log the next 8 statements. Output includes the statement numbers, any statement labels and the statement type. Program execution can be monitored on the basis of the %TRACE and source listings.

```
/%T 8 %STMT IN PROG=SORT
    26  4                    IF                    (1)
    31  41                   IF                    (2)
    39  21                   IF                    (3)
    40                       IF                    (4)
    45                       THEN/ELSE, ASSIGN     (5)
    46                       ASSIGN                (6)
    47                       ASSIGN                (7)
    50                       IF                    (8)
 STOPPED AT SRC REF: 50, SOURCE: SORT, PROC: SORT
```

The program first branches to IF statement (1) with the query (CARRAY(J).LE.IMIDPT), and from there to IF statement (2) with the query (I.GE.J). I is greater than J, which triggers a branch to the CONTINUE statement with label 21. The %TRACE listing exhibits statement label 21 and the statement number of the subsequent IF statement [see (3)]. The program then branches to IF statement (4) with the query (LI.EQ.J-1) and executes the assignment statements Z=Z+1, L(Z)=I and R(Z)=J [see (5), (6) and (7)]. The last statement to be logged is the IF statement (8) with the query (I.LT.RI).

Step 10

The AID command %T 2 corresponds to the command %T 2 %STMT IN PROG=SORT. The default for *criterion* (type of statement) is %STMT, the default for *trace-area* is the program unit containing the current interrupt point. Command %T 2 causes statement 61 [see (9)] to be logged and the sorting algorithm to be executed anew with the updated interval end points [see (10)].

```
/%T 2
    61                       IF                    (9)
    12  1                    ASSIGN                (10)

 STOPPED AT SRC REF: 12, SOURCE: SORT, PROC: SORT
```

Step 11

The %DISPLAY command outputs the number Z of subintervals to be sorted and the end points of the subinterval which has not yet been processed.

```
/%D Z
SRC_REF:     12 SOURCE: SORT      PROC: SORT ********************************
 Z                =           1

/%D L(1),R(1)
 L( 1)            =           9
 R( 1)            =           8
```

Following these intermediate results, the left interval end point is greater than the right end point. The statement with number 46 must read L(Z)=LI and not L(Z)=I.

Step 12

The errored statement can be corrected via the %SET command without recompiling the program. For this purpose, the program is reloaded:

```
/LOAD-PROGRAM FROM-FILE=*MODULE(LIBRARY=*OMF),-
      TEST-OPTIONS=AID
%  BLS0001 DLL VER 823
%  BLS0517 MODULE 'B1' LOADED
```

Step 13

The test point of the %INSERT command is set at statement 47 so that element L(Z) is overwritten with the correct value LI after execution of the invalid statement L(Z)=I. A test point for S’46’ would cause the program, prior to execution of the statement, to execute the subcommand and insert the correct value LI; then the invalid statement L(Z)=I would be executed. Correction with the aid of the %SET command causes the program to output a correctly sorted result:

```
/%INSERT PROG=SORT.S'47' <%SET LI INTO L(Z); %RESUME>
/%R
BS2000  F O R 1 : FORTRAN PROGRAM "B1"
STARTED ON 91-04-29 at 09:47:47
   CARRAY UNSORTED
Jimmy
Maria
Jamie
Lesly
Jonny
```

```
Donna
Marie
Carol
Frank
   CARRAY SORTED
Carol
Donna
Frank
Jamie
Jimmy
Jonny
Lesly
Maria
Marie
BS2000  F O R 1 : FORTRAN PROGRAM "B1" ENDED PROPERLY AT 09:47:58
CPU - TIME USED :       0.2067 SECONDS
ELAPSED TIME    :      11.7150 SECONDS
```

Glossary

address operand

This is an operand used to address a memory location or memory area. The operand may specify virtual addresses, data names, statement names, source references, keywords, complex memory references or a PROG qualification. The memory location or area is located either in the program which has been loaded or in a memory dump in a dump file. To address a data element, statement name or source reference which is not located in the current program unit, the user must employ a qualification to reference the relevant position in memory.

AID input files

AID input files are files which AID requires to execute AID functions, as distinguished from input files which the program requires. AID processes disk files only. AID input files include:

1. Dump files containing memory dumps (%DUMPFILE)
2. PLAM libraries containing object modules. If the library has been assigned with the aid of the %SYMLIB command, AID is able to load the LSD records.

AID literals

AID provides the user with both alphanumeric and numeric literals (see AID Core Manual, chapter 8):

```
{C'x...x' | 'x...x'C | 'x...x'}      Character literal
{X'f...f' | 'f...f'X}                Hexadecimal literal
{B'b...b' | 'b...b'B}                Binary literal
[{±}]n                               Integer
#f...f'Hexadecimalnumber'
[{±}]n.m                             Decimal number
[{±}]mantissaE[{±}]exponent          Floating-point number
```

AID output files

AID output files are files to which the user can direct output of the %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE commands. The files are addressed via their link names (F0 through F7) in the output commands (see %OUT and %OUTFILE). The REP records are written to the file assigned to link name F6 (see %AID REP=YES and %MOVE).

There are three ways of creating an output file:

1. /%OUTFILE command with link name and file name
2. /FILE command with link name and file name
3. For a link name to which no file name has been assigned, AID issues
4. a FILE macro with the file name AID.OUTFILE.Fn.

An AID output file always has the format FCBTYPE=SAM, RECFORM=V and OPEN=EXTEND.

AID standard work area

In conjunction with debugging on machine code level, the AID standard work area is the non-privileged part of virtual memory (in the user task) which is occupied by the program and all its connected subsystems.
In conjunction with symbolic debugging, the AID standard work area is the current program unit of the program which has been loaded. If no presetting has been made with the %BASE command and no base qualification is specified, the AID standard work area applies by default.

AID work area

The AID work area is the address area in which the user may reference addresses without having to specify a qualification.
In symbolic debugging, the AID work area is the current program unit. Only the data/statement names and source references within the current program unit can be addressed without a qualification. In the case of the loaded program, the current program unit is the one currently executing. In the case of a memory dump, the current program unit is the one which was executing when the memory dump took place.
You may deviate from the AID work area in a command by specifying a qualification in the address operand. Using the %BASE command, you can shift the AID work area from the loaded program to a memory dump, or vice versa.

area check

In the case of byte offset, length modification and the *receiver* of a %MOVE, AID checks whether the area limits of the referenced memory objects are exceeded and issues a corresponding message if necessary.

area limits

Each memory object is assigned a particular area, which is defined by the address and length attributes in the case of data names and keywords. For virtual addresses, the area limits are between V'0' and the last address in virtual memory (V'7FFFFFFF'). In PROG qualifications, the area limits are determined by the start and end addresses of the program unit (see AID Core Manual, chapter 6).

attributes

Each memory object has up to six attributes:
address, name (opt), content, length, storage type, output type.
Selectors can be used to access the address, length and storage type. Via the name, AID finds all the associated attributes in the LSD records so they can be processed accordingly.
Address constants and constants from the source program have only up to five attributes:
name (opt), value, length, storage type, output type.
They have no address. When a constant is referenced, AID does not access a memory object but merely inserts the value stored for the constant.

base qualification

The base qualification is the qualification the user employs to place the AID work area in the loaded program or in a memory dump in a dump file. The specification is made using E={VM | Dn}.
The base qualification can be declared globally with %BASE or specified explicitly in the address operand for a single memory reference.

command mode

In the AID documentation, the term "command mode" designates the EXPERT mode of the SDF command language. Users working in a different mode (`GUIDANCE={MAXIMUM|MEDIUM|MINIMUM|NO}`) and wishing to enter AID commands should switch to EXPERT mode via `MODIFY-SDF-OPTIONS GUIDANCE=EXPERT`.
AID commands are not supported by SDF syntax:

– Operands are not queried via menus.
– If an error occurs, AID issues an error message but does not offer a correction dialog.

In EXPERT mode, the system prompt for command input is "/".

command sequence

Several commands are linked to form a sequence via semicolons (;). The sequence is processed from left to right. A command sequence may contain both AID and BS2000 commands, like a subcommand. Commands not permitted in a command sequence are the AID commands %AID, %BASE, %DUMPFILE, %HELP, %OUT and %QUALIFY as well as the BS2000 commands listed in the appendix of the AID Core Manual.
If a command sequence contains one of the commands for runtime control, the command sequence is aborted at that point and the program is started (%CONTINUE, %RESUME, %TRACE) or halted (%STOP). As a result, any commands which follow as part of the command sequence are not executed.

constant

A constant represents a value which cannot be accessed via an address in program memory.
Constants include the symbolic constants defined in the source program, the results of length selection, length function and address selection, and the statement names and source references.
An address constant represents an address. Address constants include statement names, source references and the result of an address selection. They can be used, in conjunction with a pointer operator (->), to address the corresponding memory location.

CSECT information

is contained in the object structure list.

current call hierarchy

The current call hierarchy represents the status of subprogram nesting at the interrupt point. It ranges from the subprogram level on which the program was interrupted to the subprograms exited by CALL statements (intermediate levels) to the main program.
The hierarchy is output using the %SDUMP %NEST command.

current program

The current program is the one loaded in the task in which the user enters AID commands.

current program unit

The current program unit is the unit in which the program was interrupted. Its name is output in the STOP message.

data element

Data element is a collective term for all data which can be defined in FORTRAN.

dataname

This operand stands for all names assigned for data in the source program. With the aid of *dataname* the user addresses variables, constants and arrays during symbolic debugging. Array elements can be addressed via an index as in FORTRAN.

data type

In accordance with the data type declared in the source program, AID assigns an AID storage type to each data element:

– binary string (≙ %X)
– character (≙ %C)

– numeric (≙ %F, %D)

This storage type determines how the data element is output by %DISPLAY, transferred or overwritten by %SET, and compared in the condition of a subcommand.

ESD

The External Symbol Dictionary (ESD) lists the external references of a module. It is generated by the compiler and contains, among other items, information on CSECTs, DSECTs and COMMONs. The linkage editor accesses the ESD when it creates the object structure list.

global settings

AID offers commands facilitating addressing, saving input efforts and enabling the behavior of AID to be adapted to individual requirements. The presettings specified in these commands continue to apply throughout the debugging session (see %AID, %AINT, %BASE and %QUALIFY).

index

The index is part of an address operand and permits the position of an array element to be defined. It can be specified in the same way as in FORTRAN or by means of an arithmetic expression from which AID calculates the index value.

input buffer

AID has an internal input buffer. If this buffer is not large enough to accommodate a command input, the command is rejected with an error message identifying it as too long. If fewer of the repeatable operands are specified, the command will be accepted.

interrupt point

The interrupt point is the address at which a program has been interrupted. From the STOP message the user can determine both the address at which and the program unit in which the interrupt point is located. The program is continued at this point. A different continuation address can be specified with the aid of the %JUMP command (FOR1 and COBOL85 only).

LIFO

Stands for the "last in, first out" principle. If statements from different entries concur at a test point (%INSERT) or upon occurrence of an event (%ON), the ones entered last are processed first (see AID Core Manual, section 5.4).

localization information

%DISPLAY %HLLOC(memref) for the symbolic level and %DISPLAY %LOC(memref) for the machine code level cause AID to output the static program nesting for a given memory location.
Conversely, %SDUMP %NEST outputs the dynamic program nesting, i.e. the call hierarchy for the current program interrupt point.

LSD

The List for Symbolic Debugging (LSD) is a list of the data/statement names defined in the module. It also contains the compiler-generated source references. The LSD records are created by the compiler. AID uses them to fetch the information required for symbolic addressing.

memory object

A memory object is formed by a set of contiguous bytes in memory. At program level, this comprises the program data (if it has been assigned a memory area) and the instruction code. Other memory objects are all the registers, the program counter, and all other areas that can only be addressed via keywords. Conversely, any constants defined in the program, as well as statement names, source references, the results of address selection, length selection and length function, and the AID literals do not constitute memory objects because they represent a value that cannot be changed.

memory reference

A memory reference addresses a memory object. Memory references can either be simple or complex.
Simple memory references include virtual addresses, names whose address AID fetches from the LSD information, and keywords. Statement names and source references are allowed as memory references in the AID commands %CONTROLn, %DISASSEMBLE, %INSERT, %JUMP and %REMOVE although they are merely address constants.
Complex memory references instruct AID how to calculate a particular address and which type and length are to apply. The following operations are possible here: byte offset, indirect addressing, type modification, length modification, address selection.

monitoring

%CONTROLn, %INSERT and %ON are monitoring commands. When the program reaches a statement of the selected group (%CONTROLn) or the defined program address (%INSERT), or if the declared event occurs (%ON), program execution is interrupted and AID processes the specified subcommand.

name range

This comprises all data names stored for a program unit in the LSD records.

object structure list

On the basis of the External Symbol Dictionary (ESD), the linkage editor generates the object structure list, provided the default SYMTEST=MAP applies or the user has entered SYMTEST=ALL.

output type

This is an attribute of a memory object and determines how AID outputs the memory contents. Each storage type has its corresponding output type. The AID Core Manual, chapter 9, lists the AID-specific storage types together with their output types. This assignment also applies for the data types used in FORTRAN. A type modification in %DISPLAY and %SDUMP causes the output type to be changed as well.

program state

AID makes a distinction between three program states which the program being tested may assume:

1. The program has stopped.
   %STOP, the K2 key, a PAUSE statement or completion of a %TRACE interrupted the program. The task is in command mode. The user may enter commands.
2. The program is running without tracing.
   %RESUME started or continued the program. %CONTINUE does the same, with the exception that any active %TRACE is continued.
3. The program is running with tracing.
   %TRACE started or continued the program. The program sequence is logged in accordance with the declarations made in the %TRACE command. %CONTINUE has the same effect if a %TRACE is still active.

program unit

A FORTRAN program is made up of individual program units. A program unit is actually a series of program lines which is terminated by an END statement. A distinction is made between main programs and subprograms. In a subprogram, the first statement is a SUBROUTINE, FUNCTION or BLOCK DATA statement; BLOCK DATA program units cannot be addressed using AID commands. In a main program, the first statement is, as a rule, a PROGRAM statement, although any other FORTRAN statement is likewise permissible as the first statement.

qualification

A qualification is used to reference an address which is not in the AID work area or not uniquely defined therein. The base qualification specifies whether the address is in the loaded program or in a memory dump. The PROG qualification specifies the program unit in which the address is situated.
If a qualification is found to be superfluous or contradictory, it will be ignored. This is the case, for example, if a PROG qualification is specified for a data element of the current program unit.

source reference

A source reference designates an executable statement and is specified via *S'n'*. *n* is the number of a statement; it is created by the compiler and can be found in the compiler listing under the STMT column.
S'n' source references, just like *L'n'* statement names, are address constants.

statement name

This designates the first executable FORTRAN statement following a statement label. The corresponding specification is *L'n'*, where *n* is a source statement label (up to 5 digits) assigned by the programmer. Leading zeros must not be specified.
L'n' statement names, just like *S'n'* source references, are address constants.

storage type

This is either the data type defined in the source program or the one selected by way of type modification. AID knows the storage types %X, %C, %E, %P, %D, %F and %A (see AID Core Manual, chapters 6 and 9).

subcommand

A subcommand is an operand of the monitoring commands %CONTROLn, %INSERT or %ON. A subcommand can contain a name, a condition and a command part. The latter may comprise a single command or a command sequence. It may contain both AID and BS2000 commands. Each subcommand has an execution counter. Refer to the AID Core Manual, chapter 5, for information on how an execution condition is formulated, how the names and execution counters are assigned and addressed, and which commands are not permitted within subcommands.
The command part of the subcommand is executed if the monitoring condition (*criterion*, *test-point*, *event*) of the corresponding command is satisfied and any execution condition defined has been met.

tracing

%TRACE is a tracing command, i.e. it can be used to define the type and number of statements to be logged. Program execution can be viewed on the screen as a standard procedure.

update dialog

The update dialog is initiated by means of the %AID CHECK=ALL command. It goes into effect when the %MOVE or %SET command is executed. During the dialog, AID queries whether updating of the memory contents really is to take place. If N is entered in response, no modification is carried out; if Y is entered, AID will execute the transfer.

user area

This is the area in virtual memory which is occupied by the loaded program and all its connected subsystems. It corresponds to the area represented by the keyword %CLASS6 (or %CLASS6ABOVE and %CLASS6BELOW).

Related publications

You will find the manuals on the internet at *http://manuals.ts.fujitsu.com*. You can order printed copies of those manuals which are displayed with an order number.

[1] **AID** (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

[2] **AID** (BS2000)
Advanced Interactive Debugger
Debugging on Machine Code Level
User Guide

[3] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of COBOL Programs
User Guide

[4] **AID** (BS2000)
Advanced Interactive Debugger
Debugging under POSIX
User Guide

[5] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of ASSEMBH Programs
User Guide

[6] BS2000
Executive Macros
User Guide

[7] BS2000
Programmiersystem
Technische Beschreibung
(Programming System, Technical Description)

[8] **FOR1** (BS2000)
FORTRAN Compiler
User's Guide

[9] **FOR1** (BS2000)
FORTRAN Compiler
Reference Manual

Index