English

FUJITSU Software BS2000

AID V3.4B

Debugging of ASSEMBH 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 control 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.

AID - Debugging on Machine Code Level [2]

AID - Debugging of COBOL Programs [3]

AID - Debugging of FORTRAN Programs [4]

AID - Debugging of ASSEMBH Programs

AID - Debugging under POSIX [5]

AID - Debugging of C/C++ Programs [6]

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 or Assembler.

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 In the body of the text, operands are shown in *lowercase italics*.

ℹ This symbol marks points to be specially noted.

2 Prerequisites for symbolic debugging

The user can control generation of the LSD records AID requires for symbolic debugging by specifying the operands described below; these operands must be specified for compiling, linking and loading operations. A more detailed description of these operands is given in the "ASSEMBH User Guide" [9].

2.1 Assembly

The ASSEMBH Assembler can be controlled in two ways:

- via SDF options or
- via COMOPT statements.

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

SDF control

```
                                   ⎧NO  ⎫
/START-ASSEMBH ....,TEST-SUPPORT = ⎨    ⎬
                                   ⎩YES ⎭
```

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

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

COMOPT control

```
/START-ASSEMBHC
*...
          ⎧NOISD⎫
*COMOPT = ⎨-----⎬
          ⎩ISD  ⎭
```

NOISD No LSD records are generated.

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

Example

```
/START-ASSEMBH
//COMPILE SOURCE = SOURCE.TEST,
       TEST-SUPPORT = YES,
                   MODULE-LIBRARY = PROGRAMLIB
```

An object module is to be generated when compiling the source program SOURCE-TEST. The object module is written directly in the PLAM-library PROGRAMLIB.

If COMOPT control is used, the example reads as follows:

```
/DELETE-SYSTEM-FILE FILE-NAME = OMF
/START-ASSEMBHC
**COMOPT SOURCE=SOURCE.TEST
**COMOPT ISD
**COMOPT MODULE=PROGRAMLIB
**END HALT
```

2.2 Linking, loading and starting

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, they are same for all programming languages.

The LSD information generated by Assembler ASSEMBH must be forwarded to the dynamic binder loader DBL to permit symbolic debugging.

You link, load and start ASSEMBH programs with the SDF commands described in chapter 3 of the AID Core Manual which are valid for all languages.

3 ASSEMBH-specific addressing

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

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, and under the %BASE command. A base qualification can be immediately followed by a data name, statement name, source reference, keyword or complex memory reference.

PROG=program-name

In ASSEMBH, the user can employ the PROG qualification as the area qualification, where *program-name* designates a program unit from an ASSEMBH program. *program-name* is the name specified in a START or CSECT 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 Assembler, and may be subjected to all the operations described in the AID Core Manual.
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

is the name of constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and common control sections defined in the source program.

Data defined in the source program via a dummy section, an external dummy section or a dummy register (see DSECT, XDSEC and DXD statements in the ASSEMBH Reference Manual [10]) can only be referenced by means of a pointer operation when symbolic debugging takes place.

dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM Assembler statement (see ASSEMBH Reference Manual [10]).

dataname can be specified in all commands for displaying and modifying data, i.e. %DISPLAY, %MOVE, %SDUMP and %SET, and in the %FIND command (find a character string).

```
dataname - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 ⎧name                                                                    ⎫
 ⎪_Rn                                                                     ⎪
 ⎪                                                                        ⎪
 ⎪                 ⎧dsect-name                   ⎫                        ⎪
 ⎪                 ⎪[dsect-name•.]name           ⎪                        ⎪
 ⎪⎧_Rn    ⎫        ⎪xdsec-d-name                 ⎪                        ⎪
 ⎨⎨       ⎬  ->   ⎨[xdsec-d-name•.]name          ⎬                        ⎬
 ⎪⎩c-name ⎭        ⎪xdsec-r-name                 ⎪                        ⎪
 ⎪                 ⎪[xdsec-r-name•.]name         ⎪                        ⎪
 ⎪                 ⎩dxd-name                     ⎭                        ⎪
 ⎪                                                                        ⎪
 ⎪com-name                                                                ⎪
 ⎩[com-name•.]name                                                        ⎭

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

name

is the name entry of a DC, DS, EQU or CSECT statement.

– Names of DC or DS statements are used to reference the relevant memory contents. AID specifies the memory contents in the data type and length as defined in the source program.

– Names of EQU statements are used to reference either the allocated value or the memory contents at the relevant address. Output is effected in acordance with the length attribute of the EQU name as efined by the length attribute.

– Names of CSECT statements are used to reference the start address or continuation address of a control section.

_Rn

Predefined name for a general register. If this name is specified, AID outputs the contents of the related register. _Rn corresponds to the AID keyword %n.
n   *i*s a number in the range 0 ≤ n ≤ *1*5

Dummy section (DSECT statement)

```
⎧_Rn    ⎫     ⎧dsect-name          ⎫
⎨       ⎬ ->  ⎨                    ⎬
⎩c-name ⎭     ⎩[dsect-name•.]name  ⎭
```

_Rn

specifies the base address register of the dummy section.

If the dummy section was referenced via a Q constant, *_Rn* defines the start address of the dummy register vector (see the ASSEMBH Reference Manual [10]).

c-name

is the name of an A, Y or V constant whose contents represent the base address of the dummy section.
If the dummy section was referenced via a Q constant, *c-name* defines the start address of the dummy register vector (see the ASSEMBH Reference Manual [10]).

dsect-name

is the name of the dummy section.

This addresses all named fields of the dummy section in accordance with their type and sorted by addresses in ascending order.

[dsect-name.]name

dsect-name is the name of the dummy section.

name

is the name of an individual field within the dummy section.

Aid interprets the field according to the type and length attributes.

Definition of an external dummy section (XDSEC D)

```
⎧_Rn    ⎫    ⎧xdsec-d-name             ⎫
⎨       ⎬ -> ⎨                         ⎬
⎩c-name ⎭    ⎩[xdsec-d-name•.]name     ⎭
```

_Rn

specifies the base address register of the external dummy section.

c-name

is the name of an A, Y or V constant whose contents represent the base address of the external dummy section (see the ASSEMBH Reference Manual [10]).

xdsec-d-name

is the name of the external dummy section.

This addresses all named fields of the dummy section in accordance with their type and sorted by addresses in ascending order.

[xdsec-d-name•]name

xdsec-d-name is the name of the external dummy section.

name

is the name of an individual field within the external dummy section.
AID interprets the field according to the type and length attributes.

Reference to an external dummy section (XDSEC R)

```
⎧_Rn    ⎫    ⎧xdsec-r-name             ⎫
⎨       ⎬ -> ⎨                         ⎬
⎩c-name ⎭    ⎩[xdsec-r-name•.]name     ⎭
```

_Rn

specifies the base address register of the external dummy section.

c-name

is the name of an A, Y or V constant whose contents represent the base address of the external dummy section (see the ASSEMBH Reference Manual [10]).

xdsec-r-name

is the name of the external dummy section.

This addresses only those fields of the dummy section (in any order) that were referenced in the program.

[xdsec-r-name•]name

xdsec-d-name is the name of the external dummy section.

name

is the name of an individual field within the external dummy section. Only fields referenced in the program can be accessed. Output is effected according to the type and length attribute.

Dummy register (DXD)

```
⎧_Rn    ⎫      ⎧            ⎫
⎨       ⎬  ->  ⎨dxd-name    ⎬
⎩c-name ⎭      ⎩            ⎭
```

_Rn

specifies the register which is loaded with the start address of the dummy register vector.

c-name

is the name of an A, Y or V constant whose contents represent the start address of the dummy register vector (see the ASSEMBH Reference Manual [10]).

dxd-name

is the name of the dummy register.

com-name

is the name of a common control section (see COM statement in the ASSEMBH Reference Manual [10]).

This addresses all named fields of the common control section.

You need only specify a base qualification (not a PROG qualification) ahead of *com-name* if *com-name* is not located in the current AID work area.

[com-name•]name

com-name is the name of the common control section.

name

This addresses the name of an individual field within the common control section.

L’name’

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro (@ macro; see the ASSEMBH Reference Manual [10]).

name is the name of an Assembler instruction (in the source program) which can be up to 64 characters in length, or a call of a predefined macro (@ macro).
name is abbreviated to 32 characters by AID.

L’name’ may be specified in all operands either designating an address in the executable part of the program (%DISASSEMBLE, %FIND, %INSERT) or serving for the output and modification of memory locations (%DISPLAY, %MOVE, %SET).

S’stmt-no’

is a source reference via which a named Assembler instruction or the call of a predefined macro (@ macro) can be referenced.
stmt-no is the statement number; it is assigned by the Assembler and can be found in column STMNT of the assembly listing.

stmt-no is an integer between 1 and 2^{31}-1.

S’stmt-no’ 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).

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.

5 AID commands

%AID

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

- With the *CCS* operand, you specify a CCS for interpreting characters if no CCS is explicitly indicated in the %DISPLAY command. Unicode character sets are not allowed.
- 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 Assembler 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 *LEV* operand, you can activate the output of levels within the call hierarchy produced by the %SDUMP %NEST AID command.

```
Command       Operand

              ⎧ CCS = {<coded-character-set> | *USRDEF}            ⎫
              ⎪                                                     ⎪
              ⎪ CHECK [= {ALL|NO}]                                  ⎪
              ⎪                                                     ⎪
              ⎪ REP [= {YES|NO}]                                    ⎪
              ⎪                                                     ⎪
              ⎪ SYMCHARS [= {STD|NOSTD}]                            ⎪
              ⎪                                                     ⎪
              ⎪ OV [= {YES|NO}]                                     ⎪
%AID          ⎨                                                     ⎬
              ⎪ LOW [= {ON|OFF|ALL}]                                ⎪
              ⎪                                                     ⎪
              ⎪            ⎧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.

```
-------
| CCS |
-------
```

<coded-character-set>

Name of the CCS (`<name 1..8>`) for interpreting AID data. XHCS must know the indicated character set. Otherwise, AID rejects the statement with the message `AID0555`.

*USRDEF

CCSNAME of the character set, that is assigned to the user ID. *USRDEF is the default value of *CCS*.

If you specify the *CCS* operand in a %AID command, AID checks if the CCSNAME is permitted by XHCS. If XHCS doesn‘t know the CCSNAME, the command is rejected and the current *CCS* value is kept.

The following AID command enables you to display a complete list of CCSNAMEs, that are supported by XHCS:

```
%SHOW %CCSN
```

```
_________
| 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**.

<u>NO</u>

%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 Assembler 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 |
-------
```

<u>ON</u>

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.

%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]

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

When debugging Assembler 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 $0 \le n \le 7$.

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.
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. The program has been halted by the BKPT macro.

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.

%CONTROL can only be used for structured Assembler programs with calls of predefined macros and no more than one control section (CSECT). Assembler programs written without predefined macros and/or containing more than one control section cannot be debugged with %CONTROL. For such programs, the #<gt>%CONTROL command must be entered on machine code level (see AID, Debugging on Machine Code Level [2]).

– By means of the *criterion* operand you may select different types of Assembler instructions. If an instruction 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 or all active %CONTROL 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 Assembler instructions 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 executed prior to
%CALL	the predefined macro @PASS (Assembler procedure call)
%COND	the predefined macros for selection structure blocks @IF, @THEN, @ELSE, @CASE, @BEGI, @CAS2, @OF, @OFRE
%GOTO	the predefined macros @BREA and @EXIT
%PROC	the predefined macro @ENTR (Assembler procedure start)
%STMT	every predefined macro that is executed.

```
-----------------
| 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 Assembler instruction 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 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'stmt-nr' : S'stmt-nr' )  ⎭

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

•

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.

This program unit must have been loaded at the time the %CONTROL command or the subcommand containing %CONTROLn 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’stmt-no’ : S’stmt-no’)

is a source reference via which every call of a predefined macro (@ macro) can be referenced.
stmt-no is the statement number from the assembly listing; see STMNT column.

control-area is defined by specifying a start *stmt-no* and an end *stmt-no* and thus comprises a particular segment of the source program.

The start *stmt-no* must be less than the end *stmt-no*.

If *control-area* is to comprise only one line, the start *stmt-no* and end *stmt-no* must be identical.

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

subcmd is processed whenever an Assembler instruction that satisfies the *criterion* is awaiting execution in the *control-area*. *subcmd* is processed before execution of the *criterion* instruction.

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, %PROC  IN(S'123':S'250') <%DISPLAY  COUNTER;%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 lines 123 and 250 (inclusive).

   If one of the Assembler instructions 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 procedure call (@ PASS), 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 @BREA or @EXIT macro, 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 the first instruction of an Assembler procedure (@ENTR) is executed.

%DISASSEMBLE

%DISASSEMBLE enables memory contents to be "retranslated" into symbolic Assembler notation and displayed accordingly.

– The *output-quantity* operand defines the amount of memory contents that are to be disassembled and output.

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

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

 ⎧%DISASSEMBLE          ⎫
 ⎨                      ⎬ [output-quantity]    [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-EXECUTABLE-PROGRAM, START-EXECUTABLE-PROGRAM, %BASE). AID continues disassembly at the memory address following the address last processed by the previous %DISASSEMBLE command. If *output-quantity* is not specified either, AID generates the same amount 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 *output-quantity* 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.

```
-------------------
| output-quantity |
-------------------
```

Specifies the amount of the memory contents that are to be disassembled and output. If you don‘t specify *output-quantity*, AID inserts the default value 10 in the first %DISASSEMBLE after loading the program.

For each further %DISASSEMBLE command the last specified *output-quantity* is used.

```
output-quantity-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - -

⎧number ⎫
⎨length ⎬
⎩ALL    ⎭

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

number

Specifies, how many Assembler instructions are to be disassembled and output.

is an integer with the value:
1 ≤ *number* ≤ 2^{31}-1

length

Specifies the size of the memory content that is to be interpreted and output within a single, prompted %DISASSEMBLE command.

is a hexadecimal number #’f..f’ with the value:
1 ≤ *length* ≤ 2^{31}-1

ALL Specifies that the Assembler instructions are to be disassembled and output until the end of the CSECT, in which the *start* value is located. If *start* is not specified, the current %DA position determines the CSECT.

If the *start* value is not located within a CSECT, the command is rejected with an error message.

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

defines the address at which disassembly of memory contents into Assembler instructions 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 instruction last disassembled.

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

                     ⎧program-name               ⎫
                     ⎪L'name'                    ⎪
FROM  [•][qua•]     ⎨S'stmt-no'                 ⎬
                     ⎪                           ⎪
                     ⎩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 one or more qualifications 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’name’

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

With this specification you set *start* to the machine code generated for an Assembler instruction.

name can also be specified without L’...’ since it is not possible to confuse it with a data name in this command.

S’stmt-no’

is a source reference via which you can reference every executable Assembler instruction with a name and every call of a predefined macro.
stmt-no is the statement number from the assembly listing; see the STMNT column.

With this specification you set *start* to the machine code generated for an Assembler instruction.

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):

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

A statement name *L’name’* or a source reference *S’stmt-no’* can be used within *compl-memref*, but only in connection with the pointer operator, e.g. L’name’ ->.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. %3G.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

```
/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 20 FROM AID_DISPLAY`

   This command causes 20 instructions to be disassembled, starting at the address of the first executable instruction after the name entry: AID_DISPLAY.

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

   Two instructions are to be disassembled in the dump file with the link name D1 beginning at the start address of program unit BEISPIEL.

3. `%DA FROM S'123'`

   As no value was specified for *ausgabe-menge*, AID assumes either the default value 10 (if this is the first %DISASSEMBLE for this program) or accepts the value of the preceding %DISASSEMBLE.
   Disassembly begins at the first instruction generated for the instruction with the statement number 123.

%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 fields, 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).

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.

AID as of version 3.4B10 supports also the output of data in different EBCDIC character sets and ASCII character sets. As BS2000 terminals only support selected EBCDIC character sets directly, the following character sets must be distinguished:

- Character set of the data: Character set, in which the data is available or interpreted
- Character set of the output: Character set, with which the data is displayed

AID interprets the data using the character set that is specified with the %DISPLAY command. If no character set is specified there, the character set specified by the `CCS` operand of the %AID command is used.

First of all you must specify the character set of the output with the MODIFY-TERMINAL-OPTIONS command. It must be an EBCDIC character set that is supported by the terminal. UTFE is not allowed. Furthermore the character set of the output must be in the same group as the character set of the data. If, for example, the character set of the data is ISO88592, first of all specify the corresponding character set of the output with /MOD-TERM-OPT CODE=EDF042 (see the XHCS manual).

```
%DISPLAY <data-start> { %C|%X }[Lddd] ['<coded-character-set>']
```

If you prompt the %DISPLAY command with the `%C` or `%X` storage type, AID outputs the characters in accordance with the explicitly specified character set `<coded-character-set>`, or in accordance with the current character set `CCS` if '`<coded-character-set>`' is not specified. `%C` and `%X` define different output layouts.

```
%DISPLAY <char-variable> ['<coded-character-set>']
```

If `char` variables are to be output, AID outputs them in accordance with the explicitly specified character set `<coded-character-set>`, or in accordance with the current character set `CCS`. The output layout differs from the layouts that are determined by `%C` or `%X`.

To display the current character set `CCS` use the following AID command:

```
%SHOW %AID
```

To modify the current character set use the following AID command:

```
%AID CCS = {<coded-character-set>|*USRDEF}
```

```
---------
| data  |
---------
```

This operand defines the information AID is to output. You may output the contents, address and length of constants and data fields, plus the address of Assembler instructions or calls of predefined macros. The contents of registers and execution counters and the system information relevant to your program can be addressed via keywords. Registers can also be addressed via names predefined in the source program. 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 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). 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 field can be viewed, for instance by employing the type modification %X to edit the contents in hexadecimal form. Modification of the output type via the operand AS {BIN/CHAR/DEC/DUMP/HEX} is supported for the last time in this version (see AID Core Manual, appendix).

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'name'              ⎪     ⎪
⎪ [•][qua•][...]      ⎨S'stmt-no'           ⎬     ⎪
⎪                     ⎪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

One or more qualifications 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 “ASSEMBH-specific addressing” on page 11).

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 constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and and common control sections as defined in the source program.

dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM statement (see chapter “ASSEMBH-specific addressing” on page 11).

L’name’

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

If *L’name’* is entered without a pointer operator, the corresponding address is output in hexadecimal representation. With a pointer operator, i.e. with %DISPLAY L’name’->, AID outputs 4 bytes of the machine code contained at the relevant address.

S’stmt-no’

is a source reference via which every named executable Assembler instruction and every call of a predefined macro can be referenced.
stmt-no is the statement number from the assembly listing; see the STMNT column.

If *S’stmt-no’* is entered without a pointer operator, the corresponding address is output in hexadecimal representation. With a pointer operator, i.e. with %DISPLAY S’stmt-no’->, 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).
keyword can only be preceded by a base qualification.

```
%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
%MR                 All 16 general registers in tabular form
%FR                 All 4 floating-point registers with double precision
       edited 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 of the test object
%AUD1               P1 audit table, plus the SAVE table (if any)

%•subcmdname        Execution counter
%•                  Execution counter of the currently active subcommand

%HLLOC(memref)      Localization information on the symbolic level for a
       memory reference in the executable part of the
       program (high-level location)
%LOC(memref)        Localization information on machine code level for a
       memory reference in the executable part of the
                    program (low-level location)
```

compl-memref

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

- 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'name'* and the source references *S'stmt-no'*).

Example

```
%DISPLAY %@(AFIELD)
```

The address of AFIELD will be output.

%L(...)

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

Example

```
%DISPLAY %L(AFIELD)
```

The length of AFIELD will be output.

%L=(expression)

With the length function you can have a value calculated, see AID Core manual.

expression is formed from memory references and arithmetic operators.

Example

```
%DISPLAY %L=(AFIELD)
```

If AFIELD 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 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 ≤ *n* ≤ 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 - - - - - - - - - - - - - - - - - - - - - - - - - -
```

$$\left\{\begin{array}{l}\underline{\texttt{T}}\\ \texttt{H}\\ \texttt{Fn}\\ \texttt{P}\end{array}\right\} = \left\{\begin{array}{l}\texttt{MIN}\\ \underline{\texttt{MAX}}\\ \texttt{XMAX}\\ \texttt{XFLAT}\end{array}\right\}$$

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

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

T Terminal output

H Hardcopy output

Fn File output

P Output to SYSLST

<u>MAX</u> Output with additional information

MIN Output without additional information

XMAX 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 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 M2,INPUT,PACK,_R5`

```
SRC_REF:   212 SOURCE: SUM     PROC: SUM
************************************
M2              = |SUM:|
INPUT           = 00060000 F9F9
I375  SYMBOL    PACK NOT FOUND
_R5             = 0000000B
```

The default value for *medium-a-quantity* is T=MAX.

Every output has an AID header identifying the source line at which the program has stopped at the time of output. A typing error causes error message I375 to be output, after which AID processes the last name _R5.

The definitions in the Assembler source are as follows:

```
 M2      DC    C'SUM:'
 INPUT   DC    XL6'00'
 PACK    DC    PL2'0'
```

2. `/%DISPLAY E=D1.INPUT,'LAST VALUE'`

```
** D1: DUMP.NAME.2069.00001
*************************************************
INPUT           = 00060000 F2F9
LAST VALUE
```

3. `/%DISPLAY _R10 -> DS`

```
 DS
 DS1             = |ABCDE|
 DS3             =        1234
 DS4             = -.1300000 E+003
```

For this example, the following Assembler source was used:

```
CS      START
        USING *,15
        LA    10,DAT1
        TERM
DAT1    DC    CL5'ABCDE'
        DC    CL6'XXXXXX'
        DC    FL4'1234'
        DC    EL4'-1.3E2'
```

```
DS      DSECT
DS1     DS    CL5
        DS    CL6
DS3     DS    FL4
DS4     DS    EL4
              END
```

%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  [=file]]
 ⎩%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 ≤ *n* ≤ 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. `%DUMPFILE D3=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. the class 6 memory for the currently set base qualification is searched (see %BASE).

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

                 ⎧dataname                ⎫
                 ⎪L'name'->               ⎪
IN [•][qua•]    ⎨S'stmt-no'->            ⎬
                 ⎪                        ⎪
                 ⎩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

One or more qualifications 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 “ASSEMBH-specific addressing” on page 11).

dataname

specifies the name of constants, data fields, predefined macros, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and common control sections as defined in the source program.

dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM statement (see chapter “ASSEMBH-specific addressing” on page 11).

L’name’

designates the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

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

S’stmt-no’

designates the memory location via which every named executable Assembler instruction and every call of a predefined macro can be referenced.
stmt-no is the statement number from the assembly listing; see the STMNT column.

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

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):

- 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.

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

⎧%AID | %AINT | %BASE | %CONT[INUE] | %C[ONTROL]                   ⎫
⎪%DISASSEMBLE | %DA | %D[ISPLAY] | %DUMPFILE | %DF                 ⎪
⎪%F[IND] | %H[ELP] | %IN[SERT] | %JUMP | %M[OVE]                   ⎪
⎨%ON | %OUT | %OUTFILE | %Q[UALIFY]                                ⎬
⎪%REM[OVE] | %R[ESUME] | %SD[UMP]                                  ⎪
⎪%S[ET] | %STOP | %SYMLIB | %TITLE | %T[RACE]                      ⎪
⎪                                                                  ⎪
⎩In                                                                ⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

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.

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

defines the media via which information on the *info-target* is to be output. The specification { MIN | MAX | XMAX | XFLAT} has no effect with %HELP, but the syntax requires one of these two specifications.

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-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - -
⎧T    ⎫   ⎧MIN  ⎫
⎪H    ⎪   ⎪MAX  ⎪
⎨Fn   ⎬ = ⎨XMAX ⎬
⎩P    ⎭   ⎩XFLAT⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

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

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 principle (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 an Assembler instruction. *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'name'               ⎪
 [•][qua•]     ⎨S'stmt-no'            ⎬
                ⎪                      ⎪
                ⎩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

One or more qualifications are 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 “ASSEMBH-specific addressing” on page 11).

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'name'

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

With this specification you set *test-point* to the machine code generated for an Assembler instruction.

name may not be located within a dummy section or dummy register (see DSECT, XDSEC and DXD statements in the ASSEMBH Reference Manual [10]).

name can also be specified without L'...' since it is not possible to confuse it with a data name in this command.

S'stmt-no'

is a source reference via which every named executable Assembler instruction and every call of a predefined macro can be referenced.
stmt-no is the statement number from the assembly listing; see the STMNT column.

With this specification you set *test-point* to the machine code generated for an Assembler instruction.

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):

– byte offset (•)

– indirect addressing (->)

– type modification (%A)

– length modification (%Ln)

– address selection (%@(...))

A statement name *L'name'* or a source reference *S'stmt-no'* can be used within *compl-memref*, but only in connection with the pointer operator (e.g. L'name' ->.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. %3G.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 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 [⎨C    ⎬]
               ⎩S    ⎭

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

n is a number with the value 1 ≤ *n* ≤ 65535, 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'118'`

   test-point is specified with an *stmt-no*. This designates the Assembler instruction 118 in the assembly listing.

2. `%IN PROG=PRO2.PRO2 <%DISPLAY NO> ONLY 10 S`

   test-point is set to the start of module PRO2, i.e. deleted. Whenever the program sequence arrives at the first instruction in module PRO2, the #<gt>%DISPLAY command from *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.

3.
```
%IN ST2 <%DISPLAY TEXTDAT, 'ST2'>
   %IN ST3 <%DISPLAY 'INSERT1', TEXTDAT; %IN OUUTPUT <%D 'INSERT2', -
    I,J,K, NUMBER; %IN S'172' <%D 'INSERT3' ,I,J; %REMOVE OUTPUT>>>
```

With the first %INSERT command, the *test-point* set is the Assembler instruction with the name ST2. If, after the end of command input, the program execution reaches ST2, the subcommand is executed. It consists of a %DISPLAY command (for field TEXTDAT) and the AID literal ST2. Afterwards the program is continued.

By means of the second %INSERT command, *test-point* ST3 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 Assembler instruction ST3, the corresponding *subcmd* is executed, i.e. the %DISPLAY command for the AID literal ’INSERT1’ and the field TEXTDAT is executed and the *test-point* OUTPUT is set.
The *subcmd* for *test-point* OUTPUT 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: ST2, ST3 and OUTPUT.

As the *subcmd* for *test-point* ST3 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 OUTPUT, 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 *stmt-no* S’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 fields I and J.

By way of the second command in this *subcmd*, the %REMOVE OUTPUT command, *test-point* OUTPUT 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 OUTPUT:

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

4.
```
%IN ST4  <%D TEXTDAT>
        .
        .
        .
   %IN ST4
```

These two commands show how chaining in conjunction with the default value inserted by AID affects *subcmd*.

For the missing *subcmd* in the second %INSERT, AID inserts the following *subcmd*:
`<%STOP>`
Since the second %INSERT denotes the same *test-point*, chaining is performed and the *subcmd*
`<%STOP; %DISPLAY TEXTDAT>`
is produced. A *subcmd* is aborted by any %STOP, %RESUME or %TRACE.
The two consecutive %INSERT commands therefore yield the same results as if you had deleted the first %INSERT by means of
`%REMOVE ST4`
and then written `%INSERT ST4` or just entered `%INSERT ST4`.

%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 data field, a statement name, a source reference, a length, an address, an execution counter, a 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 data field, an execution counter or or a 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.

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 data field or a complex memory reference, or an execution counter or a register. Statement names, source references, addresses and lengths of data fields 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.

```
sender-OPERAND  - - - - - - - - - - - - - - receiver-OPERAND - - - - - - - - -

⎧           ⎧dataname              ⎫ ⎫
⎪           ⎪                      ⎪ ⎪
⎪           ⎪L'name'               ⎪ ⎪
⎪[•][qua•]  ⎨                      ⎬ ⎪
⎪           ⎪S'stmt-no'            ⎪ ⎪
⎪           ⎪keyword               ⎪ ⎪               ⎧dataname         ⎫
⎪           ⎩compl-memref          ⎭ ⎪               ⎪                 ⎪
⎨                                    ⎬  INTO  [•][qua•] ⎨                 ⎬
⎪⎧%@⎫        ⎧dataname             ⎫ ⎪               ⎪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

One or more qualifications are necessary only for memory objects 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 “ASSEMBH-specific addressing” on page 11).

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 constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and and common control sections as defined in the source program.

dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM statement (seechapter “ASSEMBH-specific addressing” on page 11).

L’name’

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

S’stmt-no’

is a source reference via which every named executable Assembler instruction and every call of a predefined macro can be referenced.
stmt-no is the statement number from the assembly listing; see the STMNT column.

Statement names and source references are address constants and can therefore only be specified for *sender*. The address designated using *L’name’* or *S’stmt-no’* is then transferred.

Example

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

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

With *L’name’->* or *S’stmt-no’->* you designate 4 bytes of the machine code at the corresponding address (see AID Core Manual).
%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 *stmt-no* 24 is overwritten by that of *stmt-no* 12. The specification %L=(S’13’-S’12’) yields the length of the machine code generated for *stmt-no* 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):

- 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 issues 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 data fields CFIELD and CFIELD1 each occupy 5 bytes. The last 2 bytes of CFIELD as well as the 3 following bytes are to be transferred to CFIELD1.
AID would reject the following command as a violation of the CFIELD area:

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

The correct command reads:

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

%@(...)

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

%L(...)

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

Example

```
%MOVE %L(FIELD1) INTO %0G
```

The length of FIELD1 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). 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=(FIELD1) INTO %0G
```

The contents of FIELD1 are transferred.

AID literal

The following AID literals (see AID Core Manual) 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.

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 constants and fields are defined in a program:

```
IFIELD  DC   F'123,456'
        DS   0F
JFIELD  DS   10F
CVAR    DC   X'F0F0F0F0'
```

1. `%MOVE IFIELD INTO JFIELD`

   AID transfers the contents of IFIELD to the symbolic address JFIELD in hexadecimal format and left-justified.

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

   AID writes a word containing an integer with the value 20 to the field JFIELD.

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

   The contents of the CVAR constant 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 *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
----------------------------------------------------------------------------

%ON                event  [<subcmd>]

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

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.

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

The %ON command does not alter the program state.

```
---------
| 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).
For selection of the SVC numbers see the "Executive Macros" manual [7].

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 (up to 8 alphanumeric chars.)
%LPOV	after loading of any arbitrary segment
%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
%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.

4. `%ON %ARTHCHK < %SD AMOUNT,MATRIX; %STOP >`

   If a data error, division error, exponent overflow or "mantissa equals zero" occurs, the data fields AMOUNT and MATRIX are output via %SDUMP. The %STOP command interrupts the program and you can proceed to a detailed check of the error situation by means of further commands.

5. `%ON %LPOV (MON12) <%D INDEX, GRAND-TOTAL> ONLY 37 S`

   After every loading of segment MON12, AID outputs the data fields INDEX and GRAND-TOTAL as a result of %D. After the 37th occurrence of the *event* %LPOV (MON12) and subsequent output, the *event* is deleted and the program interrupted.

%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.

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 the program terminates.

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

1. Enter an ADD-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 ≤ *n* ≤ 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. `%Q E=D1.PROG=INITIAL`
   ```
    %D .TAB1
   ```

   Because of the *prequalification*, the %DISPLAY command has the same effect as the following %DISPLAY command in full format: `%D E=D1.PROG=INITIAL.TAB1`

2. `%Q PROG=LEADER`
   ```
    %SET .E_RECD INTO .A_RECD
   ```

   Because of the *prequalification*, the %SET command has the same effect as the following %SET command:
   `%SET PROG=LEADER.E_RECD INTO PROG=LEADER.A_RECD`

%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 | event                 ⎬
 ⎩%•[subcmdname]              ⎭

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

%C[ONTROL]

The declarations for all %CONTROLn commands entered are deleted.

%C[ONTROL]n

The %CONTROLn command with the specified number (1 ≤ *n* ≤ 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.

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). 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 S'58' <SUB1: %D CHAR, NUMB>
```

```
%IN S'58' <SUB2: %D RESULT; %REM %.>
%R
...
%REM S'58'
```

   When the test point S’58’ is reached, RESULT is output. Then subcommand SUB2 is deleted, i.e. this subcommand is executed only once. Subsequently CHAR and NUMB are output, and the program continues. Whenever test point S’58’ is reached in the program sequence, subcommand SUB1 is executed. `%REM S'58'` deletes the test point later on. `%REM SUB1` would have the same effect, as this subcommand is the only remaining entry for test point S’58’.

%RESUME

With %RESUME you start the loaded program or continue it at the interrupt point. 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 or data areas, all data areas 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, over the subprograms invoked through to the main program.

– With *dump-area* you designate the data or data areas 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][,...]]

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

The following applies for a structured Assembler program:

– %SD %NEST produces a minimal output, i.e. AID lists only the current call hierarchy.

– %SD without operands results in the maximum output, i.e. the named data defined in the current call hierarchy will be output together with all further Assembler instructions which have a name in the name entry. Multiply defined data will also be output multiply. A header line identifies the program unit in which data definition took place.

– If one or more names are explicitly specified in the command, all data with this name defined in the current call hierarchy will be output. Data with the same name defined in different modules will be multiply output.

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).

In unstructured Assembler programs, only the data of the current programming unit can be referenced:

– %SD without operands causes the entire data area and all further Assembler instructions with a name to be output.

– If one or more names are explicitly specified in the command, only the corresponding data of the current program unit will be output.

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. AID edits the data 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 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                   ⎧ dataname   ⎫
[•][qua[•]][⎨            ⎬]
                   ⎩ %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.

qua

Specify one or more qualifications if the interrupt point is not within the scope of the addressed object or if the memory object is not visible at the interrupt point. Only enter the qualification required for unique addressing.

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.

NESTLEV= level-number

level-number A level number in the current call hierarchy

level-number can only be followed by *dataname*.
The %SDUMP command is to output a symbolic dump of all data defined at the specified level or to output *dataname* defined at the specified level of the call hierarchy.

dataname

is the name of constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and common control sections as defined in the source program.
dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM statement (see chapter “ASSEMBH-specific addressing” on page 11).

A *dataname* not identified in the LSD records causes AID to issue an error message. Subsequent *dump-areas* of the same command are output normally.

%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

<u>MAX</u> 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.

```
<BYTES(size),X>
bytestring-name = bytestring
```

size Storage length in bytes.

bytestring-name Specifies an element of the type string.

bytestring String of hexadecimal bytes (X); value of *bytestring-name*. Four hexadecimal bytes are combined to form a hexadecimal word and are separated by a blank.

If the output is longer than 80 characters, with XFLAT only the first 8 hexadecimal words (i.e. 32 hexadecimal bytes) 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.

```
<BITS(size),B>
bits-name = bitstring
```

size Storage length in bytes.

bits-name Specifies an element of the type bit string.

bitstring Sequence of binary numbers (B); value of *bits-name*.

If *bitstring* contains more than 80 digits, with XFLAT only the first 72 hexadecimal bytes (i.e. 8 hexadecimal words) 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.

```
<PACKED(size),D>
packed-name = packed-value
```

size Storage length in bytes.

packed-name Specifies an element of the type packed decimal.

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

```
<ZONED(size),D>
zoned-name = zoned-value
```

size Storage length in bytes.

zoned-name Specifies an element of the type zoned decimal (unpacked decimal number)

zoned-value Decimal value (D); value of *zoned-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.

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. ....
  ```

i In order to query the entire content of a structure and of its substructures, use XMAX instead of XFLAT.

Examples

1. The program SUM from PLAM library PLAMBIB is to be executed.

   ```
   /START-PROG (PLAMBIB,SUM),TEST-OPTION=AID

   %  BLS0001 *** DBL VERSION 070 RUNNING ***
   %  BLS0517 MODULE 'SUM' LOADED

   PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END: 00
   *29
   *37
   *00
   *
   /
   ```

   The program branches to the entry although the end criterion ’00’ was specified. A program error is present. The program is interrupted by pressing the K2 key and a memory dump requested with %SDUMP. This command requests a symbolic dump of the entire module. The value for *medium-a-quantity* is T=MAX. The source program for this %SDUMP output can be found in chapter 6. The various %SDUMP lines are followed by explanatory texts.

```
/%SDUMP

** ITN: ^#000B018F'***TSN:4J60*******************************************'
SRC_REF:    60  SOURCE: SUM       PROC: SUM
*******************************
```

SRC_REF line: this contains the statement number at which the program was interrupted and the module name.

```
R0              =           0

R1              =           1

R2              =           2

R3              =           3

R4              =           4

R5              =           5
```

Names of the EQU statements that were assigned a constant value.

```
SUM             = 00000000

START           = 4D102022

S0002D          =     2363650

S0002S          = 0A27

LOOP           = 5A50217E   !&.=00002A 5A 50 217E     00000180     53
                 LOOP A     R5,=F'1'

READ            = 4D10205E

S0006D          =     2363906

S0006S          = 0A27

VERGL           = D5052129 2142

ADD             = F211212B 2129

FROM            = F3632139 2144

S0010D          =     2363650

S0010S          = 0A27

END             =       16656
```

```
S0014D          =      403457
S0014S          = 0A09
ERROR           = 4D1020E6
S0016D          =     2363650
S0016S          = 0A27
```

Assembler instructions with a name; AID outputs the memory contents at the relevant address in accordance with the length attribute.

```
MESS1           = 0039
M1              = |PLEASE ENTER UP TO 10 2-DIGIT NUMBERS! END: 00|
INPUT           = 00060000 F0F0   ....00
PACK            =     +0
MESS2           = 0012
M2              = |SUM:|
RESUL           = |       |
TEN             =          10
NULL            = |00|
TOTAL           =        +66
ZONE            = F0
MESS3           = 0034
M3              = |NO MORE THAN 10 NUMBERS CAN BE PROCESSED|
```

Data area of the module; output comprises the names of the DC and DS statements with the respective memory contents.

```
_R0             = 00000000
_R1             = 9F00003C
_R2             = 00000002
```

```
_R3             = 00000000

_R4             = 00000000

_R5             = 00000005

_R6             = 00000000

_R7             = 00000000

_R8             = 00000000

_R9             = 00000000

_R10            = 00000000

_R11            = 00000000

_R12            = 00000000

_R13            = 00000000

_R14            = 00000000

_R15            = 00000000
```

General register; AID outputs the predefined name of the register together with its contents.

2. Examples for XMAX und XFLAT

The following Assembler program is to be debugged:

```
HELLO   START
BALR    BALR    10,0
        USING   *,10
*       WROUT   H,*
        TERM
*
DS      0D
C120    DS      0CL120
X120    DS      0XL120
B200    DS      0BL25
H       DC      H'25'
        DC      X'404001'
M1      DC      CL20'HELLO, WORLD!'
*
```

```
D       DS      0D E DS 0E
Z16     DC      ZL16'1234567890123456'
P16     DC      PL16'-1234567890123456789012345678901'
F       DS      0F
P1      DC      PL1'1'
Z1      DC      ZL1'-1'
*
Y       DC      Y(REGS)
A       DC      A(REGS)
S       DC      S(REGS)
V       DC      V(NONE)
Q       DC      Q(REGS)
*
REGS DS 16F END
```

After the Assembler program has been loaded, the following AID commands are entered:

```
%AID LOW=OFF
%INSERT BALR
%RESUME
```

The two variants below show the effect of XFLAT and XMAX:

XFLAT without data operand

The long strings C120, X120 and B200 are truncated.

```
/%SD T=XFLAT
SRC_REF:     2 SOURCE: HELLO PROC: HELLO *************
<ADDR(4),X>
HELLO           = 00000000

<AREA(2),X>
BALR            = 05A0     ..

<CHARS(120),C>
C120            =
|.. .HELLO, WORLD! .......123456789012345F.................J......| ...

<BYTES(120),X>
X120            =
00194040 01C8C5D3 D3D66B40 E6D6D9D3 C45A4040 40404040 40000000 00000000 ...

<BITS(25),B>
B200            =
0000000000011001010000000100000000000000111001000110001011101001111010 01 ...

<INT(2),D>
```

```
H               = 25

<CHARS(20),C>
M1              = |HELLO, WORLD! |

<FLOAT(8),E>
D               = -.9531502657561182 E+059

<FLOAT(4),E>
E               = -.9531502 E+059

<ZONED(16),D>
Z16             = +1234567890123456

<PACKED(16),D>
P16             = -1234567890123456789012345678901

<INT(4),D>
F               = 483459188

<PACKED(1),D>
P1              = +1

<ZONED(1),D>
Z1              = -1

<POINTER(2),X>
Y               = 0074

<POINTER(4),X>
A               = 00000074

<POINTER(2),X>
S               = A072

<POINTER(4),X>
V               = FFFFFFFF

<POINTER(4),X>
Q               = 00000074

<INT(4),D>
REGS            =          0

<POINTER(4),X>
_R0             = 00000000

<POINTER(4),X>
_R1             = 00000000
```

```
<POINTER(4),X>

_R2             = 00000000

<POINTER(4),X>
_R3             = 00000000

<POINTER(4),X>
_R4             = 00000000

<POINTER(4),X>
_R5             = 00000000

<POINTER(4),X>
_R6             = 00000000

<POINTER(4),X>
_R7             = 00000000

<POINTER(4),X>
_R8             = 00000000

<POINTER(4),X>
_R9             = 00000000

<POINTER(4),X>
_R10            = 00000000

<POINTER(4),X>
_R11            = 00000000

<POINTER(4),X>
_R12            = 00000000

<POINTER(4),X>
_R13            = 00000000

<POINTER(4),X>
_R14            = 00000000

<POINTER(4),X>
_R15            = 00000000
```

XMAX for long byte string

The string is displayed in full.

```
/%SDUMP X120 T=XMAX
SRC_REF:     2 SOURCE: HELLO PROC: HELLO ***************************
```

```
<BYTES(120),X>
X120 =
00194040 01C8C5D3 D3D66B40 E6D6D9D3 C45A4040 40404040 40000000 00000000
F1F2F3F4 F5F6F7F8 F9F0F1F2 F3F4F5C6 12345678 90123456 78901234 5678901D
1CD10074 00000074 A0720000 FFFFFFFF 00000074 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000
```

%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 data field, a statement name, a source reference, 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 data field, an execution counter or a 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.

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.

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 data fields, a complex memory reference, an execution counter or a register. Statement names, source references, addresses, lengths of data areas and AID literals can only be used as *sender*.

sender may be located either in the virtual memory area of the loaded program 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'name'         ⎪    ⎪
 ⎪ [•][qua•]  ⎨S'stmt-no'       ⎬    ⎪
 ⎪             ⎪keyword         ⎪    ⎪
 ⎪             ⎩compl-memref    ⎭    ⎪                       ⎧dataname         ⎫
 ⎪                                   ⎪                       ⎪                 ⎪
⎨ ⎧%@⎫            ⎧dataname      ⎫   ⎬ INTO  [•][qua•]  ⎨                 ⎬
 ⎪ ⎨  ⎬ ([•][qua•] ⎨             ⎬)  ⎪                       ⎪kexword          ⎪
 ⎪ ⎩%L⎭            ⎩compl-memref  ⎭   ⎪                       ⎩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

One or more qualifications 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 “ASSEMBH-specific addressing” on page 11).

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 constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and common control sections as defined in the source program.
dataname is the name entry of a DC, DS, EQU, CSECT, DSECT, XDSEC, DXD or COM statement (see chapter “ASSEMBH-specific addressing” on page 11).

L’name’

is a statement name, designating the address of an executable Assembler instruction or a call of a predefined macro.
name is the name entry of an Assembler instruction or a call of a predefined macro (@ macro).

S’stmt-no’

is a source reference via which every named executable Assembler instruction and every call of a predefined macro can be referenced.
stmt-no is the statement number from the assembly listing; see the STMNT column.

Statement names and source references are address constants and can thus only be specified as *sender*. The address designated with *L’name’* or *S’stmt-no’* 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’name’*-> or *S’stmt-no’*-> you designate 4 bytes of machine code at the corresponding address (see AID Core Manual).
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 an execution counter, the program counter or a register. The AID Core Manual lists the implicit storage types of the keywords.

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

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

– 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).
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 field without regarding its area boundaries.

Example

The data fields CFIELD and CFIELD1 are of type ’character’ and occupy 5 bytes each. The last 2 bytes of CFIELD as well as the next 3 bytes are to be transferred to CFIELD1.
AID would reject the command shown below, since it represents a violation of the CFIELD area:

```
%SET CFIELD.3%CL5 INTO CFIELD1
```

The correct command reads:

```
%SET %@(CFIELD)->.3%CL5 INTO CFIELD1
```

%@(...)

The address selector can be used to specify the address of a data field or complex memory reference as *sender* (see also AID Core Manual). The address selector produces an address constant as a result.

%L(...)

The length selector can be used to specify the length of a data field or complex memory reference as *sender* (see also AID Core Manual). The length selector produces an integer as a result.

Example

```
%SET %L(FIELD1) INTO %0G
```

The length of FIELD1 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). 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=(FIELD1) INTO %0G
```

The contents of FIELD1 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 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

	Receive field Data type:															
Send field	A	B	C	D	E	F	H	L	P	Q	S	V	X	Y	Z	Reg. Rn
Data type A	2	–	–	–	–	–	–	–	–	–	–	2	–	2	–	2
B	–	1	–	–	–	–	–	–	–	1	1	–	1	–	–	–
C	–	–	2a	–	–	–	–	–	–	–	–	–	–	–	–	–
D	–	–	–	2	2	4a*	4a*	2	–	–	–	–	–	–	4c*	–
E	–	–	–	2	2	4a*	4a*	2	–	–	–	–	–	–	4c*	–
F	–	–	–	4b	4b	1	1	4b	–	–	–	–	–	–	4c	–
H	–	–	–	4b	4b	1	1	4b	–	–	–	–	–	–	4c	–
L	–	–	–	2	2	4a*	4a*	2	–	–	–	–	–	–	4c*	–
P	–	–	–	4b	4b	4a	4a	4b	–	–	–	–	–	–	–	–
Q	–	1	–	–	–	–	–	–	–	1	1	–	1	–	–	–
S	–	1	–	–	–	–	–	–	–	1	1	–	1	–	–	–
V	2	–	–	–	–	–	–	–	–	–	–	2	–	2	–	2
X	–	1	–	–	–	–	–	–	–	1	1	–	1	–	–	–
Y	2	–	–	–	–	–	–	–	–	–	–	2	–	2	–	2
Z	–	–	–	4b	4b	4a	4a	4b	–	–	–	–	–	–	3	–
Register: Rn	2	–	–	–	–	–	–	–	–	–	–	2	–	2	–	2
AID literals alphanum.	–	–	2a	–	–	–	–	–	–	–	–	–	–	–	–	–
Alphanum. hexadecimal	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Bit	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Numeric	–	–	–	4a	1	1	4a	4a	–	–	–	–	–	–	4c	–
Numeric hexadecimal	1	–	–	2b	2b	1	1	2b	–	–	–	1	–	1	–	1
Float. pt.	–	–	–	2	2	4a*	4a*	2	–	–	–	–	–	–	4c	–

The table above provides an overview of permissible combinations of the sender and receiver types in conjunction with the %SET command (see also the ASSEMBH Reference Manual [10], "DC and DS statements, Types of constants").

Meaning of 1, 2, 2a, 2b, 3, 4a-c, *, -

1 The receive field is overwritten right-justified with the contents of the send field. If the lengths differ, padding with X’00’ or truncation occurs on the left.

2 The receive field is overwritten left-justified with the contents of the send field. If the lengths differ, padding with X’00’ or truncation occurs on the right.

2a The receive field is overwritten left-justified with the contents of the send field. If the lengths differ, padding with X’40’ or truncation occurs on the right.

2b Refers to the mantissa only: The receive field is overwritten left-justified with the contents of the send field. If the lengths differ, padding with X’00’ or truncation occurs on the right.

3 The receive field is overwritten right-justified with the contents of the send field. If the lengths differ, padding with X’F0’ or truncation occurs on the left.

4 The type of the sender is converted to the internal representation of the receiver type. If the lengths of the sender and the converted receiver differ,

4a truncation or padding with X’00’ occurs on the left

4b truncation or padding with X’00’ occurs on the right

4c truncation or padding with X’F0’ occurs on the left.

* The types F, H, Z must not be used as receiver if the sender of type E, D or L is not an integer.

− Transfer not possible

Examples

1.
```
%SET PROG=PRO1.MESS1 INTO PROG=PRO1.MELD2
```

MESS1 and MELD2 are defined in program unit PRO1. The contents of MESS1 are transferred to MELD2.

2.
```
%QUALIFY PROG=PRO1
```

```
%SET .MESS1 INTO .MELD2
```

This %SET command initiates the same transfer as in example 1: The prequalification defined in %QUALIFY is accepted before the leading period.

3. `%SET 'ABCDEF' INTO MESSUNG`

   Data field MESSUNG is 8 characters long. Following transfer, it contains: `ABCDEF␣␣`

4. `%SET #'0' INTO _R5`

   `%SET _R5 INTO _R10`

   The first %SET transfers the hexadecimal number #’0’ to register 5, i.e. register 5 is cleared. The second %SET clears register 10.

%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

```
/%INSERT S'214' <%DISPLAY M2,TOTAL; %STOP>
/%RESUME

M2              = |SUM:|
TOTAL           = +.110000000000000000000000000000000 E+002
USTOPPED AT LABEL: ADD , SRC_REF: 214, SOURCE: SUM ,PROC: SUM
```

%INSERT sets a test point for statement 214. The subcommand comprises the %DISPLAY and %STOP commands. After M2 and TOTAL have 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,ASSOUTPUT
```

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.

%TRACE can only be used for structured Assembler programs with calls of predefined macros. Also, these programs may only contain one control section (CSECT). Assembler programs not created with predefined macros and/or comprising more than one CSECT cannot be processed with %TRACE. Such programs can only be traced via a %TRACE command on machine code level (see AID, Debugging on Machine Code Level [2]).

– By means of the *number* operand you can specify the maximum number of Assembler instructions 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 Assembler instructions 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 instructions 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 calling the BKPT macro.

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 Assembler instructions of type *criterion* which are to be executed and logged.

number

is an integer 1 ≤ *number* ≤ 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 instructions 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 instruction 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 instructions 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	Tracing and logging takes place prior to execution of
%CALL	the predefined macro @PASS (Assembler procedure)
%COND	the predefined macros for selection structure blocks @IF, @THEN, @ELSE, @CASE, @BEGI, @CAS2, @OF, @OFRE
%GOTO	the predefined macros @BREA and @EXIT
%PROC	the predefined macro @ENTR (Assembler procedure start)
%STMT	each predefined macros that is executed.

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.

```
trace-area-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                  ⎧PROG=program-name                     ⎫
IN  [•][E=VM•]   ⎨                                       ⎬
                  ⎩[PROG=program-name•]( S'stmt-no')      ⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

•

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.

This program unit must already be loaded at the time the %TRACE command is input or at the time the subcommand containing the %TRACE command is executed.

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’stmt-no’:S’stmt-no’)

stmt-no is the statement number from the assembly listing; see the STMNT column.

trace-area is defined by entering a start *stmt-no* and an end *stmt-no*, which together identify a certain part of the source program.

The start *stmt-no* must be less than the end *stmt-no*.

If *trace-area* is to comprise only one line, the start *stmt-no* and end *stmt-no* 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).

A %TRACE listing with additional information (T=MAX) contains the statement number and type of Assembler instruction that was executed. If a name entry exists, it will be output as well.

A %TRACE listing without additional information (T=MIN) does not show the instruction 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

     7  FRAME                       EXT.PRO
  1037  TEST1                       CALL
  1281  TEST2                       JUMP
STOPPED AT LABEL: TEST2 , SRC_REF:  1281,  SOURCE: FRAME , PROC: FRAME
```

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 Assembler instructions. After the third instruction, the termination message for this %TRACE command follows, to the effect that program execution was interrupted at instruction TEST2 with statement number 1281, that instruction TEST2 is in the program unit FRAME and that the load module has the same name.

2.

```
/%OUT %T T=MIN
/%T 3

     7  FRAME
  1037  TEST1
  1281  TEST2
STOPPED AT SRC_REF:  1281,  SOURCE: FRAME, PROC: FRAME
```

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 Assembler 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 Assembler program is shown in section 6.1, the test run in section 6.2. In the examples below, input is printed in bold for better legibility.

6.1 Assembler program

Objective

The program SUM is to read in up to 10 two-digit numbers and output the resulting total. Input of the number 00 serves as the end criterion .
If more than 10 numbers are entered, a message is issued together with the calculated total.

Source program listing

```
 COMPUTE THE SUM OF N NUMBERS (N <= 10)                                                             11:09:30  91-11-
05
  LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
  000000                                       1     SUM      START
                       2              TITLE 'COMPUTE THE SUM OF N NUMBERS (N <= 10)'
                       3              PRINT NOGEN
  000000                00000000               4     R0       EQU   0
  000000                00000001               5     R1       EQU   1
  000000                00000002               6     R2       EQU   2
  000000                00000003               7     R3       EQU   3
  000000                00000004               8     R4       EQU   4
  000000                00000005               9     R5       EQU   5
                      10     SUM      AMODE ANY
                      11     SUM      RMODE ANY
                      12              GPARMOD 31
                      14   2                *,VERSION 010
  000000 0D 20                                15              BASR  R2,R0
  000002                00000002              16              USING *,R2
                      17     START    WROUT MESS1,END
                      24   2                *,FHDR VERSION 105 / 1988-06-13
                      48   2                *,@DCEO      952    900503   53531004
                      51   1                *,WROUT      005    910215   53121058
  000026 58 50 2176     00000178              52              L     R5,=F'1'
  00002A 5A 50 2176     00000178              53     LOOP     A     R5,=F'1'
  00002E 49 50 2138     0000013A              54              CH    R5,ZEHN
  000032 47 20 20BE     000000C0              55              BH    ERROR
                      56     READ     RDATA INPUT, END
                      63   2                *,FHDR VERSION 105 / 1988-06-13
                      92   2                *,@DCEI      920    881104   53531002
                      95   1                *,RDATA      006    910215   53121057
  000062 D5 05 2121213A 00000123 0000013C     96     COMP     CLC   INPUT+4,NULL
  000068 47 80 207A     0000007C              97              BE    FROM
  00006C F2 11 21232121 00000125 00000123     98     ADD      PACK  PACK,INPUT+4(2)
  000072 FA 31 213C2123 0000013E 00000125     99              AP    TOTAL,PACK
  000078 47 F0 2028     0000002A             100              B     LOOP
  00007C F3 63 2131213C 00000133 0000013E    101     FROM     UNPK  RESUL,TOTAL
  000082 D3 00 21372140 00000139 00000142    102              MVZ   RESUL+6(1),ZONE
                         103              WROUT MESS2,END
                         109   2                *,FHDR VERSION 105 / 1988-06-13
                         133   2                *,@DCEO      952    900503   53531004
                         136   1                *,WROUT      005    910215   53121058
                         137     END      TERM  DUMP=Y
                         140   2                *,VERSION 010
                         152     ERROR    WROUT MESS3,END
                         159   2                *,FHDR VERSION 105 / 1988-06-13
                         183   2                *,@DCEO      952    900503   53531004
                         186   1                *,WROUT      005    910215   53121058
  0000E2 47 F0 207A     0000007C             187              B     FROM
                         188              EJECT
                         189     *
                         190     * DEFINITIONEN
                         191     *
  0000E6 0039                                192     MESS1    DC    Y(L'M1+5)
  0000E8 404001                              193              DC    X'404001'
  0000EB C2C9E3E3C540C2C9                      194     M1       DC     C'PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END:
00'
  00011F 000000000000                        195     INPUT    DC    XL6'00'
  000125 000C                                196     PACK     DC    PL2'0'
                         197     *
  000128 0012                                198     MESS2    DC    Y(L'M2+L'RESUL+5)
  00012A 404001                              199              DC    X'404001'
  00012D E2E4D4D4C57A                        200     M2       DC    C'SUM:'
  000133 40404040404040                      201     RESUL    DC    CL7' '
                         202     *
  00013A 000A                                203     ZEHN     DC    H'10'
  00013C F0F0                                204     NULL     DC    C'00'
  00013E 0000000C                            205     TOTAL    DC    PL4'0'
  000142 F0                                  206     ZONE     DC    X'F0'
                         207     *
  000144 0034                                208     MESS3    DC    Y(L'M3+5)
  000146 404001                              209              DC    X'404001'
  000149 C5E240D2D6C5D5D5                    210     M3       DC    C'NO MORE THAN 10 NUMBERS CAN BE PROCESSED'
  000000                                     211              END   SUM
  000178 00000001                            212                    =F'1'
  00017C 9101221427002852                      213                     =X'9101221427002852' CONSISTENCY CONSTANT FOR
AID
 FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
 HIGHEST ERROR-WEIGHT : NO ERRORS
 THIS PROGRAM WAS ASSEMBLED BY ASSEMBH                    V1.0B00     ON 1991-11-05 AT 11:07:54
```

6.2 Test run

Step 1

The Assembler source program SUM in the file SOURCE.TEST is assembled using ASSEMBH. The input TEST-SUPPORT=YES causes ASSEMBH to create LSD information and pass it to the object module. The source program is assembled without errors.

```
/DEL-SYS-FILE OMF
/START-ASSEMBH

%  BLS0500 PROGRAM 'ASSEMBH', VERSION '1.XXXX' OF 'yy-mm-dd' LOADED.
%  BLS0552 COPYRIGHT (C) FUJITSU TECHNOLOGY SOLUTIONS   2009. ALL RIGHTS
RESERVED
%  ASS6010 V 1.XXXX OF BS2000 ASSEMBH READY

//COMPILE SOURCE=SOURCE.TEST,
      TEST-SUPPORT=YES

%  ASS6011 ASSEMBLY TIME: 80 MSEC
%  ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
%  ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
%  ASS6006 LISTING GENERATOR TIME: 102 MSEC

//END

%  ASS6012 END OF ASSEMBH
```

Step 2

Program SUM is to be executed.

```
/START-PROG (*OMF)

%  BLS0001 *** DBL VERSION 070 RUNNING ***
%  BLS0517 MODULE 'SUM' LOADED

PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END: 00
*05
*16
*48
*00
*0
*00
*EN
/
```

The program always branches back to input, therefore a program error must exist. The program is interrupted by pressing the K2 key.

Step 3

The program is reloaded with TEST-OPTION=AID so that it can be symbolically tested.

```
/LOAD-PROG (*OMF),TEST-OPTION=AID

%  BLS0001 *** DBL VERSION 070 RUNNING ***
%  BLS0517 MODULE 'SUM' LOADED

/%IN S'96' <%D INPUT;%STOP>
/%R
```

The %INSERT command is used to set a test point at the line with the statement number 96, i.e. the CLC instruction. Every time the program reaches this address, the contents of field INPUT are to be output.
Following output, the program is to be switched to the STOP status so that new commands can be entered.

The loaded program is started with %RESUME.

```
PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END: 00
*05

** ITN: #004B012E'***TSN:2069*******************************************'
SRC_REF:    96 SOURCE: SUM       PROC: SUM   ********************************
INPUT           = 00060000 F0F5   ....05
STOPPED AT LABEL: COMP , SRC_REF:  96, SOURCE: SUM ,PROC: SUM

/%R
*48
INPUT           = 00060000 F4F8   ....48
STOPPED AT LABEL: COMP , SRC_REF:  96, SOURCE: SUM ,PROC: SUM

/%R
*16
INPUT           = 00060000 F1F6   ....16
STOPPED AT LABEL: COMP , SRC_REF:  96, SOURCE: SUM ,PROC: SUM

/%R
*00
INPUT           = 00060000 F0F0   ....00
STOPPED AT LABEL: COMP , SRC_REF:  96, SOURCE: SUM ,PROC: SUM
```

Field INPUT contains the correct value in each case. The program obviously does not recognize the end criterion.

Step 4

The %DISASSEMBLE command specifies that 5 lines are to be output in "retranslated" format starting at line 96, i.e. the CLC instruction.

```
/%DA 5 FROM S'96'

SUM+62          CLC   121(6,R2),13A(R2)      D5 05 2121 213A
SUM+68          BC    B'1000',7A(R0,R2)      47 80 207A
SUM+6C          PACK  123(2,R2),121(2,R2)    F2 11 2123 2121
SUM+72          AP    13C(4,R2),123(2,R2)    FA 31 213C 2123
SUM+78          BC    B'1111',28(R0,R2)      47 F0 2028
```

This shows that the length field of the CLC instruction contains ’6’ instead of ’2’. This is why the end criterion is not recognized.

The correct Assembler instruction reads:

```
COMP   CLC  INPUT+4(2),NULL
```

Step 5

This error can be provisionally amended by means of the %SET command. The program is reloaded for this purpose.

```
/LOAD-PROG (*OMF),TEST-OPTION=AID

%  BLS0001 *** DBL VERSION 070 RUNNING ***
%  BLS0517 MODULE 'SUM' LOADED

/%SET X'D5012121213A' INTO COMP
/%DA 1 FROM COMP

SUM+62          CLC   121(2,R2),13A(R2)      D5 01 2121 213A

/%R
```

%SET changes the memory contents at address COMP. An AID literal with the same length as the CLC instruction and containing the length entry ’01’ instead of ’05’ is transferred. The CLC instruction is then checked using %DISASSEMBLE and the program restarted with %RESUME.

```
PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END: 00
*05
*16
*48
*12
*10
*15
*17
*19
*29
NO MORE THAN 10 NUMBERS CAN BE PROCESSED
SUM:0000171

%  IDA0N51 PROGRAM INTERRUPT AT LOCATION '000000BE (SUM), (CDUMP), EC=90'
%  IDA0N45 DUMP DESIRED? REPLY (Y=USER-/AREADUMP;Y,SYSTEM=SYSTEMDUMP;N=NO)?Y
%  IDA0N53 DUMP BEING PROCESSED. PLEASE HOLD ON
%  IDA0N54 USERDUMP WRITTEN TO FILE 'userid.DUMP.name.2069.00001'
%  IDA0N55 TITLE: 'TSN-2069  UID-userid  AC#-xxxxxxxx  USERDUMP
   PC-0000BE EC=90  VERS-100  DUMP-TIME  11:26:51  91-11-05'
```

Another program error exists, since the user has entered only 9 numbers. A dump for further diagnosis is therefore generated on program termination.

Step 6

The %DUMPFILE command opens the dump file and and assigns it the link name D1. The %BASE command switches the AID work area to the opened dump file. From now on, an address without its own base qualification will always cause the dump file data to be accessed.

```
/%DUMPFILE D1=DUMP.NAME.2069.00001
/%BASE E=D1

/%D INPUT
** D1: DUMP.NAME.2069.00001 ************************************************
INPUT          = 00060000 F2F9   ....29

/%D _R5
_R5            = 0000000B
```

The last number entered in the INPUT field is to be output. The output and log are identical.

As the number of inputs is counted in register 5, it is now queried.

Register 5 contains the value ’11’, although only 9 numbers were entered. A comparison with the assembly listing shows that register 5 has the initial value ’1’ and not ’0’.

The correct Assembler instruction reads:

```
L      R5,=F'0'
```

Step 7

This error can be provisionally amended by means of the %SET command. The program is reloaded for this purpose.

```
/LOAD-PROG (*OMF),TEST-OPTION=AID

%  BLS0001 *** DBL VERSION 070 RUNNING ***
%  BLS0517 MODULE 'SUM' LOADED

/%BASE
/%SET X'D5012121213A' INTO COMP
/%IN LOOP <%SET 0 INTO _R5;%REM%.>

/MOD-TEST-OPT DUMP=NO
/%R
```

First, %BASE is issued to assign the loaded program as the AID work area.

Reloading the program causes the corrections that have been made to be deleted. To ensure an errorfree program run, the %SET command from Step 5 is issued again here.

%INSERT sets a *test point* to the Assembler instruction with the name entry LOOOP. This means AID is to execute the following *subcmd* prior to the add instruction.

The %SET command giving register 5 the initial value ’0’ is contained in the *subcmd* of %INSERT. This *subcmd* is deleted with %REM after the first run (as no further subcommand has has been entered for this *test-point*, the *test-point* is also deleted), and the program is then resumed.

As the TERM macro is defined in the source program with the DUMP=Y operand, a dump is offered every time the program terminates. This can be prevented before the program is started (%RESUME) with the following command: /MODIFY-TEST-OPTIONS DUMP=NO

```
PLEASE ENTER UP TO 10 2-DIGIT NUMBERS. END: 00
*05
*16
*48
*12
*10
*15
*17
*19
*29
*11
NO MORE THAN 10 NUMBERS CAN BE PROCESSED
```

```
SUM:0000182

%  IDA0N51 PROGRAM INTERRUPT AT LOCATION '000000BE (SUM), (CDUMP), EC=90'
%  IDA0N47 DUMP PROHIBITED BY /OPTION COMMAND
/
```

After this correction the program executes without errors. The errors can now be definitively eliminated in the source program.

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 constant

The AID constants comprise all constants defined in the program, the statement names, the source references, the results of address selection, length selection and length function, and the AID literals. They represent a value which cannot be changed. They have no address attribute.
An address constant represents an address, i.e. statement names, source references and the result of an address selection. In conjunction with an operator pointer (->), they allow you to address the required memory location.

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, or of assigning an output file:

- %OUTFILE command with link name and file name
- ADD-FILE-LINK command with link name and file name
- For a link name to which no file name has been assigned, AID issues a FILE macro with the file name AID.OUTFILE.Fn.

An AID output file always has the format FCBTYPE=SAM, RECFORM=V and is opened with MODE=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).

Assembler procedure

This is a program unit in structured programming with an entry and an exit. It can be called using a name, with current parameters if required. An Assembler procedure starts and ends with the predefined macros @ENTR (start of procedure) and @END (static end of procedure).

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

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, through the exited subprograms and on to the main program. When symbolic debugging of Assembler programs is performed, the call hierarchy can only be observed if the program is composed of Assembler procedures, i.e. if a structured Assembler program is involved.

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 program (assembly) unit in which the program was interrupted.
The following should be noted for structured Assembler programs:
If the current program interrupt is the the ASSEMBH runtime module, the current program unit is no longer the user's assembly unit. In this case the appropriate PROG qualification must be used in AID commands.
The AID STOP message outputs the name of the program unit.

dataname

This operand stands for all names assigned for data in the source program. *dataname* can be used to address constants, data fields, predefined general registers, control sections, dummy sections, external dummy sections, dummy registers and common control sections when symbolic debugging is performed (see chapter 3).

data type

In accordance with the data type declared in the source program, AID assigns an AID storage type to each data field:

- binary string (≙ %X)
- character (≙ %C)
- numeric (≙ %F, %D)

This storage type determines how the data field is output by %DISPLAY, transferred or overwritten by %SET, and compared in the condition of a subcommand.

dump file

A disk file containing a program dump.

ESD

The External Symbol Dictionary (ESD) lists the external references of a module. It is generated by the Assembler 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).

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 AID 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 Assembler-generated source references (statement numbers). The LSD records are created by the Assembler. 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 ASSEMBH. A type modification in %DISPLAY and %SDUMP causes the output type to be changed as well.

predefined macros

These are used in structured programming with ASSEMBH. The first character of a predefined macro is always "@".

ASSEMBH provides the user with a set of predefined macros which can be used for structured programming. ASSEMBH must be notified of the relevant macro library when assembly runs for Assembler programs containing these macros take place. The runtime system must be available for running the programs.

program state

AID makes a distinction between three program states which the program being tested may assume:

- The program has stopped.

  %STOP, the K2 key, the BKPT macro or completion of a %TRACE interrupted the program. The task is in command mode. The user may enter commands.

- 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.

- 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

This is an assembly unit.
ASSEMBH generates an object module for each assembly unit.
The name of an assembly unit is the name of the first control section named (START or CSECT statement) or the name of the first Assembler procedure (@ENTR macro).

qualification

A qualification is used to reference an address which is not in the current AID work area. 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 field of the current program unit.

source reference

designates a statement number allocated by the Assmebler in the STMNT column of the assembly listing. Every named executable Assembler instruction and every predefined macro (@ macro) can be referenced.
The source reference is specified with *S'stmt-no'*.
stmt-no is an integer between 1 and 2^{31}-1.

statement name

designates the address of an executable Assembler instruction or of a call of a predefined macro (@ makro).
The statement name is specified with *L'name'*.
name is the name entry of an Assembler instruction or of a call of a predefined macro (@ macro) and can be up to 64 characters in length.
name is shortened to 32 characters by AID.
name can also be entered without L'...' if it cannot be confused with a data name in a command.

statement number

In the Assembler source program, each instruction and each comment is regarded as a statement and is given a statement number in the assembly listing. If an Assembler instruction extends over more than one line in the listing, the entire instruction still has only one statement number. The statement number of Assembler instructions generated by macros can be seen in the assembly listing if PRINT GEN was used for assembly.
During symbolic debugging, the statement number can be used to reference all Assembler instructions with a name in the name entry and all predefined macros (@ macros).

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, %P, %E, %D, %F and %A (see %SET and AID Core Manual, chapters 6 and 9).

structured Assembler programs

Structured Assembler programs comply with the rules for structured programming with ASSEMBH. They are programmed with the aid of predefined macros and are composed of Assembler procedures.

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 of FORTRAN Programs
User Guide

[5] **AID** (BS2000)
Advanced Interactive Debugger
Debugging under POSIX
User Guide

[6] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of C/C++ Programs
User Guide

[7] BS2000 OSD/BC
Executive Macros
User Guide

[8] BS2000 OSD/BC
Programmiersystem*
Technische Beschreibung
(Programming System, Technical Description)

[9] **ASSEMBH** (BS2000)
User Guide

[10] **ASSEMBH** (BS2000)
Reference Manual

[11] **XHCS**
8-Bit Code and Unicode Processing in BS2000
User Guide

Index