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 always 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. During symbolic debugging, data, statements and program sections can be addressed using the names declared in the source code, and statements without names can be addressed using the source reference generated by the compiler. 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.

AID V2.1A can be installed in BS2000 versions as of V10; ESA support and write monitoring are provided as of BS2000/OSD V1.0.
1.1 Target group

AID is the interactive debugging aid for all software developers and diagnostic engineers who work in BS2000 with the programming languages ASSEMBH, COBOL, FORTRAN, C, C++ or PL/I and who wish to test and also correct programs. This manual is aimed at people wishing to debug COBOL programs.

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. For experienced AID users there is also a Reference Guide [11], showing the syntax of the commands and operands with brief explanations. The Reference Guide also contains the %SET tables and a comparison of AID and IDA. 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 AID messages, BS2000 commands not permitted in command sequences, operands supported for the last time, and a comparison of AID and IDA.

AID User Guides

The User Guides contain list of the commands in alphabetical order. All simple memory references are described in these Guides. In addition to this manual

AID - Debugging of COBOL Programs

the following other User Guides are available:

AID - Debugging of FORTRAN Programs [3]
AID - Debugging of PL/I Programs [4]
AID - Debugging of ASSEMBH Programs [5]
AID - Debugging of C and C++ Programs [10]

In the language-specific manuals, the description of the operands is tailored to fit the programming language in question. A prerequisite for this is that the user knows the particular language scope and operation of the relevant compiler.
The additional options for debugging on the machine code level are described in AID - Debugging on Machine Code Level [2].

This manual is required for debugging 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.

1.3 Changes made since AID V2.0A

In chapter 5 the texts appearing in each address operand, for example relating to the qualifications, statement names etc., have been limited where possible. Initially therefore it will certainly be necessary relatively often to the section on COBOL-specific addressing. After some experience with the product it is then preferable if the repetitions are omitted.

A description of the %AINT command has also been included in this manual.

The %SHOW command, via which information can be requested about current commands, operand values and subcommands, is also described.

Keywords %AMODE, %LINK and %MAP have been included.

%DISPLAY %LINK outputs the name of the segment that was dynamically loaded with the last successful %LPOV event (see %ON).

%DISPLAY %MAP allows the user to output an overview of contexts, with the COMMONs and CSECTs. In contrast with the %SORTEDMAP keyword, long names are output unabbreviated.

Keywords %IFR, %IMR and %ISR are no longer described.

The context name and existing COMMON areas are now also included in AID outputs.

The CTX qualification is described briefly in chapter 4, but for this AID version it has not been included in the description of the address operands. Supplementary information, including about use on machine code level, is given in the AID Core Manual, section 7.1.

The S and PROC qualifications have been included in the syntax of the address operands.

For programs which have been compiled with COBOL85 compilers ≤ V2.1A and which have a name with more than 8 characters in the PROGRAM-ID, it is no longer possible to use the PROG qualification, only a combination of S qualification and PROC qualification. The effect of the SDF operand for compiler control, STMT-REFERENCE, on the generation of the source reference is described in more detail.

Write access to a storage location can be monitored with %ON. The %WRITE event (memref) was introduced with AID V2.0B for this purpose. In contrast with the first version, it is now possible to monitor an area of 64Kbytes.

For the %ON command the order of processing when several events occur at the same time has changed.

LMS correction statements are now only generated in SDF format and they can
only be processed with LMS ≥ V 2.0A.
The message code for AID messages has been changed from In to AID0n. There are
now also AID message texts in German. The messages can now be queried with
/HELP-MESSAGE-INFORMATION. In this version it is still possible to address the messages
with the old message code In using the AID command %HELP.
2 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.

**lowercase italics**
Operand names in the continuous text of the manual appear in *lowercase italics*.

```
[ 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.
3 Prerequisites for symbolic debugging

For symbolic debugging, AID requires a "List for Symbolic Debugging" (LSD) which contains the symbolic names defined within the program. This LSD information is generated by the compiler and can be taken over during linking, and also loaded. AID also offers the option if necessary of dynamically loading the LSDs if they have been stored by the compiler in a PLAM library. This LSD information is generated by the compiler and taken over by the linking loader or the static binder and starter. The control statements for creating the LSDs by the COBOL85 compiler are described in brief below. General information on LSD records and on linking, loading and starting is given in chapter 4 of the AID Core Manual.

3.1 Compilation

As of V1.2A, the COBOL85 compiler can be controlled in two ways:

- via SDF options or
- via COMOPT statements.

The COBOL85 compiler generates LSD information in accordance with the control option selected when the following operands are entered:

SDF control

/START-COBOL85-COMPILER ...,TEST-SUPPORT = AID{(...)}

Further options which influence debugging with AID can be specified in the parentheses after "AID":

- The STMT-REFERENCE option specifies whether the source references are to be formed from the line numbers contained in the source program (columns 1-6) or from the line numbers assigned by the compiler.
- The PREPARE-FOR-JUMPS option determines whether dummy commands are to be generated in the procedure division for every start of paragraph or section so that the AID command %JUMP can be used.
- The SHARABLE-CODE option defines whether the PROCEDURE DIVISION code (without DECLARATIVES) is to be written into a separate object module.
**Prerequisites for symbolic debugging**

**COMOPT control**

/START-PROGRAM $COBOL85
...
COMOPT SYMTEST=ALL

The other COMOPT statements which can influence debugging with AID are:

COMOPT TEST-WITH-COLUMN (corresponds to SDF option STMT-REFERENCE)
COMOPT SEPARATE-TESTPOINTS (corresponds to SDF option PREPARE-FOR-JUMPS)
COMOPT GENERATE-SHARED-CODE (corresponds to SDF option SHARABLE-CODE)

A detailed description of the corresponding operands is given in the COBOL85 User Guide [9].

**Segmented or shareable programs**

Normally COBOL85 also generates an object module from a source program. However, additional object modules are generated for segmented and shareable programs.

**Segmentation**

The sections in COBOL programs are defined by a system of segment numbers. The segment number is contained in the section header (SECTION). Segment numbers 50 through 99 designate independent segments. Separate object modules are generated for these segments.

The name of such a module comprises the PROGRAM-ID, abbreviated to 6 characters if required, to which the segment number is appended. This name is referred to as a *segmentname* in the AID syntax.

To allow AID to take the overlay structure of a program into consideration, you must enter %AID OV=YES.

**Shareability**

If a number of users (tasks) are to access individual sections of a COBOL program, the program sections can be made shareable. COBOL85 enables the requisite object modules to be generated using SDF or COMOPT control.

The name of such a module comprises the PROGRAM-ID, abbreviated to 7 characters if required, to which the character @ is appended. This name is referred to as a *sharename* in the AID syntax.
3.2 Linking, loading and starting

You link, load and start compiled programs with the SDF commands and TSOSLNK statements valid for all languages. They are described in chapter 4 of the Core Manual, where you will also find everything about the parameters which have the effect that the LSD information generated by the compiler is passed to the linkage editor (BINDER), the static linkage editor (TSOSLNK) or dynamic linking loader (DBL) or to the static starter (ELDE), to enable symbolic debugging to be carried out. There is also the possibility of dynamically loading LSDs from a PLAM library with the aid of the %SYMLIB command.

3.3 Commands at the start of a debugging session

Immediately after it has been loaded, the program is in the PROCEDURE DIVISION before the first statement and no initializations have yet been carried out. Individual commands may therefore result in an error message, and in other cases it may be necessary to specify qualifications, which can be avoided if a %TRACE 1 is used to run the program in response to the first statement in the program.

If you have previously debugged a program in a programming language which does not allow the hyphen in names or which uses lowercase letters in names, you should first enter the %AID SYMCHARS or %AID %LOW=OFF command as appropriate or view the current settings of global parameters using %SHOW %AID (see chapter 5).

In order to be able to interrupt a relatively long AID output with the K2 key, the option must be set with the following command:

/modify-terminal-option overflow-control=user-acknowledge
COBOL-specific addressing

This chapter describes the memory references used for symbolic debugging of COBOL programs. For a general description of addressing methods please refer to the AID Core Manual, chapter 7. The symbolic memory references (see section 4.2) to be used are all names of files, data and statements from the program as contained in the LSD records, and the source references generated by the compiler. It may be necessary to precede them by qualifications, as described below.

In all operands in which it is possible to use `compl-memref` it is permitted to switch as the need arises between the memory references described in this manual and those for debugging on machine code level [2].

4.1 Qualifications

Qualifications are used when a memory object is not located within the current AID work area or is not unique in that area, or in order to identify a subarea. There are two types of qualification: the base qualification, by means of which the AID work area is defined, and the area qualifications, by means of which parts of the work area are addressed. The path to an area or to a memory object is also described by linking qualifications.

Qualifications are delimited by periods. Likewise a period must be inserted between the final qualification and the following operand.

Base qualification

\[ E=\{VM|Dn\} \]

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). It is described in the AID Core Manual, chapter 7, and under the %BASE command. A base qualification can be immediately followed by area qualifications or a file name, data name, special register, figurative constant, statement name, source reference or complex memory reference.
Area qualifications

These qualifications are used to identify a part of the work area. If an address operand ends with one of these qualifications, the command relates only to the part that is identified by the last qualification. An area qualification delimits the area in which a command takes effect, or it renders a data name or statement name unique within the work area, or it makes it possible to reach a name that would otherwise not be addressable at the current interrupt point.

CTX=context
The CTX qualification designates a context (see AID Core Manual, section 7.1 [1]). It can only precede an S qualification. An address operand can only end with a CTX qualification in the %SDUMP and %QUALIFY commands. This qualification is required if it is intended to address a compilation unit or CSECT which does not contain the current interrupt point and which is contained in a number of contexts. context is the name of the context as explicitly assigned in the BIND macro or the implicitly assigned name LOCAL#DEFAULT. Programs that are loaded with the DBL are also given the context name assigned as the default option, LOCAL#DEFAULT. If they were linked with TSOSLNK, the name of the context isCTXPHASE. Other program contexts may result from connection to a shared code program.

So as to prevent further inflation of the syntax for the address operands of the individual commands, the CTX qualification was not included there, particularly as they currently tend to be used only rarely. The AID Core Manual, chapter 7.1, contains further information, also in relation to debugging on machine code level.

Examples

%CONTROL1 IN CTX=LOCAL#DEFAULT.S=MAIN.PROC=PART
The control-area in this case is not in the current context in which the program was interrupted but in the LOCAL#DEFAULT context.

%SDUMP CTX=CTXPHASE
The current interrupt point is in a different context in the call hierarchy. In this %SDUMP the command is limited to the specified context.

%INSERT CTX=LOCAL#DEFAULT.S=SOURCE.PROC=UNDER.UNDER
The compilation unit SOURCE is both in the current context and in the LOCAL#DEFAULT context. A context qualification is needed in order to be able to define the test point.
S=srcname

The S qualification designates a compilation unit.

`srcname` is formed during compilation, from the program name in the PROGRAM-ID of a "complete" COBOL program (see Language Reference Manual "COBOL Compiler", chapter on "Program communication" [9], or User Guide "COBOL Compiler", chapter on "Compiler output" [8].)

`srcname` may have up to 8 characters when designating an object module (OM) and up to 30 characters for a link and load module (LLM). If an `srcname` ending with a hyphen is produced for an object module as a result of truncation, the S qualification must be written as follows: `S=N'srcname'`

If it cannot locate the 30-character name of a link and load module, AID makes a further attempt with the name abbreviated to the first eight characters.

PROC=program-id [ •program-id ]

The PROC qualification designates a COBOL program. It may be a single program or the outermost or an outer or inner program of a nested program.

`program-id` consists of the maximum of 30 characters of the name from the the PROGRAM-ID in the source program.

Operands specifying an address area (%CONTROL, %TRACE) or a name range (%SDUMP) can end with the PROC qualification. The address range or name range then encompasses the entire program. Otherwise you specify the PROC qualification if you address a name in the LSD records which is not contained in the current program or is not unique in the compilation unit, i.e. in front of a file name, data name, statement name or a complex memory reference if the latter begins with a name.

•program-id

If the name of a program is repeated directly after a PROC qualification, the user is thus designating the address of the first program statement which can be executed. If the current interrupt point is in the same program, the PROC qualification can be omitted. This specification can be used in %DISASSEMBLE and %INSERT.

PROG=program-id [ •program-id ]

This area qualification is a combination of the S and PROC qualification. It can only be used if the names of the compilation unit and of the program are identical, i.e. for an "outermost" program. In that case the same applies as to the PROC qualification.

The PROG qualification cannot be used if `program-id` is more than 8 characters long.
COBOL-specific addressing

The C qualifications listed below switch to the machine code level. They cannot be followed directly by a symbolic operand (see section on "Symbolic memory references"), only a compl-memref (see AID Core Manual, section 7.5 [1]). Nevertheless, AID expects or adds a symbolic criterion in %CONTROLn or %TRACE. Only an E qualification, and if appropriate a CTX qualification, can be placed in front of a C qualification.

\[ C=\text{segmentname} \]

This identifies a segment. \textit{segmentname} is composed of the first 6 places of the PROGRAM-ID and the segment number from the section header.

This C qualification allows you to define a segment as an area in %CONTROLn, %FIND, %ON write-event or %TRACE, or to declare the start address of the segment as \textit{start} in %DISASSEMBLE or \textit{test-point} in %INSERT.

\[ C=\text{sharename} \]

This identifies a module that has been compiled with the SDF option SHAREABLE-CODE=\texttt{YES}. It therefore designates an object module. \textit{sharename} is composed of the first 7 places of the PROGRAM-ID and the character \texttt{@}.

This C qualification can be used to define the object module as an area in %CONTROLn, %FIND, %ON write-event or %TRACE provided it is loaded in class 6 memory. Modules loaded in class 4 memory cannot be addressed with this C qualification.

4.2 Symbolic memory references

Symbolic memory references may include all file, data and statement names from the program which are contained in the LSD records, as well as the source references generated by the compiler and the AID keywords. No LSD records are generated for 88 levels, the NATIVE alphabet and for definitions from the REPORT-SECTION (apart from LINE-COUNTER, CBL-CTR and PAGE-COUNTER). Consequently you cannot access this data with AID.

All symbolic memory references can be subjected to the operations described in chapter 7 of the AID Core Manual. All operands in which that is possible contain the entry compl-memref. The user can then arbitrarily switch between the memory references as described in this manual and those for debugging on machine code level (see [2]).
COBOL-specific addressing

filename

is the name of a file from a file definition in the FILE-SECTION of the DATA DIVISION.
AID outputs the following information in response to the %DISPLAY and %SDUMP commands: the file status and, if the file is open, the contents of the data record area and any record key. In addition, the address and length selector can be used on filename.

dataname

stands for all the names of data items defined in the DATA DIVISION in the source program, for the COBOL special registers and the figurative constants. Data items can be data records, group items and tables, or elements in these. They can be identified and indexed.

dataname is an alphanumeric string up to 30 characters in length. It can be specified in all commands for output and modification of information; these are the %DISPLAY, %MOVE, %SDUMP and %SET commands, but also the %FIND command (search for a string) and the %ON command (write monitoring).

dataname [identifier][...] [(index[,...])] identifier

If dataname is not unambiguous within a program unit, it can be identified by being assigned to a particular data item with IN or OF. dataname must be assigned as many identifiers as are required to designate it unambiguously. If it is not identified, there must be a definition on level 01 or 77 which AID then processes, otherwise an error message will be issued.

identifier is specified as follows:

\[
\begin{align*}
\{ & \text{IN} \\
\text{OF} & \}
\end{align*}
\]

data-item-name

index

If dataname is the name of an element in a table, it can be indexed and subscripted as in a COBOL statement. In contrast to COBOL, multiple indexes have to be separated by a comma.

Data definitions which are subordinate to a dataname with an OCCURS clause must be assigned as many indexes in the %SET or %SDUMP as have to be specified for access in a COBOL statement. The index entry for the data name that is addressed with dataname can be omitted from the %DISPLAY, %FIND and %MOVE, and it is then only necessary to specify index entries for higher index levels (see example). Otherwise it is possible to specify a dataname without index in the %DISPLAY, %FIND and %MOVE if the dataname was itself defined with the OCCURS clause. This has the effect of addressing all elements with that name.
index is specified as follows:

\[
\begin{align*}
\text{n} & \quad \text{index-name} \\
& \quad \text{data-name} \quad \text{TALLY} \\
& \quad \text{arithmetic-expression}
\end{align*}
\]

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

index-name

is the symbolic name defined in the INDEXED BY clause for indexing a table level.

data-name

designates a numerical data item (not floating point) from the DATA DIVISION that can be identified. It must be contained in the same program unit as the table.

TALLY

is the special register generated by the COBOL compiler for each program.

arithmetic-expression

AID calculates the value for index. Valid entries are the arithmetic operators (+, -, /, *) and the above-listed operands \( n \), data-name and TALLY. index-name can only be combined with \( n \) and may only be used to index the table level to which it was assigned via the INDEXED BY clause.

Example

01 TABLE.  
  02 GROUP1 OCCURS 10.  
    04 ELEMENT1 PIC X(5).  
    04 ELEMENT2 PIC 9(2) OCCURS 6.  
  02 GROUP2 PIC 9(2) OCCURS 12.  
  01 FIELD PIC X(70).  
  01 INPUT-STRUCTURE.  
    02 GR1.  
      04 ELEM1 PIC X(5).  
      04 ELEM2 PIC 9(2).  
    02 GR2 PIC 9(2) OCCURS 12.  

The various data names can be addressed in an AID command in the following way:

%DISPLAY GROUP1

All elements in GROUP1 are output.

%MOVE GR1 INTO GROUP1

This command overwrites the entire vector GROUP1 with the contents of GR1.

%MOVE GR1 INTO GROUP1(1)

The first element of GROUP1 is overwritten.

%MOVE GROUP2 INTO GR2
The entire contents of GROUP2 are transferred to GR2. It is not possible, on the other hand, to write the following command:

```cobol
%SET GROUP2 (1) INTO GR2
```

Full indexing is required in the `%SET` command, as in COBOL statements.

```cobol
%SET GROUP2 (1) INTO GR2 (12)
%SET GR2 (ELEM2) INTO ELEMENT2 (5,ELEM2)
```

COBOL special registers

Only those special registers may be specified that have been created by the COBOL compiler for the program and that have already been supplied with the current values. For instance, SORT special registers may be specified here only if the program contains a sort section.

- LINAGE-COUNTER
- RETURN-CODE
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY

Figurative constants

dataname is one of the COBOL names for figurative constants or one of the names defined for literal and symbolic character in the SPECIAL-NAMES paragraph. HIGH-VALUE and LOW-VALUE always represent the value that corresponds to them by default or in accordance with the definition in the PROGRAM COLLATING SEQUENCE clause.

- ZERO
- SPACE
- HIGH-VALUE
- LOW-VALUE
- QUOTE
- literal
- symbolic character

statement-name

designates the address of the first instruction in a section or paragraph in the PROCEDURE DIVISION.

```
L'section'
{L'paragraph' [IN L'section']}
```

In the `%CONTROLn`, `%DISASSEMBLE`, `%INSERT`, `%JUMP` and `%TRACE` commands, an alphanumeric section or paragraph name can be specified without L’...’ since in these commands this name cannot be confused with a data name. If in a complex memory reference statement-name is followed by a pointer ( -> ), the L’...’ format must be used. If a paragraph name is not unambiguous within a
program, it must be identified by the section name of the section in which it was defined: L’paragraph’ IN L’section’

You thus define the address in the %DISPLAY, %FIND, %MOVE and %SET commands. %DISASSEMBLE, %INSERT and %JUMP are used to define the memory location at this address. %CONTROLn and %TRACE are used to define the entire section or the entire paragraph.

source-reference
is an address constant for the compiler-generated designation of a statement. Its structure varies in accordance with the SDF option TEST-SUPPORT with the operand STM-REFERENCE.

STM-REFERENCE=LINE-NUMBER
S’n[verb[m]]’

n is the line number in the PROCEDURE DIVISION, assigned by the compiler. It is not permitted to enter leading zeros. In this case the source-reference is unambiguous within a compilation unit. You specify S’n’ for lines with paragraph or section names only if no COBOL verb is present.

verb
is the defined abbreviation of a COBOL verb in the statement line designated with n.
You specify S’nverb’ for lines containing a COBOL verb.

m
is a single-digit number > 1. m is specified only if the same COBOL verb appears more than once in a line and the first COBOL verb is not to be addressed. It is thus declared to be the m-th COBOL verb within the line.

STM-REFERENCE=COLUMN1-TO-6
S’xverb[m]’
x is the unchanged contents of columns 1 to 6 of a source program line. Any blanks included must be specified.
If a source code line cannot be uniquely identified by x...x within the compilation unit as a whole, the source reference is not unambiguous either. Paragraphs and sections cannot be addressed via a source reference in this case.

verb
is the designated abbreviation of a COBOL verb in the statement line identified by x. S’xverb’ must be specified for lines containing a COBOL verb.
m is a single-digit number > 1.
It identifies the m-th COBOL verb within a line. A line in this case is understood to
be all statements up to a new line number.

In %FIND and %ON write-event, the source reference must be followed by the
pointer operator. This identifies four bytes of the machine code starting from the
address that is stored in the address constant. You thus define the address in the
%DISPLAY, %MOVE and %SET commands. %DISASSEMBLE, %INSERT and
%JUMP are used to define the memory location at this address. In the
%CONTROLn and %TRACE commands you can define an area using two source
references.

Example

%DISPLAY S’95ADD2’
The program was compiled with STM-REFERENCE-LINE-NUMBER. The source
reference specifies the address associated with the second ADD statement stored
in line 95 in the LSD records. This is the address of the memory location of the
first command generated for this statement.

List of COBOL verbs and their abbreviations:

<table>
<thead>
<tr>
<th>COBOL Verb</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
</tr>
<tr>
<td>ADDC</td>
<td>ADD CORRESPONDING</td>
</tr>
<tr>
<td>ALT</td>
<td>ALTER</td>
</tr>
<tr>
<td>CALL</td>
<td>CALL</td>
</tr>
<tr>
<td>CANC</td>
<td>CANCEL</td>
</tr>
<tr>
<td>CLO</td>
<td>CLOSE</td>
</tr>
<tr>
<td>COM</td>
<td>COMPUTE</td>
</tr>
<tr>
<td>CON</td>
<td>CONNECT</td>
</tr>
<tr>
<td>CONT</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>DEL</td>
<td>DELETE</td>
</tr>
<tr>
<td>DIS</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>DIV</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>DSC</td>
<td>DISCONNECT</td>
</tr>
<tr>
<td>ENTR</td>
<td>ENTRY</td>
</tr>
<tr>
<td>ERA</td>
<td>ERASE</td>
</tr>
<tr>
<td>EVAL</td>
<td>EVALUATE</td>
</tr>
<tr>
<td>EXI</td>
<td>EXIT</td>
</tr>
<tr>
<td>EXIT</td>
<td>EXIT PROGRAM</td>
</tr>
<tr>
<td>FET</td>
<td>FETCH</td>
</tr>
<tr>
<td>FIN</td>
<td>FINISH</td>
</tr>
<tr>
<td>FND</td>
<td>FIND</td>
</tr>
<tr>
<td>FRE</td>
<td>FREE</td>
</tr>
<tr>
<td>GEN</td>
<td>GENERATE</td>
</tr>
<tr>
<td>GET</td>
<td>GET</td>
</tr>
<tr>
<td>GOT</td>
<td>GO TO</td>
</tr>
<tr>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>INIT</td>
<td>INITIALIZE</td>
</tr>
<tr>
<td>INI</td>
<td>INITIATE</td>
</tr>
<tr>
<td>INS</td>
<td>INSPECT</td>
</tr>
<tr>
<td>KEE</td>
<td>KEEP</td>
</tr>
<tr>
<td>MOD</td>
<td>MODIFY</td>
</tr>
<tr>
<td>MOV</td>
<td>MOVE</td>
</tr>
<tr>
<td>MOVCC</td>
<td>MOVE CORRESPONDING</td>
</tr>
<tr>
<td>MRG</td>
<td>MERGE</td>
</tr>
<tr>
<td>MUL</td>
<td>MULTIPLY</td>
</tr>
<tr>
<td>OPE</td>
<td>OPEN</td>
</tr>
<tr>
<td>PER</td>
<td>PERFORM or EXIT PERFORM</td>
</tr>
<tr>
<td>REA</td>
<td>READ</td>
</tr>
<tr>
<td>REDY</td>
<td>READY</td>
</tr>
<tr>
<td>REL</td>
<td>RELEASE</td>
</tr>
<tr>
<td>RET</td>
<td>RETURN</td>
</tr>
<tr>
<td>REW</td>
<td>REWRITE</td>
</tr>
<tr>
<td>SEA</td>
<td>SEARCH</td>
</tr>
<tr>
<td>SET</td>
<td>SET</td>
</tr>
<tr>
<td>SOR</td>
<td>SORT</td>
</tr>
<tr>
<td>STA</td>
<td>START</td>
</tr>
<tr>
<td>STO</td>
<td>STOP</td>
</tr>
<tr>
<td>STOR</td>
<td>STORE</td>
</tr>
<tr>
<td>STRG</td>
<td>STRING</td>
</tr>
<tr>
<td>SUB</td>
<td>SUBTRACT</td>
</tr>
<tr>
<td>SUBC</td>
<td>SUBTRACT CORRESPONDING</td>
</tr>
<tr>
<td>TER</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>UNST</td>
<td>UNSTRING</td>
</tr>
<tr>
<td>WRI</td>
<td>WRITE</td>
</tr>
</tbody>
</table>
5 AID commands
%AID

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

- With **CHECK** you define whether an update dialog is to be initiated prior to execution of the %MOVE or %SET commands.
- With **REP** you define whether memory updates of a %MOVE command are to be stored as REPs.
- With **SYMCHARS** you define whether AID is to interpret a "-" in program, data and statement names as a hyphen or as a minus sign.
- With **OV** you direct AID to take the overlay structure of a program into account.
- With **LOW** 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.
- With **DELIM** you define the delimiters for AID output of alphanumeric data. The vertical bar is the default delimiter.
- With **LANG** you define whether AID is to output %HELP information in English or German.

%AID

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHECK</strong></td>
<td>= {ALL</td>
</tr>
<tr>
<td><strong>REP</strong></td>
<td>= {YES</td>
</tr>
<tr>
<td><strong>SYMCHARS</strong></td>
<td>= {STD</td>
</tr>
<tr>
<td><strong>OV</strong></td>
<td>= {YES</td>
</tr>
<tr>
<td><strong>LOW</strong></td>
<td>= {ON</td>
</tr>
<tr>
<td><strong>DELIM</strong></td>
<td>= {C'x'</td>
</tr>
<tr>
<td><strong>LANG</strong></td>
<td>= {D</td>
</tr>
</tbody>
</table>

Declarations made using %AID remain valid until superseded by a new %AID command or until /LOGOFF or /EXIT-JOB.
%AID can only be issued as an individual command, it must never be part of a
command sequence or a subcommand.

The %AID command does not alter the program state.

CHECK

ALL

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

OLD CONTENT:
AAAAAAAA
NEW CONTENT:
BBBBBBBB
% AID0274 CHANGE DESIRED? REPLY (Y = YES; N = NO) ?

N

AID0342 NOTHING CHANGED

If Y is entered, the old memory contents 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. The old or new contents are output to SYSOUT. If SYSOUT is
reassigned, these outputs cannot be seen at the terminal. This also applies if the
%MOVE or %SET command was specified with the CMD macro and output to
SYSOUT has been defined. In contrast, message AID0274 and, where appropriate,
AID0342 are always sent to the terminal medium.

NO

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

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

REP

YES

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

AID stores the corrections in a file with the link name F6. The MODIFY-ELEMENT
statement must then also be inserted for the LMS run. Care should 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), AID creates the AID.OUTFILE.F6 file, to which it then writes the REP. User-specific REP files must be created with access method SAM. REP files created by AID are likewise defined with access method SAM, record format V and opening method EXTEND. The file remains open until it is closed via %OUTFILE or until /LOGOFF or /EXIT-JOB.

**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).

**OV**

**YES**

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

**NO**

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

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

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

OFF
All lowercase letters from user entries are converted to uppercase.

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). The SDF command MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE=D also allows you to receive the AID messages in German. The update dialog (see CHECK operand) is not affected by this.
The %AINT command can be used to specify whether AID is to work with 24-bit addresses or 32-bit addresses for indirect addressing. For AID, the address before the pointer operator (->) then consists of 24 or 31 bits accordingly. The addressing mode for the test object is not affected as a result.

- **aid-mode** specifies the mode of address interpretation for indirect addressing within an AID work area.

---

### Command Operand

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%AINT</td>
<td>[aid-mode] [,..]</td>
</tr>
</tbody>
</table>

%AINT is only of benefit when debugging programs on XS computers.

As the default, AID interprets indirect address specifications according to the current addressing mode for the test object. Specification of %AINT with the keyword %MODEn deactivates automatic adaptation in this way. The test object’s addressing mode can be interrogated with %DISPLAY %AMODE. It can be changed with %MOVE. %SHOW %AID or %SHOW %BASE reveals the addressing mode valid for the current AID work area, in addition to other information.

If no qualification is specified, %AINT applies to AID commands which reference or use indirect addresses in the current AID work area.

An %AINT without operands switches back to the default address interpretation. The same effect is achieved by %AINT with a base qualification and without %MODEn. Otherwise the declared addressing mode applies until /LOGOFF or /EXIT-JOB.

%AINT does not change the program state.

**aid-mode** defines how indirect addresses are to be interpreted in subsequent AID commands, applicable in the current AID work area or the work area identified by the specified base qualification.

If a keyword is specified for address interpretation but no qualification is specified, the %AINT command applies to the processing of the current AID work area.

If a base qualification is specified but no keyword for address interpretation, the default AID address interpretation applies in the corresponding AID work area.
aid-mode-OPERAND

\[ \text{[•][E=\{VM\}][\{\text{%M[ODE]31}\}]} \]

- If a period is placed at the beginning, it is an identifier for a prequalification. It must have been defined via a previous %QUALIFY command. A period must be placed between a base qualification and the keyword for address interpretation.

\[ \text{VM} \]

This is specified if it is not intended that the change in address should apply to the current AID work area. If only a base qualification is specified, the default address interpretation applies again for the area which this addresses.

\[ \text{%M[ODE]31} \quad \text{%M[ODE]24} \]

Keyword specifying how many bits are to be taken into account in indirect addressing in AID commands.

\[ \text{%M[ODE]31} \quad \text{31-bit addressing.} \]
\[ \text{%M[ODE]24} \quad \text{24-bit addressing.} \]

**Examples**

The contents of address V'100' are: 1200000C
The contents of register 5 are: 010001A0

1. %AINT %MODE24
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
   The %AINT command has the effect of switching to 24-bit address interpretation. The switch applies to the current AID work area. The %DISPLAY outputs 4 bytes starting at address V'00000C'. The %MOVE transfers 4 bytes starting from address V'0001A0' to AID register 5.

2. %AINT %MODE31
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
   Address interpretation for the current AID work area is switched to 31-bit interpretation. The %DISPLAY outputs 4 bytes starting at address V'1200000C'. The %MOVE transfers 4 bytes starting at address V'010001A0' to AID register 5.
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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%BASE</td>
<td>[base]</td>
</tr>
</tbody>
</table>

With the %BASE command the user also defines the location of the AID work area. When debugging COBOL programs, the AID work area corresponds to the area which the load 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 the non-privileged part in virtual memory which is occupied by all connected subsystems from the loaded program (AID standard work area).

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

The current base qualification is added to all memory references in a command, and also in a subcommand, immediately on 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.
The virtual memory area of the program which has been loaded is declared as the base qualification. VM is the default value.

A dump in a dump file with the link name $Dn$ is declared as the base qualification.

$n$ is a number with a value $0 \leq n \leq 7$.

Before declaring a dump file as the base qualification, the user must assign the corresponding dump file a link name and open it, using the `%DUMPFILE` command.
The %CONTINUE command is used to start the program which has been loaded or to continue it at the interrupt point or at the location specified by %JUMP.
As opposed to %RESUME, an interrupted but still active %TRACE command is not terminated by %CONTINUE, rather it is continued depending on the declarations which have been made.

Command Operand

%CONTINUE

A %TRACE command is active as soon as it has been entered. In the following cases the %TRACE command is only interrupted and can be resumed by a %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 (see section 3.3).

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.

- With criterion you may select different types of COBOL statements. If a statement of the selected type is waiting to be executed, AID interrupts the program and processes subcmd.

- With control-area you may define the program area in which criterion is to be taken into consideration.

- With subcmd 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%CONTROLn</td>
<td>[criterion],... [IN control-area] &lt;subcmd&gt;</td>
</tr>
</tbody>
</table>

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 a specific or all active %CONTROLn declarations.

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

%CONTROLn does not alter the program state.
%CONTROLn

criterion

is the keyword defining the type of the COBOL statements prior to whose execution AID is to process subcmd.
You can specify several keywords at the same time, which are then valid at the same time. Any two keywords must be separated by a comma.
If no criterion is declared, AID works with the default value %STMT, unless a criterion declared in an earlier %CONTROLn command is still valid.

<table>
<thead>
<tr>
<th>criterion</th>
<th>subcmd is processed prior to</th>
</tr>
</thead>
<tbody>
<tr>
<td>%STMT</td>
<td>Every COBOL statement</td>
</tr>
<tr>
<td>%ASSGN</td>
<td>COBOL statements which modify the contents of a data item: ADD [CORRESPONDING], COMPUTE, DIVIDE, INITIALIZE, INSPECT, MOVE [CORRESPONDING], MULTIPLY, SET, STRING, SUBTRACT [CORRESPONDING], UNSTRING</td>
</tr>
<tr>
<td>%CALL</td>
<td>CALL, PERFORM statements as well as prior to SORT/MERGE statements, since these may call an INPUT or OUTPUT procedure.</td>
</tr>
<tr>
<td>%COND</td>
<td>EVALUATE, IF and SEARCH statements and the conditional THEN, ELSE and WHEN statement branches.</td>
</tr>
<tr>
<td>%DB</td>
<td>COBOL statements for calling a database: CONNECT, DISCONNECT, ERASE, FETCH, FIND, FINISH, FREE, GET, KEEP, MODIFY, READY, STORE</td>
</tr>
<tr>
<td>%EXCEPTION</td>
<td>The conditional statement branches and their admissible negations: AT END, AT END OF PAGE, INVALID KEY, ON SIZE ERROR, ON OVERFLOW, ON EXCEPTION as well as prior to the execution of a USE PROCEDURE.</td>
</tr>
<tr>
<td>%GOTO</td>
<td>ALTER, CONTINUE, GOTO statements.</td>
</tr>
<tr>
<td>%IO</td>
<td>COBOL statements which initiate I/O operations: ACCEPT, DISPLAY, OPEN, CLOSE, DELETE, READ, REWRITE, START, WRITE, GENERATE, INITIATE, TERMINATE</td>
</tr>
<tr>
<td>%LAB</td>
<td>COBOL statements which have a section or paragraph name or which directly follow such a name.</td>
</tr>
<tr>
<td>%PROC</td>
<td>Program or module start at the beginning of the PROCEDURE-DIVISION or at ENTRY. Program or module end by the statement STOP RUN or EXIT PROGRAM.</td>
</tr>
<tr>
<td>%SORT</td>
<td>MERGE and SORT statements, RELEASE and RETURN statements.</td>
</tr>
</tbody>
</table>
control-area

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

control-area is limited to a compilation unit in programs without segmentation, and to a segment in programs with segmentation. The limitation to one segment applies only for independent segments (segment No.>50).

A control-area definition is valid until the next %CONTROLn command with the same number is issued with a new definition, until the corresponding %REMOVE %CONTROLn command is issued, or until the end of the program is reached. %CONTROLn without a control-area operand of its own results in a valid area definition being taken over. To be valid, such a control-area operand must be defined in a %CONTROLn command with the same number, and the current interrupt point must be within this area. If no valid area definition exists, the control-area comprises the current compilation unit or current segment by default.

control-area-OPERAND

[S=srcname] [[•]PROC=program-id]

IN [•][E=VM•] [S=srcname•]

C=segmentname
C=sharename

• 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).

S=srcname

This is specified if control-area is not to be included in the current compilation unit or if a declared area restriction is no longer to apply.
%CONTROLn

PROC=program-id
This is specified if control-area is not contained in the current program,
if it is to be defined with statementname and if this name is not unique in the
compilation unit
or in order to overwrite a previously valid control-area declaration. If control-area
ends with a PROC qualification, the area covers the entire program specified. This
must have been loaded at the time the %CONTROLn is entered or when the
subcommand containing the %CONTROLn is processed.

If the srcname in the S qualification is identical to the program-id, instead of these two
you need only write the PROG qualification.

Although you switch to machine code level with the following C qualifications, as the
next step you can only select a criterion from the preceding table or AID will insert the
default %STMT.

C=segmentname
This declares the designated segment for the control-area. It is only required if the
interrupt point is not in this segment or if a previous area limitation applying to
parts of this segment is to be removed.

C=sharename
This declares the designated object module for the control-area. It need only be
specified if the interrupt point is not in the specified object module or if an area
limitation applying to the object module is to be removed.

statement-name
The control-area is defined by a statement name and comprises a section or
paragraph in the PROCEDURE DIVISION.

{L’section’
[L’paragraph’ [IN L’section’]]}

An alphanumeric section or paragraph name can be specified without L’...’ since
this name cannot be confused with a data name in this command.

If a paragraph name is not unambiguous within a program, it must be identified
by the section name of the section in which it was defined: L’paragraph’ IN
L’section’
The control-area is defined by specifying a start source reference and an end source reference. Both of these must be within the same compilation unit, where the following applies: start source reference \leq end source reference.

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

control-area cannot be limited to individual COBOL verbs within a line.

source-reference
designates the address of the first instruction generated for a statement in the PROCEDURE DIVISION and must be specified in one of the following formats:

- S’n’ for lines with paragraph or section names in which no COBOL verb occurs. This specification is not possible for programs which have been compiled with STMT-REFERENCE=COLUMN1-TO-6.
- S’nverb’ | S’xverb’ for lines containing a COBOL verb.

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

Specification of subcmd is mandatory, since AID inserts no <STOP> for %CONTROLn.

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

subcmd-OPERAND

\(<\{\text{subcmdname:} \}((\text{condition}): \}\{\text{AID-command } \}\{\text{BS2000-command } \}\{\text{;}\}\})>

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 statement lines 123 and 250 (inclusive). Statement line 123 contains no COBOL verb; statement line 250 contains the COBOL verb GO TO.

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

2. %CONTROL1 %CALL <%DISPLAY ‘CALL’ T=MAX; %STOP>

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

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

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

4. %C3 %PROC <%STOP>

   The %C3 command declares that AID is to execute a %STOP command before the first PROCEDURE DIVISION statement or the first statement following an ENTRY is executed or the module is quit or the program is terminated.

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

   %C4 is used to declare that AID is to output the first table element with the name TAB prior to the first program or module start or program or module end provided that the SLF value is less than or equal to 10.
%DISASSEMBLE

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

- The number operand enables you to determine how many instructions are to be disassembled and output.
- The start operand enables you to determine the address where AID is to begin disassembling.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%DISASSEMBLE</td>
<td>[number] [FROM start]</td>
</tr>
<tr>
<td>DA</td>
<td></td>
</tr>
</tbody>
</table>

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 (/EXEC-PROGRAM, /LOAD-PROGRAM, %BASE). AID continues disassembly at the memory address following the address last processed by the previous %DISASSEMBLE command. If number is not specified either, AID generates the same number of output lines as declared before.

If the user has not entered a %DISASSEMBLE command during a test session or has changed the test object and does not specify current values for one or both operands in the %DISASSEMBLE command, AID works with default values (10 for number and V’0’ for start). If the program was not loaded from V’0’, start must be specified.

The %OUT command can be used to control how processed memory information is to be represented and whether it is to be output to SYSOUT, SYSLST or to a cataloged file. The format of the output lines is explained after the description of the start operand.

The %DISASSEMBLE command does not alter the program state.
%DISASSEMBLE

**number**

Specifies how many Assembler commands are to be output. If no value has been specified for `number` and no value from a previous %DISASSEMBLE command applies, AID inserts the default value (10). Number is an integer with the value $1 \leq number \leq 2^{31} - 1$.

**start**

Defines the address at which disassembly of memory contents into Assembler commands is to begin. If the `start` value is not specified, AID assumes the default value V'0' for the first %DISASSEMBLE after a program is loaded. If a program has not been loaded from V'0', AID issues an error message. On every further %DISASSEMBLE, AID continues after the Assembler command last disassembled.

```
start-OPERAND

[•][qua][...]
```

- 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**

Qualifications must be specified if an address operand does not apply to the current AID work area, the current compilation unit or the program, or if it is not unique in some other way.

```
E={VM | Dn}
```

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

```
S=srcname
```

This is only specified if `start` is not to be contained in the current compilation unit.
PROC=program-id
This is only specified if start is not to be contained in the current program (see chapter 4), or if it is to be defined with statementname and this is not unique in the compilation unit.
If the srcname in the S qualification is the same as the program-id, instead of both of these only the PROG qualification should be written.

Only the base qualification or the CTX qualification can be placed before the C qualifications listed below. The C qualification takes the user away from the symbolic level. No symbolic operands can be written directly afterwards (see section on "Symbolic memory references"), only a compl-memref.

C=segmentname
The effect of this entry is to set start to the start address of the designated segment.

C=sharename
The effect of this entry is to set start to the start address of the designated object module.

program-id
This specification is possible following an explicit PROC/PROG qualification with the same program-id, or if the current interrupt point is in the program identified by program-id. The consequence is to set start at the first executable statement in the designated program.

statement-name
designates the address of the first instruction in a section or paragraph in the PROCEDURE DIVISION.

{L’section’}
{L’paragraph’ [IN L’section’]}
An alphanumeric section or paragraph name can be specified without L’...' since this name cannot be confused with a data name in this command.

If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: L’paragraph’ IN L’section’. If the user intends to follow this with a byte offset, a pointer operator ( -> ) must be entered first.
source-reference

designates the address of the first instruction generated for a statement in the
PROCEDURE DIVISION and must be specified in one of the following formats:

- S’n’ for lines with paragraph or section names in which no COBOL verb
  occurs. This specification is not possible for programs which have been
  compiled with STMT-REFERENCE=COLUMN1-TO-6.

- S’nverb[m]’ | xverb[m]’
  for lines containing a COBOL verb.

If the user intends to follow this with a byte offset, a pointer operator ( -> ) must
be entered first.

compl-memref

This should produce the start address of a machine instruction, otherwise the
disassembly obtained will be meaningless. compl-memref may contain the
following operations (see AID Core Manual, chapter 7.5 [1]):

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

If a statement name or a source reference is to be used as a memory reference,
it must be followed by a pointer operator ( -> ). In this case statementname must
be specified with L’...’. Without the pointer operator the statement name and
source reference can be used anywhere where hexadecimal numbers can be
written.

**Example:** %DISASSEMBLE L’PUTOUT’->.4
A position 4 bytes on from the first instruction in the PUTOUT section is moved to
and disassembly takes place from there.

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.

**Example:** %1G.2%AL2->
The last two bytes from AID register %1G are used as the address.

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.

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

- CSECT-relative memory address
— memory contents retranslated into symbolic Assembler notation, displacements being represented as hexadecimal numbers (as opposed to Assembler format)
— for memory contents which do not begin with a valid operation code: Assembler statement DC in hexadecimal format and with a length of 2 bytes, followed by the note INVALID OPCODE
— hexadecimal representation of the memory contents (machine code).

Example of line format with T=MAX

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

```
%DISASSEMBLE 8 FROM L'LEADER'->.4
M085+9FC UNPK O(4,R4),12C(1,R12) F3 30 4000 C12C
M085+A02 LA R4,28(R0,R3) 41 40 3028
M085+A06 LA R0,R0 1A 00
M085+A08 L R15,98(R0,R11) 58 F0 B098
M085+A0C BALR R14,R15 05 EF
M085+A0E STH R0,0(R0,R0) 40 00 0000
M085+A12 DC X'0004' INVALID OPCODE 00 04
M085+A14 DC X'0000' INVALID OPCODE 00 00
```

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 representation of the memory contents is omitted.

Example of line format with T=MIN

```
%OUT %DA T=MIN
%DISASSEMBLE 8 FROM L'LEADER'->.4
00009FC UNPK O(4,R4),12C(1,R12)
0000A02 LA R4,28(R0,R3)
0000A06 LA R0,80
0000A08 L R15,98(R0,R11)
0000A0C BALR R14,R15
0000A0E STH R0,0(R0,R0)
0000A12 DC X'0004' INVALID OPCODE
0000A14 DC X'0000' INVALID OPCODE
```

Examples

1. %DISASSEMBLE FROM PROG=EXAMPLE.OUT2 IN PUTOUT
   This command initiates disassembly of 10 instructions (default), starting with the address of the first executable instruction of paragraph OUT2 in section PUTOUT.

2. %DA 2 FROM E=D1.PROG=EXAMPLE.EXAMPLE
   Starting with the start address of the EXAMPLE program in the dump file with link name D1, two instructions are to be disassembled.
3. %DA FROM S‘45INIT’
   Since no value is specified for number, AID either inserts the default value (in the case of the first %DISASSEMBLE for this program) or takes the value from the previous %DISASSEMBLE. Disassembly starts with the first instruction generated for the statement S‘45INIT’.
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.

- With data you specify data items, their addresses and lengths, statements, data definitions, registers, execution counters of subcommands, system information, COBOL special registers and figurative constants. Here you also define AID literals or you control feed to SYSLST.

- With medium-a-quantity 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%DISPLAY</td>
<td>data {,...} [medium-a-quantity][,...]</td>
</tr>
</tbody>
</table>

A %DISPLAY command which does not have a qualification for data addresses data of the current program, if you do specify a qualification, you can access data in a dump file or in any other compilation unit or program unit which has been loaded.

If the medium-a-quantity operand is not specified, AID outputs the data in accordance with the declarations in the %OUT command or, by default, to SYSOUT, together with additional information (cf. AID Core Manual, chapter 8 [1]).

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

Immediate entry of the command right after loading the program is not recommended as not all entries in the DATA DIVISION will have been initialized (e.g. record definitions and special registers).

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

%DISPLAY does not alter the program state.
This operand defines the information AID is to output. You may output file definitions, the contents, address and length of data items and special registers, figurative constants, as well as the addresses of statements. The contents of registers and execution counters as well as the system information relevant to your program can be addressed via keywords. AID literals can be defined to improve the readability of debugging logs, and feed to SYSLIST can be controlled for the same purpose.

AID edits data items in accordance with the definitions in the source program, provided that you have not defined another type of output using a type modification (see also AID Core Manual, section 6.8 [1]). If the contents do not match the defined storage type, output is rejected and an error message is issued. Nevertheless the contents of the data element can be viewed, for instance by employing the type modification %X to edit the contents in hexadecimal form.

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.

If for data a name is specified which is 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.

```plaintext
data-OPERAND
{ [filename] [dataname] [statement-name] [source-reference] [keyword] [compl-memref] [filename] [dataname] [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**

Qualifications need only be specified if an address operand does not apply to the current AID work area or if an address is to be referenced which is not in the current compilation unit or the current program.

\[ E = \{ \text{VM} \mid \text{Dn} \} \]

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

\[ S = \text{srcname} \]

Specified only if data is not contained in the current compilation unit.

\[ \text{PROC} = \text{program-id} \]

Specified only if a file name, data name or statement name is addressed which is not contained in the current program (see chapter 4) or which is not unique in the current compilation unit. It is also required for a global data name that is locally hidden.

If `srcname` in the S qualification is the same as the `program-id`, only the PROG qualification need be written.

**filename**

is the name of a file from a file definition in the FILE-SECTION of the DATA DIVISION.

AID outputs the following information:

the file status and, if the file is open, the contents of the data record area and any record key.

**dataname**

specifies the name of a data item, the name of a COBOL special register or a figurative constant as defined in the source program.

If `dataname` is not unique within a program, it can be identified.

If `dataname` is the name of a table element, it can be indexed or subscripted in the same way as in a COBOL statement (see section 4.2 on `dataname`).

\[ \text{dataname} \{ \text{identifier} \}[\ldots][\{\text{index}[\ldots]\}] \]

**identifier**

`dataname` is assigned to a particular group item with IN or OF. `dataname` must have as many identifiers as are required to designate it unambiguously.

If it is not identified, AID only outputs data for `dataname` if a data definition is...
provided for it at level 01 or 77. If this is not the case, an error message is issued.

index
is written as in a COBOL statement, except that in the AID command multiple indexes must be separated by commas.

index can be specified as follows:

\[
\text{index-name, dataname, TALLY, arithmetic-expression}
\]

COBOL special registers
- LINAGE-COUNTER
- RETURN-CODE
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY

Figurative constants
- The address selector cannot be used with figurative constants.
- ZERO
- SPACE
- HIGH-VALUE
- LOW-VALUE
- QUOTE
- literal
- symbolic character

statement-name
designates the address of the first statement in a section or paragraph in the PROCEDURE DIVISION.

\[
\{\text{L'\text{section'}}\}
\{\text{L'\text{paragraph'}} \text{ [IN L'\text{section'}}]\}
\]

If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: L'\text{paragraph'} IN L'\text{section'}

With the subsequent pointer operator (->) AID outputs 4 bytes of the program code generated for the first statement in the section or paragraph.
source-reference
designates the address of the first instruction generated for a statement in the
PROCEDURE DIVISION and must be specified in one of the following formats:

S’n’ for lines with paragraph or section names in which no COBOL verb
occurs. This specification is not possible for programs which have been
compiled with STMT-REFERENCE=COLUMN1-TO-6.

S’nverb[m]’ | S’xverb[m]’
for lines containing a COBOL verb.

With the subsequent pointer operator (->) AID outputs 4 bytes of the program
code generated for the statement.

keyword
Here you may specify all the keywords for program registers, AID registers,
system tables and the one for the execution counter or the symbolic localization
information (see AID Core Manual, chapter 10 [1]).

**keyword** can only be preceded by a base qualification.

| %n | General register, $0 \leq n \leq 15 |
| %D|E | Floating-point register, $n = 0,2,4,6 |
| %Q | Floating-point register, $n = 0,4 |
| %G | AID general register, $0 \leq n \leq 15 |
| %DG | 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 |
| %PM | Program mask |
| %AMODE | Addressing mode of the test object |
| %PCB | Process control block |
| %PCBLST | List of all process control blocks |
| %SORTEDMAP | List of all CSECTs and COMMONs of the user program |
(sorted by name and address) |
long names are truncated |
| %MAP [CTX-context] | List of all CSECTs and COMMONs of all contexts of the user program or of the context designated by the context qualification; the names are output in full, not abbreviated (for further operands see AID Core Manual, section 10.6 [1]) |
| %LINK | Name of the segment dynamically loaded last |
| %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) |
| %•subcmdname | Execution counter |
| %• | Execution counter of the currently active subcommand |
compl-memref

The following operations may occur in a compl-memref (see AID Core Manual, chapter 7.5 [1]):
- byte offset (*)
- indirect addressing (->)
- type modification (%T(dataname), %X, %C, %P, %D, %F, %A, %S, %SX)
- length modification (%L(...), %L=(expression), %Ln)

If a statement name or a source reference is to be used as a memory reference, it must be followed by a pointer operator (->). Without the pointer operator the statement name and source reference can be used anywhere where hexadecimal numbers can be written. Using the type modification, data may be edited in another form since the output type changes with the storage type.

With the length modification you can define the output length yourself, e.g. if you wish to output only parts of a data item or display a data item using the length of another data item. It is only permitted to exceed the implicit area limits of an address with type or length modification after first using `%@(dataname)-> to switch to machine code level, on which the area comprises the virtual memory occupied by the loaded program.

%@(...)%

With the address selector you can output the start address of a data entry, a data item, a special register or a complex memory reference (see the AID Core Manual, chapter 7.5.6 [1]). The address selector cannot be used for constants. However, the statement names, the source references and the figurative constants among these can be specified by a subsequent pointer.

Examples

%D %@('LEAD'->)
%D %@('97MOV'->)

%L(...)%

With the length selector you can output the length of a data entry, a data item or a special register (see the AID Core Manual, chapter 7.5.6 [1]).

Example: %DISPLAY %L(ITEM1)
The length of ITEM1 is output.

%L=(expression)%

With the length function you can have a value calculated.

expression is formed from memory references and arithmetic operators (see AID Core Manual, chapter 7.5.4 [1]).

Example: %DISPLAY %L=(ITEM1)
If ITEM1 is an integer (type %F), the contents of ITEM1 will be output. Otherwise AID issues an error message.
AID literal
All AID literals described in the AID Core Manual, chapter 9, 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
- \[\{\pm\}\]n Integer
- #'f...f' Hexadecimal number
- \[\{\pm\}\]n.m Fixed-point number
- \[\{\pm\}\]mantissaE\[\{\pm\}\]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.
  \(1 \leq n \leq 255\). 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 \(T = \text{MAX}\).

\[
\begin{bmatrix}
T \\
H \\
Fn \\
P
\end{bmatrix} = \begin{bmatrix}
\text{MAX} \\
\text{MIN}
\end{bmatrix}
\]

medium-a-quantity is described in full detail in the AID Core Manual, chapter 8.

- \(T\) Terminal output
- \(H\) Hardcopy output (includes terminal output and cannot be specified together with \(T\))
- \(Fn\) File output
- \(P\) Output to SYSLST

\(\text{MAX}\) Output with additional information
\(\text{MIN}\) Output without additional information

Example
Specification of several medium-a-quantity operands

\%DISPLAY DATARECORD F1=MAX, H=MIN
Examples

1. \%DISPLAY E=D1.PROG=EXAMPLE.FCOMP3S,‘CONTENTS OF DUMP’
   
   Here the contents of a dump are evaluated.
   
   ```
   ** D1: DUMP.EXAMPLE ************************************************************
   "FCOMP3S  = +999456989
   CONTENTS OF DUMP
   
   **
   ```

2. \%DISPLAY \%L=(S’13ADD’-S’12MOV’)
   
   AID outputs the length of the machine code sequence generated for statement 12MOV.
   
   +52

3. \%BASE
   
   \%DISPLAY L’PROCESSING’
   
   \%BASE switches back to the AID standard work area. AID then outputs the address of the first instruction in the paragraph PROCESSING as a hexadecimal number.
   
   ```
   ** ITN: #'00010053' *** TSN: 6567 **********************************************
   SRC_REF: 45INIT SOURCE: MOBS PROC: MOBS ****************************
   PROCESSING = 00000A84
   
   **
   ```

4. \%DISPLAY L’PROCESSING’->
   
   AID outputs 4 bytes of the machine code contained at the address of the paragraph PROCESSING. The pointer operator switches to the machine code level, which causes AID to display an additional header.
   
   ```
   CURRENT PC: 00000A04 CSECTION: MOBS ****************************
   V’00000A84’ = MOBS + #'00000A84’
   00000A84 (00000A84) 18001800 ....
   
   **
   ```

5. \%DISPLAY \%HLLOC(L’OUT1’ IN L’PUTOUT’->)
   
   AID outputs symbolic localization information for paragraph OUT1 in section PUTOUT.
   
   ```
   V’00000C2C’ = SMOD : EXAMPLE
   PROC : EXAMPLE
   SECTION : PUTOUT
   PARAGRAPH : OUT1
   SRC-REF : 77
   LABEL : OUT1
   ```
6. %DISPLAY %LOC(L'OUT1' IN L'PUTOUT' =>)

AID outputs localization information on machine code level for paragraph OUT1 in section PUTOUT.

V'00000C2C' = PROG : EXAMPLE
LMOD : %ROOT
SMOD : EXAMPLE
OMOD : EXAMPLE
CSECT : EXAMPLE (00000000) + 00000C2C

7.

/LOAD M1BS,SYM=ALL
% BLS0500 PROGRAM 'M1BS', VERSION '91-09-04' LOADED.

Unpacked numbers
12345
1234N

Packed numbers
12345
1234N

% IDA0N51 PROGRAM INTERRUPT AT LOCATION '008702 (M1BS), (CDUMP), EC=68
% IDA0N45 DUMP DESIRED? REPLY (Y = USER/AREA DUMP; Y,SYSTEM = SYSTEM ,N=NO)?
% EXC0077 PROGRAM IS STILL LOADED AND IN 'HOLD-PROGRAM' MODE. PROGRAM RUN MAY BE
CONTINUED WITH /RESUME-PROGRAM

Your program has encountered an error. Now you want to know which statement
caused this error. To find this out, enter %DISPLAY %HLLOC for the address at
which the program was interrupted by the error:

/\DISPLAY %HLLOC(V'8702')
** ITN: '#000004D' *** TSN: 4192 ***********************************************
CURRENT PC: 00008702 CSECT: UPRO ****************************************
V'00008702' = SMOD : UPRO
PROC : UPRO
SRC-REF : 33COM
\%D %LOC(V'8702')
V'00008702' = PROG : M1BS
LMOD : %ROOT
SMOD : UPRO
OMOD : UPRO
CSECT : UPRO (000008230) + 000004D2

8. %DISPLAY ALPHA-CHAR(I)

Let ALPHA-CHAR be defined as in example 9 and index I contain the value 5. The
5th element in the table will be output:

ALPHA-CHAR(5) = |E|
9. **%DISPLAY ALPHA-CHAR**

The ALPHA-CHAR element is contained in a table 26 times and defined in the DATA DIVISION as follows:

```
01 A-Z-TAB1.
  02 ALPHA-CHAR PIC X OCCURS 26 INDEXED BY I.
```

As no index was specified in %DISPLAY, AID outputs all the elements with this name:

```
** ITN: #'00010053' *** TSN: 6567 **********************************************
SRC_REF: 45INIT SOURCE: EXAMPLE PROC: EXAMPLE **************************
ALPHA-CHAR( 1: 26)
( 1) A ( 2) B ( 3) C ( 4) D ( 5) E ( 6) F ( 7) G
( 8) H ( 9) I (10) J (11) K (12) L (13) M (14) N
(22) V (23) W (24) X (25) Y (26) Z
```

10. **Comparison of AID and COBOL output of data items:**

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NUM.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
  TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 UNPKD1 PIC 99999.
  01 UNPKD2 PIC 999999 VALUE ZERO.
  01 PCKD1 PIC 999999999 VALUE ZERO.
  01 PCKD2 PIC 9999999999 VALUE ZERO.
  01 FLOAT1 PIC +999.99E-99.
  01 FLOAT2 COMP-1.
  01 FLOAT3 COMP-1 VALUE 12.
  01 FLOAT4 COMP-2.
  01 FLOAT5 COMP-2 VALUE +123456789.1234567E+10.
  01 BIN1 PIC 99999 BINARY.
  01 BIN2 PIC 99999 BINARY VALUE ZERO.
PROCEDURE DIVISION.
UNPKD.
  DISPLAY *Unpacked numbers* UPON T.
  MOVE 12345 TO UNPKD1.
  DISPLAY UNPKD1 UPON T.
  MOVE -123.45 TO UNPKD2.
  DISPLAY UNPKD2 UPON T.
PCKD.
  DISPLAY *Packed numbers* UPON T.
  MOVE 12345 TO PCKD1.
  DISPLAY PCKD1 UPON T.
  MOVE UNPKD2 TO PCKD2.
  DISPLAY PCKD2 UPON T.
FLOAT.
  DISPLAY *Floating-point numbers* UPON T.
  MOVE 12345 TO FLOAT1.
  MOVE 12345 TO FLOAT2.
  DISPLAY FLOAT1 UPON T.
  DISPLAY FLOAT2 UPON T.
  MOVE UNPKD2 TO FLOAT3.
  DISPLAY FLOAT3 UPON T.
  MOVE UNPKD2 TO FLOAT4.
  DISPLAY FLOAT4 UPON T.
  DISPLAY FLOAT5 UPON T.
```
BIN.
    DISPLAY "Binary numbers" UPON T.
    MOVE 12345 TO BIN1.
    DISPLAY BIN1 UPON T.
    MOVE UNPKD2 TO BIN2.
    DISPLAY BIN2 UPON T.
END.
    STOP RUN.

COBOL output

<table>
<thead>
<tr>
<th>Unpacked numbers</th>
<th>12345</th>
<th>1234N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packed numbers</td>
<td>12345</td>
<td>1234N</td>
</tr>
<tr>
<td>Floating-point numbers</td>
<td>+123.45E 02</td>
<td>+.123450E+05</td>
</tr>
<tr>
<td></td>
<td>+.120000E+02</td>
<td>-.12345000000000E+03</td>
</tr>
<tr>
<td></td>
<td>+.12345789123457E+19</td>
<td>Binary numbers</td>
</tr>
<tr>
<td></td>
<td>12345</td>
<td>012L</td>
</tr>
</tbody>
</table>

AID output

<table>
<thead>
<tr>
<th>%D UNPKD1, UNPKD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC_REF: 44DIS SOURCE: UPRONUM PROC: UPRONUM ***************************</td>
</tr>
<tr>
<td>UNPKD1 = 12345</td>
</tr>
<tr>
<td>UNPKD2 = -123.45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%D PCKD1, PCKD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCKD1 = 12345</td>
</tr>
<tr>
<td>PCKD2 = -123.45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%D FLOAT1, FLOAT2, FLOAT3, FLOAT4, FLOAT5</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT1 = +.12345 E+005</td>
</tr>
<tr>
<td>FLOAT2 = +.1234500 E+005</td>
</tr>
<tr>
<td>FLOAT3 = +.1200000 E+002</td>
</tr>
<tr>
<td>FLOAT4 = -.1234999999999999 E+003</td>
</tr>
<tr>
<td>FLOAT5 = +.1234567891234566 E+19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%D BIN1, BIN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN1 = 12345</td>
</tr>
<tr>
<td>BIN2 = -123</td>
</tr>
</tbody>
</table>
%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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%DUMPFILE</td>
<td>[link [=file]]</td>
</tr>
<tr>
<td>%DF</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>%F[IND]</td>
<td>[ [ALL] search-criterion [IN find-area] [alignment] ]</td>
</tr>
</tbody>
</table>

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.

In a %FIND command with a separate **search-criterion** and without any other operands, AID inserts the corresponding default value for an operand without a current value. In this case, therefore, no operands are taken over from a previous %FIND command.

In the event of a hit, output is to a maximum length of 12 bytes, from the hit to the end of **find-area** on the terminal (SYSOUT) in dump format (hexadecimal and character representation). In addition to the hit itself, its address and (insofar as possible) the name of the compilation unit in which the hit was found, and the relative address of the hit with respect to the beginning of the compilation unit, are output. 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 %0G and the continuation address of the last hit is stored in %1G. If the **search-criterion** has not been found, AID registers %0G and %1G remain 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...xC | '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'.

```
{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 item or part of the PROCEDURE DIVISION 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, chapter 10.5), i.e. the class 6 memory for the currently set base qualification is searched (see %BASE).
• 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

Qualifications need only be specified if an address operand does not apply to the current AID work area or if an address is to be referenced which is not in the current compilation unit or the current program.

E={VM | Dn}
Need only be specified if the current base qualification is not to apply for find-area (see also %BASE command).

S=srcname
Need only be specified if find-area is not within the current compilation unit (see section 4).

PROC=program-id
Need only be specified if find-area is not within the current program (see chapter 4) or if it is defined with a dataname or statementname which is not unique in the compilation unit.

If srcname in the S qualification is the same as program-id, only the PROG qualification need be written.

Only the base qualification or the CTX qualification can be placed before the C qualifications listed below. The C qualification takes the user away from the symbolic level. No symbolic operands can be written directly afterwards (see section on "Symbolic memory references"), only a compl-memref.

C=segmentname
Without a length modification the entire segment is specified as find-area.

C=sharename
Without a length modification the entire object module is specified as find-area.
dataname

is the name of a data item defined in the source program or the name of a COBOL special register.
If dataname is not unique within a program, it can be marked.
If dataname is the name of an element in a table, it can be indexed or subscripted in the same way as in a COBOL statement (see section 4.2).

datename [identifier][...] [(index[,...])] 

identifier

IN or OF can be used to assign dataname to a certain group item. dataname must have as many identifiers as are required to designate it unambiguously.
If it is not identified, AID only processes dataname if a data definition is provided for it at level 01 or 77. If this is not the case, an error message is issued.

index

This is written in the same way as in a COBOL except that multiple indexes in the AID command must be separated by commas. index can be specified as follows:

\[ \text{index-name} \]
\[ \text{dataname} \]
\[ \text{TALLY} \]
\[ \text{arithmetic-expression} \]

COBOL special registers

LINAGE-COUNTER
RETURN-CODE
SORT-CORE-SIZE
SORT-FILE-SIZE
SORT-MODE-SIZE
SORT-RETURN
TALLY

\{ \text{statement-name} \}
\{ \text{source-reference} \} ->

designates 4 bytes of the program code from the address contained in the address constant. If a different number of bytes is to be searched, you must specify a corresponding length modification.

statement-name

defines the address of the first instruction in a section or paragraph in the PROCEDURE DIVISION.

\{ L’section’ \}
\{ L’paragraph’ [IN L’section’] \}

If a paragraph name is not unambiguous within a program, it must be identified
by the section name of the section in which it was defined: L’paragraph’ IN L’section’

source-reference
designates the address of the first instruction generated for a statement in the PROCEDURE DIVISION and must be specified in one of the following formats:

`S’n’` for lines with paragraph or section names in which no COBOL verb occurs. This specification is not possible for programs which have been compiled with `STMT-REFERENCE=COLUMN1-TO-6`.

`S’nverb[m]’ | S’xverb[m]’` for lines containing a COBOL verb.

compl-memref
The following operations may occur in compl-memref (see also AID Core Manual, chapter 7.5 [1]):
- byte offset (`*`)
- indirect addressing (`->`)
- type modification (`%A, %S, %SX`)
- length modification (`%L(...), %L=(expression), %Ln`)
- address selection (`%@(...)`)

If compl-memref begins with a statement name or source reference, it must be followed by a pointer operator (`->`). In this case statement-name must be specified with L’...’. Without the pointer operator the statement name and source reference can be used anywhere where hexadecimal numbers can be written. compl-memref designates an area of 4 bytes starting from the calculated address.

If a different number of bytes is to be searched, a corresponding length modification must be added. When modifying the length of data items you must pay attention to area boundaries or switch to machine code level using `%@(dataname)->`. 
alignment defines that the search for search-criterion is to be effected at certain aligned addresses only.

alignment-OPERAND = \begin{align*}
&\text{byte boundary (default)} \\
&\text{halfword boundary} \\
&\text{word boundary} \\
&\text{doubleword boundary}
\end{align*}

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’12MOV’->S’13ADD’-S’12MOV’ ALIGN=2
   The hexadecimal literal X’D2’ is searched for at a halfword boundary in the machine code generated for statement 12MOV.

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.

- With info-target 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.

- With medium-a-quantity 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%HELP</td>
<td>[info-target] [medium-a-quantity] [...]</td>
</tr>
</tbody>
</table>

%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.

The AID messages have the message code format AID0n, while the AIDSYS messages have the format IDA0n. Both are queried using /HELP. In addition, in the current AID version the AID messages can be queried with In using the AID %HELP command, as before.

%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.

```plaintext
info-target
```

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

AID responds to a %HELP command containing an invalid info-target operand by issuing an error message. This is followed by the same overview as for a %HELP command without info-target. This overview can also be requested via the %?, %H%? or %H%? entries.
The AID command names may be abbreviated as shown above. In designates the old message code of a message for which the meaning and possible responses are to be output. 

\( n \) is a 3-digit message number. Querying of messages using the AID %HELP command will no longer be supported in future AID versions because the same information can be obtained with the SDF command HELP-MESSAGE-INFORMATION.

medium-a-quantity defines the media via which information on the info-target is to be output.

If this operand is omitted and no declaration has been made using the %OUT command, AID works with the default value \( T=\text{MAX} \). The specification \{MIN | MAX\} has no effect with %HELP, but the syntax requires one of these two specifications.

medium-a-quantity is described in detail in the AID Core Manual, chapter 8.

- **T** Terminal output
- **H** Hardcopy output (includes terminal output and cannot be specified together with \( T \))
- **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.

- With test-point you may define the address of a command in the program prior to whose execution AID interrupts the program run and to process subcmd.
- With subcmd 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.
- With control 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%INSERT</td>
<td>test-point [subcmd] control</td>
</tr>
</tbody>
</table>

A test-point is deleted in the following cases:

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

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

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

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

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

%INSERT does not alter the program state.
must be the address of an executable machine instruction generated for a COBOL statement. test-point is immediately entered by targeted overwriting of the memory position addressed and must therefore be loaded in virtual memory at the time the %INSERT command is input. Since, by entering test-point, the program code is modified, a test point which has been incorrectly set may lead to errors in program execution (e.g. data/addressing errors).

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

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

[*][qua*][...]

- 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

Qualifications must be specified if an address operand is not valid for the current AID work area, the current compilation unit or the current program, or if it is not unambiguous in some other way.

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).

S=srcname

Need only be specified if test-point is not to be contained within the current compilation unit.

PROC=program-id

Need only be specified if a statement name is not in the current program or if it is not unique in the current compilation unit (see chapter 4).

If srcname in the S qualification and program-id are the same, only the PROG
qualification need be written.

Only the base qualification or the CTX qualification can be placed before the C qualifications listed below. The C qualification takes the user away from the symbolic level. No symbolic operands can be written directly afterwards (see section on “Symbolic memory references”), only a compl-memref.

C=segmentname
   With this specification you set test-point to the start address of the designated segment.

C=sharename
   With this specification you set test-point to the start address of the designated reusable program.

program-id
   This specification is possible after an explicit PROC/PROG qualification or if the current interrupt point is in the program that is identified by program-id. The effect is to set test-point to the first executable statement of the designated program.

statement-name
   designates the address of the first instruction in a section or paragraph in the
   PROCEDURE DIVISION.

   \{L’section’
   \{L’paragraph’ [IN L’section’] \}

   An alphanumeric section or paragraph name can be specified without L’…’ since this name cannot be confused with a data name in this command.

   If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: L’paragraph’ IN L’section’ In order to be able to set a test-point at a statement name generated with COLUMBUS, the program must be compiled with the SDF option:
   PREPARE-FOR-JUMPS=YES

source-reference
   designates the address of the first instruction generated for a statement in the
   PROCEDURE DIVISION and must be specified in one of the following formats
   (see chapter 4):

   S’n’   for lines with paragraph or section names in which no COBOL verb occurs. This specification is not possible for programs which have been compiled with STMT-REFERENCE=COLUMN1-TO-6.
S’verb[m]’ | S’xverb[m]’
for lines containing a COBOL verb.

compl-memref
The result of compl-memref must be the start address of an executable machine instruction. compl-memref may contain the following operations (see AID Core Manual, chapter 7.5 [1]):
- byte offset (*)
- indirect addressing (->)
- type modification (%A)
- length modification (%Ln)
- address selection (%@(...))

If compl-memref begins with a statement name or source reference, it must be followed by a pointer operator ( -> ). In this case statement-name must be specified with L’...’. Without the pointer operator the statement name and source reference can be used anywhere where hexadecimal numbers can be written.

Example:
%INSERT L’PUTOUT’ ->.4

test-point is set to the second instruction after the PUTOUT paragraph. The first instruction was 4 bytes long.

Type modification makes sense only if the contents of a data item can be used as an address or if you take the address from a register.

Example:
%1G.2 %AL2 ->
The last two bytes from AID register %1G are used as the address.

subcmd
A subcommand 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 6.

subcmd-OPERAND

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. The \texttt{subcmd} does not overwrite an existing subcommand for the same \texttt{test-point}, rather the new subcommand is prefixed to the existing one. The \texttt{subcmd} of an \%ON or \%INSERT may contain the commands \%CONTROL\textit{n}, \%INSERT, \%JUMP and \%ON. Nesting over a maximum of 5 levels is possible.

The commands in a \texttt{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 \texttt{subcmd} and start the program (\%CONTINUE, \%RESUME, \%TRACE) or halt it (\%STOP). They are thus only effective as the last command in a \texttt{subcmd}, since any subsequent commands in the \texttt{subcmd} would fail to be executed. Likewise, deletion of the current subcommand via \%REMOVE makes sense as the last command in \texttt{subcmd} only.

\begin{center}
\textbf{control}
\end{center}

specifies whether \texttt{test-point} is to be deleted after the \textit{n}-th pass and whether the program is to be halted with the purpose of inserting new commands. If no \textit{control} operand has been specified, AID assumes the defaults \textit{65535} (for \textit{n}) and \textit{K}.

\begin{tabular}{l}
\textbf{control-OPERAND} \hline \\
\hline \\
\textbf{ONLY} \textbf{n} \{ \begin{tabular}{c} \textbf{K} \\ \textbf{S} \\ \textbf{C} \end{tabular} \} \\
\hline \\
\end{tabular}

\textit{n} is a number with the value $1 \leq n \leq 65535$, specifying after how many \texttt{test-point} passes the further declarations for this \textit{control} operand are to go into effect.

\begin{itemize}
\item \textbf{K} test-point is not deleted (KEEP).
\hspace{1cm} Program execution is interrupted, and AID expects input of commands.
\item \textbf{S} test-point is deleted (STOP).
\hspace{1cm} Program execution is interrupted, and AID expects input of commands.
\item \textbf{C} test-point is deleted (CONTINUE).
\hspace{1cm} No interruption of the program.
\end{itemize}

This operand is described for the last time in this edition.
Examples

1. \%IN S’48MOV
   test-point is specified with a source reference and is set to the memory location of
   the instruction code generated for the MOVE in statement line 48.

2. \%IN ST3 <%DISPLAY PERSNR> ONLY 10 S
   test-point is designated by the paragraph name ST3. Whenever the program
   sequence arrives at the first statement in paragraph ST3, the \%DISPLAY command
   of the subcmd is executed. When test-point is reached for the 10th time, AID sets
   the program to STOP and deletes the test point, at which time you may enter new
   commands.

3. \%IN ST2 <%DISPLAY TEXTDAT, ‘ST2’>
\%IN ST3 <%DISPLAY ‘INSERT1’, TEXTDAT; \%IN PUTOUT \%D ‘INSERT2’, -
I, J, K, NUMB-TABLE; \%IN S’172’ <%D ‘INSERT3’, I, J; \%REMOVE PUTOUT>>>
With the first \%INSERT command, paragraph ST2 is set as the test-point. If, after
the end of command input, the program execution reaches ST2, the subcommand
is executed. It consists of a \%DISPLAY command (for data name TEXTDAT) and
the 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 paragraph ST3, the corresponding subcmd is
executed, i.e. the \%DISPLAY command for the literal ‘INSERT1’ and the variable
TEXTDAT is executed and the test-point PUTOUT is set.
The subcmd for test-point PUTOUT 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 PUTOUT.

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 PUTOUT, then the \%D
command ‘INSERT2’, I, J, K, NUMB-TABLE is executed. Furthermore, subcmd
contains a further \%INSERT command, whose test-point this time is specified with
source-reference 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 data
items I and J. By way of the second command in this subcmd, the \%REMOVE
PUTOUT command, test-point PUTOUT 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 PUTOUT:

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

4. %OUT %DISPLAY P=MAX
%IN S’73SET’ <%D ‘I GE 10’, I, CHAR(I), K, NR-C(I, K)> %IN S’73SET’ <(I LT 10): %D ‘I LT 10’, I, CHAR(I); %CONT>

First, all outputs of the %DISPLAY command are directed to SYSLST. The two subsequent %INSERTs create the following subcommand at *test-point* S’73SET’:

```
<(I LT 10): %D ‘I LT 10’, I, CHAR(I); %CONT; %D ‘I GE 10’, I, CHAR(I), - K, NR-C(I, K)>
```

Every time the program sequence reaches the statement with the name 73SET, a check is made whether index I contains a value < 10. If the condition is satisfied, AID writes the comment ‘I LT 10’ and the contents of I and CHAR(I) to SYSLST and, as a result of %CONTINUE, continues the program (with tracing, if the subcommand interrupted a %TRACE).

If the value of I is $\geq$ 10, AID writes the comment ‘I GE 10’ and, in addition to I and CHAR(I), also the values of index K and table element NR-C(I, K) to SYSLST and likewise continues the program. In this case, too, any active %TRACE is continued.
%JUMP

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

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%JUMP</td>
<td>continuation</td>
</tr>
</tbody>
</table>

%JUMP can only be used for programs which were compiled with COBOL85. For compilation purposes, you must specify the SDF option PREPARE-FOR-JUMPS=YES or the COMOPT statement SEPARATE-TESTPOINTS=YES.

The continuation address must be located in the same program as the current interrupt point, otherwise the command results in an error because essential initializations have not been carried out.

The user must ensure that the prerequisites (e.g. index or counter states, file status) for error-free execution of program as of continuation have been fulfilled. This is especially important if you use the %JUMP command to reach an address which comes logically before the interrupt point in the course of program execution.

You may not enter the %JUMP command in the following cases:
- immediately after the LOAD-PROGRAM command
- if the program has been interrupted by the system, e.g. because a file to be opened has not yet been assigned
- if the K2 key has been used to interrupt the program.

The %JUMP command does not alter the program state.
continuation defines the position at which the program is to be continued. continuation must be the address of an executable statement within the current program. If the %JUMP command is part of a subcommand, continuation must designate a statement in the program in which the current interrupt point for test-point or event has occurred.

continuation-OPERAND

statement-name

source-reference

statement-name designates the address of the first instruction in a section or paragraph in the PROCEDURE DIVISION.

L’section’
L’paragraph’ [IN L’section’]

An alphanumeric section or paragraph name can be specified without L’...’ since this name cannot be confused with a data name in this command.

If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: L’paragraph’ IN L’section’

source-reference

continuation can only be the address of the first instruction in a section or paragraph in the PROCEDURE DIVISION and can thus only be specified with the following source reference:

S’n’ for lines with paragraph or section names if they do not include a COBOL verb. This means that no %JUMP source-reference is possible for a program that has been compiled with STM-REFERENCE=COLUMN1-T0-6.

Example

%JUMP S’67’
%JUMP PUTOUT

Both commands refer to the example in section 6.1. Statement line 67 contains only the paragraph name PUTOUT. Thus the same continuation address is declared with both commands, namely the first executable statement in the PUTOUT paragraph.
With the %MOVE command you transfer memory contents or AID literals to memory positions within the program which has been loaded. Transfer is effected left-justified without checking and without matching the storage type of the sender to the receiver. The %SET command is required for transfer appropriate to type, as in the COBOL MOVE statement.

- With `sender` you designate a data item, an address, an execution counter, an AID register, a COBOL special register, a figurative COBOL constant or an AID literal. `sender` can be located in virtual memory of the loaded program or in a dump file.

- With `receiver` you designate a data item, an execution counter, an AID register or a COBOL special register which is to be overwritten. `receiver` can only be located in virtual memory of the loaded program.

- With `REP` 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 the global setting (see %AID command) but affects only the current %MOVE command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%MOVE</td>
<td>sender INTO receiver [REP]</td>
</tr>
</tbody>
</table>

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.

Input of the command immediately following loading is not recommended as not all entries in the DATA DIVISION will have been initialized (e.g. record definitions and special registers).

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.
%MOVE

For *sender* or *receiver* you can specify a data item, a COBOL special register, an execution counter, a register or a complex memory reference. Statement names, source references, figurative constants, addresses and lengths of data items 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. If program areas are transferred or overwritten with instruction code, the results may be undesirable if addresses are affected which belong to a *control-area* or *trace-area* or for which a test point has been set using %INSERT (see AID Core Manual, section 12.2 [1]).

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.

```plaintext
sender-OPERAND  INTO  receiver-OPERAND
```

• 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

Qualifications need to be specified if an address operand does not apply to the current AID work area or if the intention is to reference an address that is not within the current compilation unit or the current program.
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.

S=srcname
is to be specified if sender or receiver is not contained in the current compilation unit.

PROC=program-id
is to be specified only if you address a file name, data name or statement name that is not in the current program or is not unique in the current compilation unit (see chapter 4). It is also necessary for a global data name that is locally hidden.
If srcname in the S qualification is the same as program-id, only the PROG qualification need be written.

Only the base qualification or the CTX qualification can be placed before the C qualifications listed below. The C qualification takes the user away from the symbolic level. No symbolic operands can be written directly afterwards (see section on “Symbolic memory references”), only a compl-memref.

C=segmentname
Without a length modification, specify the entire segment as the sender or receiver. If the segment is more than 3900 bytes in length, it can only be transferred by using several %MOVEs.

C=sharename
Without a length modification, specify the entire object module as the sender or receiver. If it is more than 3900 bytes in length, it can only be transferred by using several %MOVEs.

dataname
is the name of a data item defined in the source program, i.e. both individual data elements and group items and tables and their elements, or the name of a COBOL special register. Figurative constants can only be used as sender.
If dataname is not unique within a program, it can be marked.
If dataname is the name of a table element, it can be indexed or subscripted in the same way as in a COBOL statement.
dataname [identifier][...][index[,...]]

identifier
If dataname is not unambiguous within a program, it can be identified by being assigned to a particular group item with IN or OF. dataname must be assigned as many identifiers as are required to designate it unambiguously. If it is not identified, AID processes dataname if a data definition is provided for it at level 01 or 77. If this is not the case, an error message is issued.

index
is written in the same way as in a COBOL statement, except that indexes must be separated by a comma. If you specify the name of a table element without an index, this means that the entire table will be transferred (in the case of _sender_). If you specify a table element without an index in the case of _receiver_, the table will be overwritten beginning at the start address and using the length of _sender_, without taking into account the subdivision into table elements.

index may be specified as follows:

```

```

COBOL special registers
LINAGE-COUNTER
RETURN-CODE
SORT-CORE-SIZE
SORT-FILE-SIZE
SORT-MODE-SIZE
SORT-RETURN
TALLY

Figurative constants
can only be specified as _sender_; the address selector cannot be used on them. The figurative constants HIGH-VALUE and LOW-VALUE always represent the value assigned to them by default or in the declarations made with the PROGRAM COLLATING SEQUENCE clause. In contrast to the COBOL MOVE statement, only one character is transferred in the AID command %MOVE when a figurative constant is used.
ZERO
SPACE
HIGH-VALUE
LOW-VALUE
QUOTE
literal
symbolic character
statement-name
designates the address of the first instruction in a section or paragraph in the
PROCEDURE DIVISION.

{L’section’
L’paragraph’ [IN L’section’]}

If a paragraph name is not unambiguous within a program, it must be identified
by the section name of the section in which it was defined: L’paragraph’ IN
L’section’

Statement names are address constants and can only be specified for sender. The
address thus designated is then transferred.

With the subsequent pointer operator (statement-name ->) you designate 4 bytes
of the program code generated for the first statement in the section or paragraph.

For 2-byte or 6-byte instructions you must specify a corresponding length
modification. statement-name -> can be used both as sender and receiver. See
examples 5 and 6.

source-reference
designates the address of the first instruction generated for a statement in the
PROCEDURE DIVISION and must be specified in one of the following formats:

S’n’ for lines with paragraph or section names in which no COBOL verb
occurs. This specification is not possible for programs which have been
compiled with STMT-REFERENCE=COLUMN1-TO-6.

S’nverb[m]’ | S’xverb[m]’
for lines containing a COBOL verb.

Source references are address constants and can only be specified for sender.
The address thus designated is then transferred.

With the subsequent pointer operator (source-reference ->) you designate 4 bytes
of the program code generated for the statement. For 2-byte or 6-byte instructions
you must specify a corresponding length modification. source-reference -> can be
used both as sender and receiver. See examples 5 and 6.

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

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

may contain the following operations (see AID Core Manual, chapter 7.5 [1]):

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

If compl-memref begins with a statement name or source reference, it must be followed by a pointer operator ( -> ). In this case statement-name must be specified with L'...'. Without the pointer operator the statement name and source reference can be used anywhere where hexadecimal numbers can be written. 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.

Despite this, the area boundaries of the start address (for example CSECT, dataname, keyword etc.) remain in effect. They must not be exceeded as the result of byte offset or length modification, otherwise AID issues an error message. Only by combining the address selection (%@) with the pointer operator (->) can you switch to machine code level, on which the area comprises the area of virtual memory occupied by the loaded program.

Example: %MOVE CITEM.3%L5 INTO CITEM1
This command is rejected by AID on account of a violation of the CITEM area. The variables CITEM and CITEM1 each occupy 5 bytes. The last 2 bytes of CITEM as well as the 3 following bytes are to be transferred to CITEM1. The command should read: %MOVE %@(CITEM)->.3%L5 INTO CITEM1

%@(...)  
With the address selector you can use the address of a data entry, a data item, a special register or a complex memory reference as sender. The address selector produces an address constant as a result (see AID Core Manual, chapter 7.5.6 [1]). The address selector cannot be used for constants, which also include statement names, source references and figurative constants.

%L(...)  
With the length selector you can use the length of a data entry, a data item or a special register as sender. The length selector produces an integer as a result
(see AID Core Manual, chapter 7.5.6 [1]).
Example: `%MOVE %L(ITEM1) INTO %0G
The length of ITEM1 will be transferred.

%L=(expression)
The length function enables you as sender to calculate a value. expression is
formed from the contents of memory references, constants, integers and
arithmetic operators. Only memory reference contents which are integers (type
%F or %A) are permitted. The length function produces an integer as a result (see
AID Core Manual, chapter 6).
Example: `%MOVE %L=(ITEM1) INTO %0G
The content of ITEM1 is transferred provided it is an integer (type %F), otherwise
AID issues an error message.

AID literal
The following AID literals (see AID Core Manual, chapter 9) 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'` Hexadecimal number

REP
Specifies whether AID is to generate a REP record after a modification has been
performed. With REP you temporarily deactivate the global setting (see %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 correction statements (REPs) in SDF format are created for the update
caused by the current %MOVE command. If the object structure list is not
available, no correction statements are generated and AID will output an error
message.
Also, if receiver is not located completely within one CSECT, or if sender is more
than 3900 bytes in length, AID will output an error message and not write a REP
record. To obtain REP records despite this, the user must distribute transfer
operations over several %MOVE commands.
AID stores the REPs with the requisite LMS correction statements in a file with the link name F6. The MODIFY ELEMENT statement must then also be inserted for the LMS run. Ensure, therefore, that no other output is 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 items and tables are defined in a COBOL program:

```cobol
  01  NUMB-TAB-1.
    02  QNTY-1 PIC 999 OCCURS 10 INDEXED BY I.
  01  NUMB-TAB-2.
    02  QNTY-2 PIC S9(6) OCCURS 50 INDEXED BY J.
  01  FIXDPOINT-TAB.
    02  FIXD-QNTY PIC S999V99 OCCURS 26.
  01  CHAR PIC X(4).
  01  INTG-QNTY PIC S9(7) BINARY.
```

1. **%MOVE QNTY-1 INTO QNTY-2**
   No index has been specified for the two table elements: AID therefore transfers the entire table NUMB-TAB-1 to NUMB-TAB-2 in hexadecimal format and left-justified, without taking into account any subdivision into table elements.

2. **%MOVE 20 INTO INTG-QNTY**
   AID writes a word containing the value 20 (X'00000014') to the data item INTG-QNTY, which also occupies 4 bytes in the COBOL program.

3. **%MOVE 20 INTO FIXD-QNTY(5)**
   N.B.: As in example 2, a word with the contents X'00000014' is written to FIXD-QNTY(5), which of course makes no sense when a table element of the fixed-point number type is involved. To transfer value 20 to FIXD-QNTY(5), you will have to enter a %SET command (see %SET), which performs conversion prior to the transfer.

4. **%MOVE X'58F0C160' INTO CHAR REP=YES**
   The contents of the data item CHAR are overwritten with the hexadecimal literal X'58F0C160'. A REP record is created for the correction and is stored in the file AID.OUTFILE.F6 or the file assigned to link name F6.
%ON

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

- With write-event you define a write access event, accessing a memory area.
  Whenever the program writes to the specified memory area, AID is to interrupt the program and process the subcmd.

- With event you define one of the other events (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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
</table>
| %ON     | { write-event  
|         | event   }  [subcmd] |

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

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

Once an event is entered it applies until it is deleted with %REMOVE or until the end of the program.

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

The %ON command does not alter the program state.

write-event

The %WRITE keyword activates write monitoring. It is followed by the memory area to be monitored, in parentheses. If the program changes a byte within the specified area, the program is interrupted and the subcmd is executed. The interrupt is effected after the instruction that caused the change at the memory location; it may also occur in a runtime routine.
Only one write-event can be defined at any one time. The entry of a new write-event overwrites an existing one. Other events can, however, be registered at the same time. If an event arrives at the same time as a write-event, AID processes the subcommand associated with write-event first.

The write-event can be deleted with %REMOVE %WRITE without specifying the memory reference.

The following interaction occurs between %ON write-event and other AID commands:

- If a %CONTROLn or a %TRACE is registered with a criterion on the machine code level, the entry of %ON write-event is rejected with an error message.
- If a machine instruction has been overwritten with the internal AID mark (X'0A81') by a %CONTROLn or %TRACE with a symbolic criterion, AID does not notice the write access by this instruction.
- Also if a machine instruction has been overwritten with the internal AID mark by the test point declared with %INSERT, AID does not notice the write access by the instruction.

In order to ensure unbroken write monitoring it is advisable to delete all %CONTROLn and %INSERT commands using %REMOVE and to delete any %TRACE commands that may still be entered by continuing with %RESUME after the %ON.

The memory area to be monitored can be any memory object, however it is addressed. It is defined by the start address and the implicit or explicit length. The maximum length of the area is 64 Kbytes, otherwise an error message is output.

If the address of the specified memory object is overloaded in the case of a program with an overlay structure, the corresponding area in the newly loaded program section.

write-event-OPERAND

%WRITE {{
  S=srcname[PROC=program-id•]}
  C=segmentname
  C=sharename
dataname
  statement-name->
  source-reference->
  compl-memref
}}
S=srcname
This need only be specified if write-event is not to be declared for the current compilation unit.

PROC=program-id
This need only be specified if you reference a data name or statement name that is not contained in the current program (see chapter 4) or is not unique in the current compilation unit.

If srcname in the S qualification and program-id are not the same, instead of both of these you should write only the PROG qualification.

The C qualifications listed below cannot be preceded by a qualification. The C qualification takes the user away from the symbolic level.

It is not permissible to write a symbolic operand directly afterwards (see section on "Symbolic memory references"), only a compl-memref.

C=segmentname
The memory area to be monitored comprises the segment designated with this specification.

C=sharename
The memory area to be monitored comprises the object module designated with this specification.

dataname
is the name of a data item as defined in the source program or of a COBOL special register. It can be identified and indexed in the same way as in the COBOL program (see section 4.2 dataname).

dataname is an alphanumeric string consisting of up to 30 characters.

dataname [identifier][...][(index[,...])]

identifier
If dataname is not unambiguous within a program, it is assigned to a particular group item with IN or OF. dataname must be assigned as many identifiers as are required to designate it unambiguously. If it is not identified, AID processes dataname if a data definition is provided for it at level 01 or 77. If this is not the case, an error message is issued.

index
If dataname is the name of an element in a table, it can be indexed and subscripted; the notation differs from COBOL only in that indexes must be separated by a comma.

If the name of a table element is specified without an index, the entire table is referenced.
*index* may be specified as follows:

\[
\begin{align*}
&\text{index-name} \\
&\text{dataname} \\
&\text{TALLY} \\
&\text{arithmetic-expression}
\end{align*}
\]

**COBOL special registers**

- LINAGE-COUNTER
- RETURN-CODE
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY

```
{statement-name} ->
```

source-reference

designates 4 bytes of the program code from the address contained in the address constant. If a different number of bytes is to be searched, you must specify a corresponding length modification.

**statement-name**

must be specified in one of the following formats:

\[
\begin{align*}
&L'\text{section}' \\
&L'\text{paragraph}' [IN L'\text{section}']
\end{align*}
\]

If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: `L'\text{paragraph}' IN L'\text{section}'

**source-reference**

must be specified in one of the following formats:

- `S'n'` for lines with paragraph or section names in which no COBOL verb occurs. This specification is not possible for programs which have been compiled with `STMT-REFERENCE=COLUMN1-TO-6`.
- `S'nverb[m]' | S'xverb[m]'` for lines containing a COBOL verb.

**compl-memref**

The following operations may occur in `compl-memref` (see also AID Core Manual, chapter 7.5.6 [1]):

- byte offset `(*)`
- indirect addressing `(->)`
- type modification `(%A, %S, %SX)`
- length modification `(%L(\ldots), %L=(\text{expression}), %Ln)`
address selection (%@(...))

cmpl-memref designates an area of 4 bytes starting from the calculated address. If a different number of bytes is to be searched, a corresponding length modification must be added. When modifying the length of data items you must pay attention to area boundaries or switch to machine code level using %@(dataname)->. If a cmpl-memref begins with a statement name or source reference, it must be followed by a pointer operator (->). In this case statement-name must be specified with L'...'. Without the pointer operator the statement name and source reference can be used anywhere where hexadecimal numbers can be written.

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. The response to an event code that has been processed with a STXIT routine cannot be a subcmd that has been defined for that event. If a subcommand is executed in relation to the %ANY event, at the subsequent termination of the program there is no query as to whether a dump is to be output. It may be necessary for the user to initiate output of the dump in the subcommand with /CREATE-DUMP.

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

If a write-event occurs at the same time as another event, the subcommand relating to the write-event is processed first. For selection of the SVC numbers and event codes see the "Executive Macros" manual [6].
event | subcmd is processed:
---|---
%ERRFLG (zzz) | after the occurrence of an error with event code zzz and before abortion of the program
%INSTCHK | after the occurrence of an addressing error, an impermissible supervisor call (SVC), 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 event code
%SVC(zzz) | before execution of the supervisor call (SVC) with the specified number
%SVC | before execution of any supervisor call (SVC)
%LPOV(x...x) | after loading of the segment with the specified name
%LPOV | after loading of any arbitrary segment (the name is output with %D %LINK)
%TERM(N[ORMAL]) | before normal termination of a program
%TERM(A[BNORMAL]) | before abnormal termination of a program, but after output of a memory dump
%TERM(D[UMP]) | before output of a memory dump with subsequent termination of the program
%TERM(S[TEP]) | before termination of the program with subsequent branching within procedures
%TERM | before termination of a program by any of the %TERM events described above
%ANY | before termination of a program with because of a program error or as a result of the %TERM events described above

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: 0 ≤ zzz ≤ 255

No check is made whether the specified number of the error weight or the SVC number is meaningful or permissible.
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 6.

subcmd-OPERAND

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 (MONA12) <%D 'MONA12 GELADEN'; %STOP>
   Each time the segment MONA12 is loaded AID outputs the message 'MONA12 GELADEN' and halts 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 event code does not exist, therefore the subcmd defined for this event will never be started.
Whenever data has been overwritten in TABLE in the main program HPROG, the symbolic localization information about the current program count and the contents of TABLE are output. The output is sent to the file that was assigned to the link name F1. The program then continues.

A search can then be run in this file to establish when TABLE was overwritten.
%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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%OUT</td>
<td>[target-cmd [medium-a-quantity], ...]</td>
</tr>
</tbody>
</table>

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 or /EXIT-JOB.

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[SEMBLE]
- %D[ISPLAY]
- %H[ELP]
- %S[DUMP]
- %T[RACE]
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 is described in detail in the AID Core Manual, chapter 7.

T Terminal output
H Hardcopy output (includes terminal output and cannot be specified together with T)
Fn File output
P Output to SYSLST

MAX Output with additional information
MIN Output without additional information

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 designate the output file.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%OUTFILE</td>
<td>[link [ = file]]</td>
</tr>
</tbody>
</table>

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

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

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

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

%OUTFILE does not alter the program state.

link

Designates one of the AID link names for output files and has the format Fn, where n is a number with a value 0 ≤ 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). Care should therefore be taken that no other outputs are allowed to be written to the file with link name F6.
%OUTFILE

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.

- With prequalification you define qualifications which you would like to incorporate in other commands by referencing them via a prefixed period.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Q[UALIFY]</td>
<td>[prequalification]</td>
</tr>
</tbody>
</table>

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 or /EXIT-JOB.

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 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

consists of a single qualification or a sequence of 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.
%QUALIFY

prequalification operand

\[ E = \{ VM \} \{ Dn \} \{ [•] S = srcname \} \{ [•] C = segmentname \} \{ C = sharename \} \]

\[ E = \{ VM \} \{ Dn \} \]

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

S=srcname

\textit{srcname} designates a compilation unit.

PROC=program-id

designates a program unit.

If \textit{srcname} in the S qualification and \textit{program-id} are the same, only the PROG qualification need be written.

C=segmentname

\textit{segmentname} is composed of the specification in the PROGRAM-ID paragraph and the segment number. Only the first 6 places of the PROGRAM-ID are used for generating the segment name.

C=sharename

\textit{sharename} is composed of the first 7 places of the PROGRAM-ID specification and the @ character.

Examples

1. %QUALIFY E=D1.PROG=SOR T %D .TAB(1)

Because of the \textit{prequalification}, the \%DISPLAY command has the same effect as the following \%DISPLAY command in full format:

%D E=D1.PROG=SOR.TAB(1)

2. %QUALIFY PROG=SUB %SET .A INTO .B

Because of the \textit{prequalification}, the \%SET command has the same effect as the following \%SET command in full format:

%SET PROG-SUB.A INTO PROG-SUB.B
3. %QUALIFY PROG=SUB
   %D .TAB(I)
   %D .L’OUT1’ IN L’PUTOUT’
   As in examples 1 and 2, the PROG qualification from the %QUALIFY command is written before the period in the two %DISPLAY commands. Thus in the first %DISPLAY command not only do you address table element TAB from the SUB program unit; you also search for index I in the SUB program unit. The same applies for the second %DISPLAY command for identifying the paragraph: the PROG qualification refers both to paragraph OUT1 and to the identifying section PUTOUT.
%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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%REMOVE</td>
<td>target</td>
</tr>
</tbody>
</table>

If a subcommand contains a %REMOVE which deletes this subcommand or the associated monitoring condition (test-point, event or criterion), any subsequent commands in subcmd 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.

%CONTROL | %CONTROLn
%INSERT  | test-point
%ON      | event | %WRITE
%[subcmdname]

%CONTROL
The declarations for all %CONTROLn commands entered are deleted.

%CONTROLn
The %CONTROLn command with the specified number (1 ≤ n ≤ 7) is deleted.
%IN[ERT]

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(x...x):

%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.

%WRITE

The write-event is deleted.

%•[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.

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

On the other hand, several subcommands may be chained at a test-point of the %INSERT command. With the aid of %REMOVE %•[subcmdname] you can delete an individual subcommand from the chain, while further subcommands for the same test-point will still continue to exist (see AID Core Manual, chapter 6.4 [1]). 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 <CALL: %D %.`
   `%REM %C1`
   `%REM %.CALL`

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

2. `%IN S'58SEA' <SUB1: %D CHAR, QNTY>
   %IN S'58SEA' <SUB2: %D RESLT; %REM %.`
   `%R`
   `...
   %REM S'58SEA'

   When the test point S'58SEA' is reached, RESLT is output. Then subcommand
   SUB2 is deleted, i.e. this subcommand is executed only once. Subsequently CHAR
   and QNTY are output, and the program continues. Whenever test point S'58SEA' is
   reached in the program sequence, subcommand SUB1 is executed. %REM S'58SEA'
   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'58SEA'.
%RESUME

With %RESUME you start the loaded program or continue it at the interrupt point or the point specified in the %JUMP command. The program executes without tracing.
%RESUME terminates all active %TRACE commands, whereas %CONTINUE has no effect on %TRACE.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%RESUME</td>
<td></td>
</tr>
</tbody>
</table>

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 items or file definitions, all data items or file definitions 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 program was interrupted to the sequence of CALL statements to the outermost program. Output is via SYSOUT, SYSLST or to a cataloged file.

- With *dump-area* you designate the data items or file definitions 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SD[UMP]</td>
<td>&quot;[[dump-area][,]... [medium-a-quantity][,]...]]&quot;</td>
</tr>
</tbody>
</table>

If compilation units for which there are no LSD records, not even in a PLAM library, are included in the hierarchy, the user must individually specify the compilation units for which LSD records have been loaded or for which they can be loaded from a PLAM library (see %SYMLIB command). *dump-area* can be repeated up to 7 times.

%SDUMP without operands outputs all data items of the current call hierarchy, if AID is able to access the associated LSD records. Data that is defined more than once is also output more than once.

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

Input of the command immediately following loading is not recommended as not all entries in the DATA DIVISION will have been initialized (e.g. record definitions and special registers) and an error message may occur.

If you enter a name for *dump-area* which is not contained in the LSD records, AID issues an error message. The other dump-areas of the same command will be processed normally.

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 or individual data items or file definitions. AID edits the data items 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* or *filename* is defined in multiple DATA DIVISIONs of the current call hierarchy it is also output repeatedly, unless *dump-area* has been restricted by a qualification or *dataname* is identified. If a data item or DATA DIVISION that is to be output contains redefinitions, these are also output.

All data items generated by the compiler are contained in an %SDUMP with which entire DATA DIVISIONs are output. The output also includes information on the files defined in the program, e.g. file status, contents of the I/O areas and the record definitions.

```

dump-area-OPERAND = - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[*][E=(VM | Dn)[•]} [*]} [S=srcname[•]} [PROC=program-id[•]} [*]} [filename]
```

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

E = (VM | Dn)

This need only be specified 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.

S=srcname

This need only be specified if *dump-area* is not to be within the current compilation unit, which must be within the call hierarchy.

PROC=program-id

This must be specified if *dump-area* is to apply only for the specified program. It must be within the call hierarchy. If *dump-area* ends with a PROC qualification,
AID will output all data of this program.
If `srcname` in the S qualification and `program-id` are the same, only the PROG qualification need be written.

filename

is the name of a file from a file definition in the FILE-SECTION of the DATA DIVISION. AID outputs the following information: the file status and, if the file is open, the contents of the record area plus any record key.

dataname

is the name of a data item as defined in the source program, the name of a COBOL special register or a figurative constant.
`dataname` is an alphanumeric string consisting of up to 30 characters.

`dataname [identifier] [...] [(index[,...])]`

identifier

If `dataname` is not unambiguous within a program, it can be identified by being assigned to a particular group item with IN or OF. `dataname` must be assigned as many identifiers as are required to designate it unambiguously.
If it is not identified, AID only outputs data for `dataname` if a data definition is provided for it at level 01 or 77. If this is not the case, an error message is issued.

index

If `dataname` is the name of an element in a table, it can be indexed and subscripted as in a COBOL statement. The notation differs from COBOL only in that multiple indexes must be separated by a comma. If you specify the name of a table element without an index, the entire table is output.
`index` can be specified as follows:

```
[:
  [index-name
    data-name
    TALLY
    [arithmetic expression]]
```

COBOL special registers

```
LINAGE-COUNTER
RETURN-CODE
SORT-CORE-SIZE
SORT-FILE-SIZE
SORT-MODE-SIZE
SORT-RETURN
TALLY
```
Figurative constants

ZERO  SPACE  HIGH-VALUE  LOW-VALUE  QUOTE
  literal  symbolic character

%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 and the source reference of the statement where the program was interrupted. For higher hierarchical levels AID outputs the name of the calling program and the source reference 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.

\[
\begin{align*}
\text{medium-a-quantity} & \text{OPERAND} \\
T & \quad H \\
F \quad n & \quad P \\
\end{align*}
\]

\[
\begin{align*}
\{ & \quad \{ \\
\ T & \quad H \\
\ F & \quad n \\
\ P & \quad P \\
\} & \quad \{ \\
\ MAX & \quad MAX \\
\ MIN & \quad MIN \\
\} \\
\end{align*}
\]

\text{medium-a-quantity} is described in detail in the AID Core Manual, chapter 8.

\begin{itemize}
  \item T  Terminal output
  \item H  Hardcopy output (includes terminal output and cannot be specified together with T)
  \item Fn File output
  \item P  Output to SYSLST
\end{itemize}

\begin{itemize}
  \item MAX  Output with additional information
  \item MIN  Output without additional information
\end{itemize}
Examples

1. %SDUMP

With this command a symbolic dump of all DATA DIVISIONs in the current call hierarchy is requested. The value for medium-a-quantity is T=MAX. The compiler listing for this SDUMP output is given in section 6.1.

<table>
<thead>
<tr>
<th>SRC_REF:</th>
<th>SOURCE:</th>
<th>PROC:</th>
</tr>
</thead>
<tbody>
<tr>
<td>57SEA</td>
<td>MOBS</td>
<td>MOBS</td>
</tr>
</tbody>
</table>

ZERO = 0
HIGH-VALUE = FF
LOW-VALUE = 00
SPACE = |
QUOTE = |

The %SDUMP output starts with a header containing the source reference of the statement at which the program was interrupted and the name of the current program. This is followed by the figurative constants, with name and contents.

<table>
<thead>
<tr>
<th>TEXTDAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>_FILE_NAME =</td>
</tr>
<tr>
<td>_OPEN_MODE = OPEN-INPUT</td>
</tr>
<tr>
<td>_RECORD =</td>
</tr>
</tbody>
</table>

File information for the file TEXTDAT.

<table>
<thead>
<tr>
<th>RECD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEM(1:61)</td>
</tr>
<tr>
<td>( 1)</td>
</tr>
<tr>
<td>( 7)</td>
</tr>
<tr>
<td>(13)</td>
</tr>
<tr>
<td>(19)</td>
</tr>
<tr>
<td>(25)</td>
</tr>
<tr>
<td>(31)</td>
</tr>
<tr>
<td>(37)</td>
</tr>
<tr>
<td>(43)</td>
</tr>
<tr>
<td>(49)</td>
</tr>
<tr>
<td>(55)</td>
</tr>
<tr>
<td>(61)</td>
</tr>
</tbody>
</table>

RECD is the data record definition for the file TEXTDAT. The contents are in the form of a table and have a permanently allocated index. The elements of the table are alphanumeric. For this reason the element contents are enclosed in vertical lines. Each value in the table is preceded by the appropriate index value in parentheses.
No level number is output for data elements of level 77 or 01.

Group items A-Z-TAB, ABC-TAB and NUMB-TAB are in the form of a table. Each consists of 26 elements. ABC-TAB is alphanumeric and is indexed with index I. NUMB-TAB is numeric and is indexed with J. Both indexes are assigned the value 1.

Definition of items in the header and footer.
COBOL special registers are output with their current values.

2. %SDUMP %NEST

The current call hierarchy is to be output.

The program was interrupted at the statement with the name 75EXI in program unit UNTER. The second line indicates the program unit EXAMPLE, from which UNDER was called using the CALL statement. The CALL statement is located in statement line 41. The current call hierarchy has two levels.
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 the result that the `%SET` statement works in the same way as the COBOL MOVE statement, apart from exceptions mentioned later.

- With `sender` you designate a data item, a length, an address, an execution counter, an AID register, a COBOL special register, a figurative constant 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 item, an execution counter, an AID register or a COBOL special register to be overwritten. `receiver` may only be located within the virtual memory of the program which has been loaded.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%S[ET]</code></td>
<td><code>sender</code> INTO <code>receiver</code></td>
</tr>
</tbody>
</table>

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.

`sender` and `receiver` may also be defined in the FILE SECTION or SUB-SCHEMA SECTION. If they are located in the LINKAGE SECTION, the latter must be contained in the current call hierarchy.

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 not all entries in the DATA DIVISION will have been initialized (e.g. record definitions and special registers).

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 items, COBOL special registers, execution counters, registers or a complex memory reference. Statement names, source references, figurative constants, AID literals and addresses and lengths of data items can only be used as sender.

sender may be located either in the virtual memory area of the loaded program (E=VM) or in a dump file; receiver, on the other hand, may only be located in the virtual memory area of the loaded program.

If program areas are transferred or overwritten with instruction code, there may be undesirable results if addresses are affected which belong to a control-area or trace-area or for which a test point has been set with %INSERT (see AID Core Manual, section 12.2 [1]).

sender-OPERANDreceivers-OPERAND

C=segmentname
C=sharename
dataname

statement-name
source-reference
keyword
compl-memref

INTO %@

filename
keyword
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.
Qualifications need only be specified if an address operand does not apply to the current AID work area or if the intention is to reference an address that is not within the current compilation unit or the current program.

**E=VM** for *receiver*

is to be specified only if the current base qualification (see %BASE command) is not to apply to *sender* or *receiver*.

*sender* can be located either in virtual memory or in a dump file, whereas *receiver* must be located in virtual memory.

**S=srcname**

is to be specified only if *sender* or *receiver* is not contained in the current compilation unit.

**PROC=program-id**

is to be specified only if you address a file name, data name or statement name that is not in the current program or is not unique in the current compilation unit (see chapter 4). It is also necessary for a global data name that is locally hidden.

If *srcname* in the S qualification is the same as *program-id*, only the PROG qualification need be written.

Only the base qualification or the CTX qualification can be placed before the C qualifications listed below. The C qualification takes the user away from the symbolic level. No symbolic operands can be written directly afterwards (see section on “Symbolic memory references”), only a *compl-memref*.

**C=segmentname**

Without a length modification, specify the entire segment as the *sender* or *receiver*.

**C=sharename**

Without a length modification, specify the entire object module as the *sender* or *receiver*.

*dataname* is the name of a group item or data element defined in the source program or the name of a COBOL special register. Figurative constants can only be used as *sender*.

*dataname* is an alphanumeric string with up to 30 characters.

AID transfers data elements in accordance with the rules for COBOL MOVE, taking into consideration the definitions from the source program.

Data items can only be processed with %SET if both *sender* and *receiver* have been defined as data items. AID executes an alphanumeric transfer, taking neither
the format nor the data type definition into account. Numeric and alphanumeric receive items with print editing can only be modified with an AID character literal (C'...', X'...' or B'...') whose contents have already been correspondently edited for printing.

\[ \text{dataname [identifier][...][(index[,...])]]} \]

**identifier**

If `dataname` is not unambiguous within a program unit, it can be identified by being assigned to a particular data item with IN or OF. `dataname` must be assigned as many identifiers as are required to designate it unambiguously. If it is not identified, AID only outputs data for `dataname` if a data definition is provided for it at level 01 or 77. If this is not the case, an error message is issued.

**index**

If `dataname` is the name of an element in a table, it can be indexed and subscripted as in a COBOL statement. The notation differs from COBOL only in that multiple indexes must be separated by a comma. If you specify the name of a table element without an index or with an incomplete index, AID aborts transfer.

`index` can be specified as follows:

\[ \text{[n[} \text{index-name dataname TALLY}\text{arithmetic expression]} \]

**COBOL special registers**

- LINAGE-COUNTER
- RETURN-CODE
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY

**Figurative constants**

can only be specified as `sender`; the address selector cannot be used on them. The figurative constants HIGH-VALUE and LOW-VALUE always represent the value assigned to them by default or in the declarations made with the PROGRAM COLLATING SEQUENCE clause. In contrast to the COBOL MOVE
statement, only one character is transferred in the AID command %SET when a figurative constant is used.

ZERO
SPACE
HIGH-VALUE
LOW-VALUE
QUOTE
literal
symbolic character

statement-name
designates the address of the first instruction in a section or paragraph in the
PROCEDURE DIVISION.

\{L\'section' [IN L\'section']\}

If a paragraph name is not unambiguous within a program, it must be identified
by the section name of the section in which it was defined: L\'paragraph' IN
L\'section'

Statement names are address constants and can only be specified for sender. The
address thus designated is then transferred.
With the subsequent pointer operator (statement-name ->) you designate 4 bytes
of the program code generated for the statement. For 2-byte or 6-byte instructions
you must specify a corresponding length modification. statement-name -> can be
used both as sender and receiver. See examples 6 and 7.

source-reference
designates the address of the first instruction generated for a statement in the
PROCEDURE DIVISION and must be specified in one of the following formats:

S'n' for lines with paragraph or section names in which no COBOL verb
occurs. This specification is not possible for programs which have been
compiled with STMT-REFERENCE=COLUMN1-TO-6.
S'nverb[m]' | S'xverb[m]' for lines containing a COBOL verb. m is specified only if the same
COBOL verb appears more than once in a line.

Source references are address constants and can only be specified for sender. The
address thus designated is then transferred.
With the subsequent pointer operator (source-reference ->) you designate 4 bytes
of the program code generated for the statement. For 2-byte or 6-byte instructions
you must specify a corresponding length modification, source-reference -> can be
used both as sender and receiver. See examples 6 and 7.
keyword

is an execution counter, the program counter or a register. Only a base qualification can be specified before keyword.
The AID Core Manual, chapter 10.1, lists the implicit storage types of the keywords.

\begin{verbatim}
%•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
\end{verbatim}

compl-memref

The following operations may occur in compl-memref (see AID Core Manual, chapter 7.5.6 [1]):

- byte offset (+)
- indirect addressing (->)
- type modification (%T(dataname), %X, %C, %D, %P, %F, %A, %S, %SX)
- 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. A type modification with a storage type that is incompatible with the memory contents will be rejected by AID.

If a compl-memref begins with statement-name or source-reference, it must be followed by a pointer operator ( -> ). In this case statement-name must be specified with L'...'. Without the pointer operator ( -> ), statement-name and source-reference can be used anywhere where hexadecimal numbers can be written. 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. Nevertheless, the area boundaries of a start address (CSECT, dataname, keyword etc.) remain in effect. They must not be exceeded for any operand in a complex memory reference by a byte offset or length modification, otherwise AID will reject the command and issue an error message. Only by combining the address selector (%@) with the pointer operator (->) can you switch to machine code level, on which the area comprises the area of virtual memory occupied by the loaded program.

**Example:**

%SET CITEM.3%L5 INTO CITEM1

The area of CITEM is five bytes long. After the byte offset, the area of CITEM would be exceeded by three bytes as a result of length modification %L5. This is not allowed. If it is intended to use the %SET command to transfer a further three
bytes to CITEX1 after CITEX, the %SET must be written as follows:

%SET %@((CITEM)->.3L5 INTO CITEX

%@(...)  
With the address selector you can output the start address of a data entry, a data item, a special register or a complex memory reference. The result supplied by the address selector is an address constant (see AID Core Manual, chapter 7.5.6 [1]).  
The address selector cannot be used for symbolic constants (including the statement names, the source references and the figurative constants).

%L(...)  
The length selector can be used to specify the length of a data entry, data item or special register as sender. The length selector produces an integer as a result (see AID Core Manual, chapter 7.5.6 [1]).  
Example: %SET %L(ARRAY1) INTO %0G  
The length of ARRAY1 will be transferred.

%L=(expression)  
With the length function you, as sender, can have a value calculated.  
expression is formed from memory references, constants, integers and arithmetic operators. Only memory reference contents that are integers (type %F or %A) are permitted. The length function produces an integer as a result. (see AID Core Manual, chapter 7.5.6 [1]).  
Example: %SET %L=(ARRAY1) INTO %0G  
The content of ARRAY1 is transferred if it is an integer (type %F). Otherwise AID issues an error message.

AID literal  
All AID literals described in the AID Core Manual, chapter 9, may be specified.  
Note well the conversion options for matching AID literals to the respective receivers as described in that chapter:

<table>
<thead>
<tr>
<th>Character literal</th>
<th>Hexadecimal literal</th>
<th>Binary literal</th>
<th>Integer</th>
<th>Hexadecimal number</th>
<th>Decimal number</th>
<th>Floating-point number</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C'x...x'}</td>
<td>{X'f...f'X}</td>
<td>{B'b...b'B}</td>
<td>[±]n</td>
<td>'#'f...f'</td>
<td>[{±}]n.m</td>
<td>[{±}]mantissaE[±]exponent</td>
</tr>
</tbody>
</table>
The following table provides an overview on permissible combinations of the sender and receiver types in conjunction with the %SET command.

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
<th>Fixed-pt., ext. float.-point no., subscript, index, special register, %D</th>
<th>Int. float.-point no.</th>
<th>Index</th>
<th>Alphabetic</th>
<th>Alphanumeric</th>
<th>Edited</th>
<th>%X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point no. extern.</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>-</td>
<td>-</td>
<td>-*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Internal fl.-point no.</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>-</td>
<td>-</td>
<td>-*</td>
<td>bin</td>
<td></td>
</tr>
<tr>
<td>ZERO, %F, %P, %A</td>
<td>[±]n</td>
<td>[±]mantE</td>
<td>[±] exponent</td>
<td>num</td>
<td>num</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alphabet, alphanumeric, edited</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>char</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alphanumeric, SPACE, QUOTE, %C</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>char</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>bin</td>
</tr>
<tr>
<td>Numeric, edited</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>char</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HIGH / LOW-VALUE</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>char</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Symbolic character</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>char</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C’x...x’</td>
<td>num</td>
<td>num</td>
<td>num</td>
<td>char</td>
<td>char</td>
<td>char</td>
<td>bin</td>
<td></td>
</tr>
<tr>
<td>X’f...f’, B’b...b’</td>
<td>bin</td>
<td>bin</td>
<td>-</td>
<td>bin</td>
<td>bin</td>
<td>bin</td>
<td>bin</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>-</td>
<td>bin</td>
<td>-</td>
<td>bin</td>
<td>bin</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* Alphabetic content, n numeric content
* Unlike COBOL, AID does not execute this transfer.

**bin**  Binary transfer; left-justified

- **sender < receiver** padding with binary zeros on the right
- **sender > receiver** truncation on the right

For transfer to %X, integral numeric literals correspond to a signed integer value with a length of 4 bytes (%FL), which are transferred in binary form.

**char**  Character transfer; left-justified, but the JUSTIFIED RIGHT clause in receiver is taken into account.

- **sender < receiver** padding with blanks (X’40’) on the right
- **sender > receiver** truncation on the right

* Transfer only carried out if the contents of sender are alphabetic.
num Numeric transfer; value is retained
If required, sender is converted to the storage type of receiver.
The SIGN LEADING/TRAILING [SEPARATE] clause is taken into account.

n If sender of the character type contains only numbers and is no more than 18 digits long, and if receiver is of the numeric type, AID performs numeric transfer; no other senders of the character type can be transferred in numeric receivers.
1 Only values > 0 can be transferred in index. AID performs the necessary conversion of table position number to table element displacement and vice versa.

– No transfer
AID indicates the incompatibility of the storage types.

Examples
The following items and tables are defined in a COBOL program:

```
01 NUMB-TAB.
  02 QNTY PIC S9(6) OCCURS 50 Indexed by J.
01 NUMB-SUM PIC S9(6).
01 PROC-SUM PIC S999V99.
01 CHAR PIC X(10).
```

For the following examples the update dialog was activated via AID CHECK=ALL. This displays the contents of the receive field before and after the execution of %SET:

1. %SET #’061’ INTO NUMB-SUM

```
OLD CONTENT:
  1
NEW CONTENT:
  9
% IDA0129 CHANGE? (Y=YES; N=NO)? Y
```

The following command produces the same result:

```
%SET 97 INTO NUMB-SUM
```

2. %QUALIFY PROG=UPRONUM
%SET .NUMB-SUM INTO .NUMB(16)

```
OLD CONTENT:
  0
NEW CONTENT:
  10
% IDA0129 CHANGE? (Y=YES; N=NO)? Y
```
3. \%SET ‘ABCDEFG’ INTO CHAR

| OLD CONTENT: |
| 1234567890 |
| NEW CONTENT: |
| ABCDEFG |

\% IDA0129 CHANGE? (Y=YES; N=NO)? Y

4. \%SET 123.45 INTO PROC-SUM

| OLD CONTENT: |
| +0.00 |
| NEW CONTENT: |
| +123.45 |

\% IDA0129 CHANGE? (Y=YES; N=NO)? Y

5. \%SET 123.45 INTO QNTY(5)

I390 WARNING: SOURCE TRUNCATED

| OLD CONTENT: |
| 0 |
| NEW CONTENT: |
| 123 |

\% IDA0129 CHANGE? (Y=YES; N=NO)? Y

6. \%SET L’OUTPUT’ INTO %0G

The address of the first instruction starting at paragraph PUTOUT is written into AID register %0G.

7. \%DA 5 FROM L’PUTOUT’->

\%SET L’PUTOUT’->%L2 INTO %1G

With DISASSEMBLE you disassemble the instruction code located at the address allocated to the paragraph PUTOUT. The first instruction is a 2-byte instruction. This first instruction is written to AID register %1G with the \%SET command.
The %SHOW command allows the user to obtain information about the current definitions relating to individual AID commands, to find out what the last entry of a command looked like, and which command was entered last. It is also possible to use the subcommand name to request the command in which it was defined or to output a list of all entered subcommand names with the associated command type. Depending on how uppercase and lowercase notation was defined in the %AID command, the original entry of the command is either reproduced or the input string is converted to uppercase letters.

- *show-target* can be used to specify a command, a subcommand name or an AID keyword for all current subcommands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SHOW</td>
<td>[show-target]</td>
</tr>
</tbody>
</table>

The effect of %SHOW without an operand is to output the AID command entered directly beforehand. If no AID command has been entered for the task, an error message is issued. A %SHOW for one of the commands for which it is not intended results in a syntax error. The command may be used in command and subcommand strings.

%SHOW does not alter the program state.
designates an AID command, a specific subcommand or all entered subcommands. The commands permitted for this command can also be specified in the abbreviated form in `show-target`.

<table>
<thead>
<tr>
<th>Command or subcommand</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>%AID</td>
<td>The current valid settings for the %AID, %AINT and %BASE commands and the version of AID loaded.</td>
</tr>
<tr>
<td>%BASE</td>
<td>The current settings for %BASE, %AINT and %SYMLIB, the TSN, TID and the version of the operating system and type of computer are output.</td>
</tr>
<tr>
<td>%C[ONTROL]</td>
<td>The input string is output for each registered %CONTROL.</td>
</tr>
<tr>
<td>%D[IS][A][SEMBLE]</td>
<td>The current number and start address (V'...') is output.</td>
</tr>
<tr>
<td>%F[IND]</td>
<td>The entered command and if appropriate the virtual address of the last hit are output.</td>
</tr>
<tr>
<td>%IN[SERT] [test-point]</td>
<td>Without the test-point entry, all active test points are output. Otherwise AID shows the entered command in which test-point was declared.</td>
</tr>
<tr>
<td>%ON</td>
<td>The input string is output for each active %ON command.</td>
</tr>
<tr>
<td>%OUT</td>
<td>The valid medium-a-quantity values for the commands that can be controlled via %OUT are output.</td>
</tr>
<tr>
<td>%OUTFILE</td>
<td>All implicitly or explicitly entered output files are listed, with their link names.</td>
</tr>
<tr>
<td>%QUALIFY</td>
<td>The last %QUALIFY command is output.</td>
</tr>
<tr>
<td>%SYMLIB</td>
<td>The registered libraries are output with the associated base qualification and the TSN.</td>
</tr>
<tr>
<td>%TRACE</td>
<td>The default values of the %TRACE operands are output. Account is taken of whether the last %TRACE was symbolic or on machine code level. In successive lines AID outputs how many instructions or statements have already been processed with the current %TRACE and what the last current %TRACE command looked like.</td>
</tr>
<tr>
<td>%•*</td>
<td>The names of all active subcommands are output with the type of the AID command in which they were defined.</td>
</tr>
<tr>
<td>%•subcmdname</td>
<td>The command in which subcmdname was defined is output.</td>
</tr>
</tbody>
</table>
%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 level of the call hierarchy 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%STOP</td>
<td></td>
</tr>
</tbody>
</table>

If the %STOP command is contained in a command sequence or subcommand, any commands following it will not be executed.

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.

Example

```plaintext
/%IN PROG=SORT.S’20EXI’ <%D TAB; %STOP>
/%RESUME

TAB( 1: 9)
( 1) Jimmy  ( 2) Maria  ( 3) Jamie  ( 4) Lesly  ( 5) Jonny  
( 6) Donna  ( 7) Marie  ( 8) Carol  ( 9) Frank  

STOPPED AT SRC_REF: 20EXI , SOURCE: SORT , PROC: SORT
```

%INSERT sets a test point for statement EXIT from line 20. The subcommand comprises the %DISPLAY and %STOP commands. After TAB has been output, AID halts the program and writes a STOP message indicating the source reference and program of the current interrupt point.
%SYMLIB

%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 for which no LSD records have been loaded are addressed in a command.

- With 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SYMLIB</td>
<td>[qualification-a-lib][,...]</td>
</tr>
</tbody>
</table>

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 with %DUMPFILE 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 it is closed by:
- a new %SYMLIB command for the same base qualification
- a %SYMLIB without an operand
- a %DUMPFILE command with which the file is closed or by /LOGOFF or /EXIT-JOB.

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------------------------

[•][E=VM
[•][filename]

- 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 and %DUMPFILE commands).

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,COBOL85OUTPUT

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 COBOL85OUTPUT 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.
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.

- With page-header 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%TITLE</td>
<td>[page-header]</td>
</tr>
</tbody>
</table>

With a %TITLE command without a page-header operand you switch back to the AID standard header. AID resets the page counter to 1 and positions SYSLST to the top of the page before the next line to be printed.

A page header defined with %TITLE remains valid until a new %TITLE command is issued or until the program ends.

The %TITLE command does not alter the program state.

page-header

Specifies the variable part of the page title. AID completes this specification by adding the time, date and page counter.

page-header is a character literal in the format {C'x...x' | 'x...x'C | 'x...x'} and may have a maximum length of 80 characters. A longer literal is rejected with an error message outputting only the first 52 positions of the literal.

Up to 58 lines are printed on one page, not counting the title of the page.
%TRACE

With the %TRACE command you switch on the AID tracing function and start the program or continue it at the interrupt point or the point specified in the %JUMP command.

- With `number` you can specify the maximum number of COBOL statements to be traced, to be logged before execution.
- With `criterion` you select different types of COBOL statements which AID is to log. Logging takes place prior to execution of the statements selected.
- With `trace-area` you define the program area in which the `criterion` is to be taken into consideration.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>%T[TRACE]</td>
<td>[number] [criterion][,...] [IN trace-area]</td>
</tr>
</tbody>
</table>

If the program is interrupted during a %TRACE, the %TRACE can be continued with %CONTINUE. This applies to the following cases:

- A subcommand containing a %STOP command has been executed.
- An %INSERT command ends with a program interrupt because the `control` operand is K or S.
- The K2 key has been used (see section 3.3).

The %TRACE command is terminated, on the other hand, by the following events:

- The maximum number of statements to be traced has been reached.
- A subcommand containing a %RESUME or %TRACE command has been executed.
- After one of the program interrupts described above, the program continues with %RESUME.

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 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.
The %TRACE command alters the program state.

`number` specifies the maximum number of COBOL statements of type `criterion` which are to be executed and logged.

`number` is an integer $1 \leq number \leq 2^{31}-1$. The default value is 10. If there is no value from a previous %TRACE command, AID inserts the default value in a %TRACE command without the `number` operand.

After the specified `number` of statements has been traced, AID outputs a message via SYSOUT, the program is halted and the user can enter AID or BS2000 commands. The message tells you at which statement and in which program the current interrupt point is located.
criterion is a keyword which defines the type of statements to be traced during program execution. Several keywords can be specified at a time; they take effect simultaneously. A comma must be used to separate any two keywords. If no criterion is declared, AID uses the default value %STMT unless a criterion declaration from an earlier %TRACE command is still valid.

<table>
<thead>
<tr>
<th>criterion</th>
<th>logging prior to</th>
</tr>
</thead>
<tbody>
<tr>
<td>%STMT</td>
<td>Every COBOL statement that is executed</td>
</tr>
<tr>
<td>%ASSGN</td>
<td>COBOL statements which modify the contents of a data field: ADD [CORRESPONDING], COMPUTE, DIVIDE, INITIALIZE, INSPECT, MOVE [CORRESPONDING], MULTIPLY, SET, STRING, SUBTRACT [CORRESPONDING], UNSTRING</td>
</tr>
<tr>
<td>%CALL</td>
<td>CALL, PERFORM statements as well as prior to SORT/MERGE statements, since these may call an INPUT or OUTPUT procedure.</td>
</tr>
<tr>
<td>%COND</td>
<td>EVALUATE, IF, ON and SEARCH statements and the conditional THEN, ELSE and WHEN statement branches.</td>
</tr>
<tr>
<td>%DB</td>
<td>COBOL statements for calling a database: CONNECT, DISCONNECT, ERASE, FETCH, FIND, FINISH, FREE, GET, KEEP, MODIFY, READY, STORE</td>
</tr>
<tr>
<td>%EXCEPTION</td>
<td>The conditional statement branches and their admissible negations: AT END, AT END OF PAGE, INVALID KEY, ON SIZE ERROR, ON OVERFLOW, as well as prior to the execution of a USE PROCEDURE.</td>
</tr>
<tr>
<td>%GOTO</td>
<td>ALTER, CONTINUE, GOTO statements.</td>
</tr>
<tr>
<td>%IO</td>
<td>COBOL statements which initiate I/O operations: ACCEPT, DISPLAY, OPEN, CLOSE, DELETE, READ, REWRITE, START, WRITE, GENERATE, INITIATE, TERMINATE</td>
</tr>
<tr>
<td>%LAB</td>
<td>COBOL statements which have a section or paragraph name or which directly follow such a name.</td>
</tr>
<tr>
<td>%PROC</td>
<td>Program or module start at the beginning of the PROCEDURE-DIVISION or at ENTRY. Program or module end by the statement STOP RUN or EXIT PROGRAM.</td>
</tr>
<tr>
<td>%SORT</td>
<td>MERGE and SORT statements, RELEASE and RETURN statements.</td>
</tr>
</tbody>
</table>
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. trace-area can only be located within the loaded program, and the program that is specified must be loaded at the time when the %TRACE command is entered or the subcommand containing the %TRACE command is processed.

trace-area is limited to a compilation unit in programs without segmentation, and to a segment in programs with segmentation. The limitation to one segment applies only for independent segments (segment No. ≥ 50).

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 or segment containing the current interrupt point.

The continuation address for program execution cannot be influenced by the %TRACE command; such is only possible by means of the %JUMP command.

trace-area-OPERAND ----------------------------
[S=srcname] [•]PROC=program-id

IN [•][E=VM•] [S=srcname•] {statement-name}

C=segmentname C=sharename

• 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).
S=srcname
This must be specified if *trace-area* is not to be included in the current compilation unit.

PROC=program-id
This need only be specified if *trace-area* is not to be contained in the current program, if it is to be defined with *statement-name* and this is not unique within the compilation unit, or in order to overwrite a previously valid *trace-area* definition.

If *trace-area* ends with a PROC/PROG qualification, it comprises the whole of the specified program. The program must be loaded at the time when the %TRACE command is entered or the subcommand containing the %TRACE command is processed.

If the name in the S qualification is the same as *program-id*, only the PROG qualification need be written.

Although the following C qualifications have the effect of switching to machine code level, they can only be followed by a criterion selected from the preceding table or by the default value %STMT inserted by AID.

C=segmentname
This specification defines the designated segment as the *trace-area*. It is only necessary if the interrupt point is not contained in this segment or if a previously applicable area restriction applying to parts of the segment is to be removed.

C=sharename
This specification defines the designated object module as the *trace-area*. It is only necessary if the interrupt point is not located in the specified object module or if an area restriction applying to the object module is to be removed.

*statement-name*
The *trace-area* is defined by a statement name and comprises a section or paragraph in the PROCEDURE DIVISION.

```
[L'section' [IN L'section']]
```

An alphanumeric section or paragraph name can be specified without L’...’ since this name cannot be confused with a data name in this command.

If a paragraph name is not unambiguous within a program, it must be identified by the section name of the section in which it was defined: L’paragraph’ IN L’section’
The trace-area is defined by specifying a start address and an end address. The start and end addresses must both be within the same compilation unit and the following must apply:
start address \( \leq \) end address.

If the trace-area is to cover only one statement line, the start address and the end address must be identical. It is not possible to limit trace-area to individual COBOL verbs within a line.

The source-reference designates the address of the first instruction generated for a statement in the PROCEDURE DIVISION and must be specified in one of the following formats:
- \( S'n' \) for lines with paragraph or section names in which no COBOL verb occurs. This specification is not possible for programs which have been compiled with \( \text{STMT-REFERENCE=COLUMN1-TO-6} \).
- \( S'nverb[m]' \) | \( S'xverb[m]' \) for lines containing a COBOL verb.

Output of the %TRACE listing
The %TRACE listing is output in full format via SYSOUT as a standard procedure (%OUT operand value T=MAX). With the %OUT command, you can define the output media and the scope of information to be output (see AID Core Manual, chapter 8 [1]).

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

A %TRACE listing without additional information (T=MIN) does not show the statement type.
Examples

1. 

```
/OUT /TRACE T=MAX
/ATS 3
  49    I2 LABEL
  50MOV ASSIG
  51ADD ASSIG
STOPPED AT SRC_REF: 51ADD, SOURCE: EXAMPLE, PROC: EXAMPLE
```

With the aid of the %OUT command, output is switched back to the terminal and the maximum range of information is defined for output.

The %TRACE command is to trace three COBOL statements. After the third statement the termination message for this %TRACE command follows, to the effect that statement 51 is in the program unit EXAMPLE and that the load module has the same name.

2. 

```
/OUT /T T-MIN
/ATS 3
  49    12
  50MOV
  51ADD
STOPPED AT SRC_REF: 51, SOURCE: EXAMPLE, PROC: EXAMPLE
```

With the %OUT command the range of information for the %TRACE command is reduced. A subsequently entered %TRACE command outputs the log without additional information.
6 Sample application

This chapter illustrates an AID debugging session for a short COBOL 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 compiler was called with the following SDF command:

```
/START-COBOL85-COMPILED SOURCE=COB.S.SRCDAT,TEST-SUPPORT=AID-
// (PREPARE-FOR-JUMPS=YES),-
/LISTING=PARAM(NAME-INFORMATION=YES(SUPPRESS-GENERAT=AT-SEVERE-ERR),-
/OUTPUT=*SYSLST),COMPILER-ACTION=MODULE-GEN(MOD-FORM=OM,-
/SUPPR-GEN=AT-SEVERE-ERR),-
/RUNTIME-OPTIONS=PARAM(ACCEPT-STMT-INPUT=UPPERCASE-CONVERTED)
```

6.1 Source listing

```
00001 IDENTIFICATION DIVISION.
00002 PROGRAM-ID. M0BS.
00003 ENVIRONMENT DIVISION.
00004 CONFIGURATION SECTION.
00005 SPECIAL-NAMES.
00006 TERMINAL IS T.
00007 INPUT-OUTPUT SECTION.
00008 FILE-CONTROL.
00009 SELECT TEXTDAT ASSIGN TO "INPFIL".
00010 DATA DIVISION.
00011 FILE SECTION.
00012 FD TEXTDAT
00013    RECORD VARYING FROM 1 TO 256 DEPENDING ON SLF.
00014 01 REC.
00015    02 ITEM PIC X OCCURS 1 TO 256 DEPENDING ON SLF
00016 INDEXED BY K.
00017 WORKING-STORAGE SECTION.
00018 77 SLF PIC 999 COMP.
00019 77 PROCES-SWITCH PIC X.
00020 88 PROCES-END VALUE "1".
00021 01 A-Z-TAB.
00022    02 FILLER PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
00023 01 ABC-TAB REDEFINES A-Z-TAB.
00024    02 CHAR PIC X OCCURS 26 INDEXED BY I.
00025 01 NUMB-TAB.
00026    02 QNTY PIC 9999 OCCURS 26 INDEXED BY J.
```
Sample application

Source program

00027  77 NUMB-SUM PIC S9(6) VALUE ZERO.
00028  77 PROC-SUM PIC S999999 VALUE ZERO.
00029  01 FRM-HEAD.
00030    02 FILLER PIC X(24) VALUE "LETTER NUMB PERCENT".
00031  01 FRM-LINE.
00032    02 LETTER PIC X.
00033    02 FILLER PIC X(9) VALUE SPACE.
00034    02 NUMB PIC Z(5)9.
00035    02 FILLER PIC X(2) VALUE SPACE.
00036    02 PERCENT PIC ZZ9.99.
00037  01 FRM-FOOT.
00038    02 FILLER PIC X(10) VALUE "TOTAL: ".
00039    02 A-SUM PIC Z(5)9.
00040    02 FILLER PIC X(2) VALUE SPACE.
00041    02 P-SUM PIC ZZ9.99.
00042  PROCEDURE DIVISION.
00043     LEADER.
00044     INITIALIZE NUMB-TAB.
00045     MOVE "0" TO PROCES-SWITCH.
00046     OPEN INPUT TEXTDAT.
00047     PROCESSING.
00048     READ TEXTDAT
00049        AT END DISPLAY "FILE IS EMPTY" UPON T
00050        NOT AT END
00051        PERFORM WITH TEST BEFORE UNTIL PROCES-END
00052        PERFORM WITH TEST BEFORE VARYING K FROM 1 BY 1 UNTIL
00053           K > SLF
00054           IF ITEM(K) NOT = SPACE
00055             THEN ADD 1 TO NUMB-SUM
00056             SET I TO 1
00057             SEARCH CHAR VARYING J
00058             WHEN ITEM(K) = CHAR(I) ADD 1 TO QNTY(J)
00059             END-SEARCH
00060             END-IF
00061        END-PERFORM
00062        READ TEXTDAT
00063        AT END SET PROCES-END TO TRUE
00064        END-READ
00065        END-READ
00066     END-READ.
00067     PUTOUT.
00068     CLOSE TEXTDAT.
00069     DISPLAY FRM-HEAD UPON T.
00070     PERFORM WITH TEST BEFORE
00071        VARYING I FROM 1 BY 1 UNTIL I > 26
00072        MOVE CHAR(I) TO LETTER
00073        SET J TO I
00074        MOVE QNTY(J) TO NUMB
00075        COMPUTE PERCENT = QNTY(J) * 100/NUMB-SUM
00076        DISPLAY FRM-LINE UPON T
00077        COMPUTE PROC-SUM = PROC-SUM + QNTY(J) * 100/NUMB-SUM
00078        END-PERFORM.
00079        MOVE PROC-SUM TO P-SUE.
00080        MOVE NUMB-SUM TO A-SUM.
00081        DISPLAY FRM-FOOT UPON T.
00082        STOP RUN.
6.2 Contents of the input file

This is a file used as input for a program which determines the frequency of letters. The total number comprises the number of characters that are not blanks (including numbers and special characters).

DIES IST EINE DATEI, DIE ALS EINGABE DIENET FUER EIN PROGRAMM, DAS DIE HAEUFIGKEIT VON BUCHSTaben BESIMMT. DIES IST EINE DATEI, DIE ALS EINGABE DIENET FUER EIN PROGRAMM, DAS DIE HAEUFIGKEIT VON BUCHSTaben BESIMMT.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
NUN NOCH EIN PAAR NORMALE SAETZE
ANNABELL WAR AUCH IN DER ALHAMBRA
BABETTE BEMALTE BEIM BAECKER DIE BALUSTRADE
CAESAR CHECKTE SICH EIN NACH CHICAGO

6.3 Test run

Step 1

/SET-FILE-LINK LINK-NAME=INPFIL, FILE-NAME=INP
/START-PROGRAM FROM-FILE = MOBS
% BLS0517 MODULE 'MOBS' LOADED
9089 INTERRUPT-CODE= 60 AT PC= 00000C46
% EXC0732 ABNORMAL PROGRAM-TERMINATION. ERROR-CODE 'NRT0101': /HELP-MSG NRT0101

The input file INP is assigned to the program. The MOBS program is started. The program run is aborted due to a data error (EC=60).

Step 2

/SET-FILE-LINK LINK-NAME=BLSLIB00,FILE-NAME=$.SYSLNK.CRTE
/SET-FILE-LINK LINK-NAME=INPFIL,FILE-NAME=INP
/LOAD-P *MOD(BIBLIO,MOBS,RUN-MODE=ADVANCED(ALT-LIB=YES)),TEST-OPTIONS=AID
% BLS0517 MODULE 'MOBS' LOADED

The program is loaded once more, this time with LSD information. It is not started immediately so as to allow AID commands to be entered first.

/%ON %ANY; %RESUME

The %ON %ANY is intended to ensure that it is possible to enter further AID commands before the end of the program. The program is subsequently started again with %RESUME.
The program encounters the known error.

```
/*display \hlloc(v’c46’)
** TID: 00076132 ** TSN: 6DMA ***************************************
CURRENT PC: 00F4DCBC CSECT: IT0TRM@ ***************************************
V’00000C46’ = CONTEXT : LOCAL#DEFAULT
SMOD : MOBS
PROC : MOBS
SECTION :
PARAGRAPH: PROCESSING
SRC-REF : 58ADD
```

The %DISPLAY command determines the symbolic address at which the data error occurs. It is the source reference S’58ADD’.

```
/*display number(j)
% AID0379 S and PROC qualification required or LSD information missing
```

The aim here in the program, after the character from the input record has been assigned to A-Z-TAB accordingly, is to increment the counter for this letter by 1. A %DISPLAY command relating to the table location referenced by index in the COBOL statement ADD is rejected by AID. As was apparent from the previous output relating to the high level location (%HLLOC) of the current interrupt point, this is contained in the IT0TRM@ module of the runtime system. Symbolic addresses in the MOBS program must therefore be qualified in AID commands. AID requires the S and PROC qualification here, or the PROG qualification can be used for the short PROGRAM-ID ‘MOBS’.

```
/*d prog=mobs.number(j)
% AID0400 Dimension 01 of array NUMBER out of range or array has no element
```

The new error message indicates that the index J for number has an invalid value. AID cannot work with this address either.

```
/*qualify prog=mobs
```

The prequalification is defined so as not to have to write the qualification repeatedly in further AID commands. In subsequent commands all that is necessary is to insert a period in front of a symbolic address instead of the qualification.
AID outputs the content of J. The maximum value of the index is 26; however, J contains 42. This resulted in the data error.

The data error occurred during processing of the sixth character from the input record.

The value of index I is 9. It is correctly positioned at the location of the letter 'I' in the alphabet.

The entire input record is output with %DISPLAY. AID edits it as a table in accordance with the definition in the source program.
Step 3

```
/LOAD-P *MOD(BIBLIO,MOBS,RUN-MODE=ADVANCED(ALT-LIB=YES)),TEST-OPTIONS=AID
% BLS0500 PROGRAM 'MOBS' LOADED
/
%TRACE IN PROCESSING
47 PROCESSING
  48REA I-O-ACCESS
  51PER EXCEPT.DEP, THEN , CALL , LOOP INIT
  52PER CALL , LOOP INIT
  54IF IF
  55ADD THEN , ASSIGN
  56SET ASSIGN
  57SEA CASE , LOOP INIT
  58ADD WHEN/OTHERS, ASSIGN
  54IF IF
STOPPED AT SRC_REF: 54IF SOURCE: MOBS PROC: MOBS
```

The program is loaded again. The %TRACE in the PROCESSING paragraph is used to show the context of the ADD statement.

Step 4

```
/%control1 %assgn in Processing <con1: %d item(k),i,j>
CON1 subcommand is to be executed. The character to be processed from the input record and the status of indices I and J are then output.

/%in s’54if’ <ins1: (%.con1 gt 10): %stop>
The run is to be interrupted after the CON1 subcommand has been executed 10 times.

/%in s’58add’ <ins2: (j gt 26): %stop>
Before the addition in NUMBER(J) is executed, AID checks whether index J has a permissible value. If it is too high, AID interrupts the program. The program is started after input of the %INSERT command.
```

```
*** TID: 0009027E *** TSN: 634R *****************************************************
SRC_REF: 55ADD SOURCE: MOBS PROC: MOBS *****************************************************
ITEM( 1) = |D| I = +1
  J = +1
SRC_REF: 56SET SOURCE: MOBS PROC: MOBS *****************************************************
ITEM( 1) = |D| I = +1
  J = +1
```
From the AID log of subcommand CON1 it can be seen that processing of the first character from the input file is running correctly. Indices I and J run in parallel. From the second character onwards, J begins to grow more quickly. At the fourth letter, the index increases to 34. Before a data error occurs again at source reference S'58ADD', the conditional subcommand INS2 is executed, as a result of which execution is interrupted. For the letter ‘S’, index I is correctly at position 19 in the A-Z-TAB table. It can be seen from the log that index I is reset to the initial value of 1 for processing a character, but index J is not.

```%set i into j %r
```

Index J is set to the contents of I and the program is continued.
The CON1 subcommand is still active and the associated execution counter has not been changed, and the %STOP in subcommand INS1 is therefore executed again.

The INS1 subcommand is deleted and the program is continued with %RESUME.

Once again, the condition (J greater than 26) for subcommand INS2 has been met and the program halted.

Index J is again too high. Before the COBOL statement SEARCH is executed index I is set to 1 again, but it was forgotten to assign index J with the initial value as well.

Before the SEARCH statement S’57SEA’ is executed, index J is now also to be assigned the correct initial value via the new %INSERT. To ensure that the statement S’58ADD’, before which the program was interrupted by the %STOP in subcommand INS2, is executed correctly, the value of index I is transferred to J with the single %SET.
As it can be seen that the program is now running correctly, output is interrupted with the K2 key.

%SHOW is used to check which %INSERT and %CONTROL commands are still active. The INS2 subcommand is now superfluous and is deleted. The outputs of subcommand CON1 are also no longer required. Only the correction of the third %INSERT is necessary. The program is continued.
The program now runs through to the end and outputs the result list. As invalid indices were used at the start of the program run, some results may not yet be correct. The program must be executed once more with the AID correction.

Step 5
The program is loaded again. With %INSERT you set a test point to the SEARCH statement in line 56. Whenever the program reaches this test point, J is set to 1. The program is started and outputs the required table.
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, C qualifications (debugging on machine code level) or PROG qualifications (symbolic debugging). The memory location or area is located either in the program which has been loaded or in a memory dump in a dump file. If a name has been assigned more than once in a user program and thus no unique address reference is possible, area qualifications or an identifier can be used to assign the name unambiguously to the desired address.

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 9 [1]):

\begin{verbatim}
(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' Hexadecimal number
[{±}]n.m Decimal number
[{±}]mantissaE[{±}]exponent Floating-point number
\end{verbatim}
AID output files
AID output files are files to which the user can direct output of the %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE commands. The files are addressed via their link names (F0 through F7) in the output commands (see %OUT and %OUTFILE). The REP records are written to the file assigned to link name F6 (see %AID REP=YES and %MOVE).

There are three ways of creating an output file:
1. /%OUTFILE command with link name and file name
2. /FILE command with link name and file name
3. For a link name to which no file name has been assigned, AID issues a FILE macro with the file name AID.OUTFILE.Fn.

An AID output file always has the format FCBTYPE=SAM, RECFORM=V and OPEN=EXTEND.

AID standard work area
This is the non-privileged part of virtual memory (in the user task) which is occupied by the program and all its connected subsystems.

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. It comprises the non-privileged part of virtual memory in the user task, which is occupied by the program and all its connected subsystems or the corresponding area in a memory dump.

Using the %BASE command, you can shift the AID work area from the loaded program to a memory dump, or vice versa. You may deviate from the AID work area in a command by specifying a qualification in the address operand.

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 PROC/PROG qualifications, the area limits are determined by the start and end addresses of the program unit (see AID Core Manual, chapter 7 [1]).
area qualification
These qualifications are used to identify part of the work area. If an address operand ends with one of these qualifications, the command is effective only in the part that is identified by the last qualification. An area qualification delimits the active area of a command, or makes a data name or statement name unique within the work area, or allows a name to be reached that would otherwise not be addressable at the current interrupt point.

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
This is the qualification designating either the loaded program or a memory dump in a dump file. It is specified via 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.

compilation unit
This consists of a single source program or a sequence of such programs. It is addressed via the S qualification.

constant
A constant represents a value which cannot be accessed via an address in program memory.
Constants include the figurative constants, the results of length selection, length function and address selection, and the statement names and source references.
An address constant represents an address. Address constants include statement names, source references and the result of an address selection. They can be used, in conjunction with a pointer operator (->), to address the corresponding memory location.

CSECT information
is contained in the object structure list.

current call hierarchy
The current call hierarchy represents the status of subprogram nesting at the interrupt point. It ranges from the subprogram level on which the program was interrupted to the subprograms exited by CALL statements (intermediate levels) to the main program.
The hierarchy is output using the %SDUMP %NEST command.

current compilation unit
The current compilation unit is the unit containing the current interrupt point.

current program
The current program is the program unit in which the compilation unit was interrupted. Its name is output in the STOP message.

data item
This is a general term for all the data defined in the DATA DIVISION, covering group items and tables and the elements in these.
**data name**

An operand that stands for all names assigned for data in the source program. With the aid of the data name the user addresses data items during symbolic debugging. No LSD records are generated for definitions from the REPORT-SECTION, for 88 levels, for system switches in the SPECIAL-NAMES paragraph and the NATIVE alphabet. Thus you cannot use AID to address this data.

If a data name is not unambiguous within a program unit, it can be identified by being assigned to a specific group item with IN or OF.

Table elements can be addressed via an index as in COBOL.

**data type**

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

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

Not all data types that are numeric in COBOL are of the storage type numeric for AID (see %SET table).

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

**ESD**

The External Symbol Dictionary (ESD) lists the external references of a module. It is generated by the compiler and contains, among other items, information on CSECTs, DSECTs and COMMONs. The linkage editor accesses the ESD when it creates the object structure list.

**global settings**

AID offers commands facilitating addressing, saving input efforts and enabling the behavior of AID to be adapted to individual requirements. The presetsings specified in these commands continue to apply throughout the debugging session (see %AID, %AINT, %BASE and %QUALIFY).

**index**

The index is part of an address operand and permits the position of a table element to be defined. It can be specified in the same way as in COBOL (in contrast to COBOL, however, multiple indexes must be separated by commas) or by means of an arithmetic expression from which AID calculates the index value. This AID-specific index contains both the address of a table element with a subscript and the COBOL-specific index from the INDEXED BY clause.
index-name
This is the symbolic name defined in the INDEXED BY clause for indexing a table level. **index-name** may not be used to index another table. If the AID index is to be calculated from an arithmetic expression, **index-name** can be linked only with integers, not with other data items of the COBOL special register TALLY.

input buffer
AID has an internal input buffer. If this buffer is not large enough to accommodate a command input, the command is rejected with an error message identifying it as too long. If fewer of the repeatable operands are specified, the command will be accepted.

interrupt point
The interrupt point is the address at which a program has been interrupted. From the STOP message the user can determine both the address at which and the program unit in which the interrupt point is located. The program is continued at this point. A different continuation address can be specified for COBOL85 and FOR1 programs with the aid of the %JUMP command.

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 6.4 [1]).

localization information
%DISPLAY %HLOC(memref) for the symbolic level and %DISPLAY %LOC(memref) for the machine code level cause AID to output the static program nesting for a given memory location. Conversely, %SDUMP %NEST outputs the dynamic program nesting, i.e. the call hierarchy for the current program interrupt point.

LSD
The List for Symbolic Debugging (LSD) is a list of the data/statement names defined in the module. It also contains the compiler-generated source references. The LSD records are created by the compiler. AID uses them to fetch the information required for symbolic addressing.

memory object
A memory object is formed by a set of contiguous bytes in memory. At program level, this comprises the program data (if it has been assigned a memory area) and the instruction code. Other memory objects are all the registers, the program counter, and all other areas that can only be addressed via keywords. Conversely, any constants defined in the program, as well as statement names, source references, the results of address selection, length selection and length function, and the AID literals do not constitute memory objects because they represent a value that cannot be changed.
memory reference
A memory reference addresses a memory object. Memory references can either be simple or complex.
Simple memory references on machine code level are virtual addresses and CSECTs. Symbolic memory references comprise all names (recorded in the LSD information) of files, data and statements from the program, the source references generated by the compiler and the AID keywords. 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 file names, data names, special registers and figurative constants 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 linkage editor option TEST-OPTIONS=AID applies.

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 10.1, lists the AID-specific storage types together with their output types. This assignment also applies for the data types used in COBOL. A type modification in %DISPLAY and %SDUMP causes the output type to be changed as well.

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

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

2. The program is running without tracing.
   %RESUME started or continued the program. %CONTINUE does the same, with the exception that any active %TRACE is continued.
3. The program is running with tracing.
%TRACE started or continued the program. The program sequence is logged in accordance with the declarations made in the %TRACE command. %CONTINUE has the same effect if a %TRACE is still active.

program unit
A subset of a complete COBOL program with a separate name in the PROGRAM-ID, e.g. the main program or any subprogram called with CALL. It can be addressed with a PROC or C qualification (segment, shared code module).

qualification
A qualification is used to reference an address which is not in the AID work area or not uniquely defined therein. The base qualification specifies whether the address is in the loaded program or in a memory dump. The S qualification specifies the compilation unit in which the memory object is situated. The PROC qualification or C qualification specifies the program unit or segment 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 PROC qualification is specified for a data element of the current program unit, except in %SDUMP.

source reference
A source reference designates an executable statement and is specified via S'n[verb[m]] | S'xverb[m]
Source references are generated by the compiler and stored in LSD records.

verb is the defined abbreviation of a COBOL verb (see chapter 4.2).
m is a number which you only need to specify if the same COBOL verb appears more than once in a statement line.

Source references are address constants.

special register
The COBOL compiler provides special registers for every program:
LINAGE-COUNTER
RETURN-CODE
SORT-CORE-SIZE
SORT-FILE-SIZE
SORT-MODE-SIZE
SORT-RETURN
TALLY

A TALLY special register is created for each program. The RETURN-CODE special register, on the other hand, is provided just once for the entire compilation unit. The SORT special registers are generated only if the program contains a sort section.
statement name
This designates the address of the first instruction in a section or paragraph in the
PROCEDURE DIVISION.

[L’section’
   [L’paragraph’ [IN L’section’]]]

If a statement name cannot be confused with a data name, an alphanumeric section
or paragraph name can be specified without L’...’. If a paragraph name is not
unambiguous within a program unit, it can be identified with IN L’section’. Statement
names are address constants.

storage type
This is either the data type defined in the source program or the one selected by
way of type modification. AID knows the storage types %X, %C, %P, %D, %F, %A,
%S and %SX
(see %SET and AID Core Manual, sections 7.5.6 and 10.1 [1]).

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, sections 5.3 and 6, 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).
References

[1] AID (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

Target group
Programmers in BS2000

Contents
– Overview of the AID system
– Description of facts and operands which are the same for all programming languages
– Messages
– Comparison between AID and IDA

Applications
Testing of programs in interactive or batch mode

Advanced Interactive Debugger
Debugging on Machine Code Level
User Guide

Target group
Programmers in BS2000

Contents
– Description of the AID commands for debugging on machine code level
– Sample application

Applications
Testing of programs in interactive or batch mode
Advanced Interactive Debugger
Debugging of FORTRAN Programs
User Guide

Target group
FORTRAN programmers

Contents
– Description of the AID commands for symbolic debugging of FORTRAN programs
– Sample application

Applications
Testing of FORTRAN programs in interactive or batch mode

Advanced Interactive Debugger
Debugging of PL/I Programs
User Guide

Target group
PL/I programmers

Contents
– Preparations for the symbolic debugging of PL/I programs
– Description of all the AID commands available for symbolic debugging
– Examples of AID sessions
– Messages

Applications
Debugging of PL/I programs in interactive and batch modes

Advanced Interactive Debugger
Debugging of ASSEMBH Programs
User Guide

Target group
Assembly language programmers

Contents
– Description of the AID commands for symbolic debugging of ASSEMBH-XT programs
– Sample application

Applications
Testing of ASSEMBH-XT programs in interactive or batch mode
References

Executive Macros
User Guide

Target group
- BS2000 assembly language programmers (non-privileged)
- System administrators

Contents
- All Executive macros in alphabetical order with detailed explanations and examples; selected macros for DMS and TIAM
- Macro overview according to application areas
- Comprehensive training section dealing with eventing, serialization, inter-task communication, contingencies

Applications
BS2000 application programs

Programmiersystem *
Technische Beschreibung
(Programming System, Technical Description)

Target group
- BS2000 users with an interest in the technical background of their systems (software engineers, systems analysts, computer center managers, system administrators)
- Computer scientists interested in studying a concrete example of a general-purpose operating system

Contents
Functions and principles of implementation of
- the linkage editor
- the static loader
- the Dynamic Linking Loader
- the debugging aids
- the program library system

Order number
U3216-J-Z53-1
References

[8] COBOL85 (BS2000)
COBOL Compiler
Reference Manual

Target group
COBOL users of BS2000

Contents
- Structure and elements of a COBOL program (ANS 85)
- File processing
- Table handling
- Report writer
- Sort/merge
- Communication
- Segmentation
- Source text manipulation

COBOL Compiler
User’s Guide

Target group
COBOL users of BS2000

Contents
- Generation of the COBOL85 compiler and the software required for the linking, loading and debugging of COBOL programs
- File processing with COBOL programs
- Inter-program communication
- Structure of the COBOL85 system
- Compiler messages and runtime system messages
- Meldungen des Compilers und des Laufzeitsystems;
Advanced Interactive Debugger

**Debugging of C/C++ Programs**
User Guide

*Target group*
C/C++ programmers

*Contents*
Description of the AID commands for symbolic debugging of C/C++
programs;
sample application

*Applications*
Testing of C/C++ programs in interactive or batch mode

Advanced Interactive Debugger

**Ready Reference**

*Target group*
Programmers in BS2000

*Contents*
- Debugging of programs written in ASSEMBH, C/C++, COBOL,
  FORTRAN, PL/I and ar machine code level
- Summary of the AID commands and operands
- %SET tables
- Comparison of AID and IDA

*Applications*
Testing of programs in interactive or batch mode

The publication marked with an * is not published by Siemens Nixdorf
Informationssysteme AG or by Siemens AG.

**Ordering manuals**
The manuals listed above and the corresponding order numbers are to be found in the
*List of Publications* issued by Siemens Nixdorf Informationssysteme AG, which also
tells you how to order manuals. New publications are listed in the *Druckschriften-
Neuerscheinungen* (New Publications).

You can arrange to have both of these sent to you regularly by having your name
placed on the appropriate mailing list. Your local office will help you.
%1G 56
%? 62
%*subcmdname 77
%*subcmdname 112
%0G 56
%AID 22, 73, 79, 108
%AINT 26
%AMODE 26
%BASE 28, 37, 56, 57
%CLASS6 57
%CONTINUE 30, 71
%CONTROL 96
%CONTROLn 31
%DISASSEMBLE 37, 89, 91, 123
%DISASSEMBLE log 40
%DISPLAY 26, 43, 89, 91, 123
%DUMPFILE 28, 54
%ERRFLG 97
%FIND 56
%H %? 62
%H? 62
%HELP 62, 89, 91, 123
%HELP information, English or German 22
%HLOC 47
%INSERT 64, 96
%JUMP 71, 99, 124
%L=(expression) 113
%LPOV 97
%M[ODE]24 27
%M[ODE]31 27
%MOVE 22, 26, 73
%n 77, 112
%nD 77, 112
%nDG 77, 112
%nE 77, 112
Index

%NEST 103
%nG 77
%nQ 77, 112
%ON 81, 96
%OUT 37, 43, 49, 63, 89, 103, 124
%OUTFILE 80, 91
%OUTFILE command 23
%PC 77, 97
%QUALIFY 93
%REMOVE 31, 68, 87, 96
%RESUME 71, 99
%SDUMP 89, 91, 100, 123
%SET 107
%SET table 114
%STOP 64, 81, 119
%STOP within a subcommand 119
%SVC 97
%SYMLIB 100, 120
%TITLE 123
%TRACE 71, 89, 91, 99, 123, 124
terminating 124
%TRACE listing 129
24-bit address 26
31-bit address 26

A
abbreviations of COBOL verbs 19
additional information 89, 90, 103
address 43, 74, 79, 108
address operand 3, 12, 38, 45, 58, 65, 74, 93, 109
address selection 40, 48, 60, 67, 78, 84, 112
address selector 48, 78, 113
addressing mode 26
AID address interpretation 26
default 26
AID commands, help texts 62
AID literal 43, 49, 73, 74, 79, 108, 113
AID message number range 62
AID output 37, 43, 49, 63, 103, 129
delimiter 22
AID output file
assign 91
close 91
open 91
AID register 47, 56, 74, 77, 112
AID registers 56
AID standard work area 28
AID work area 26, 28, 54, 90, 93
aid-mode 26
alignment 56, 61
ALL 56
alter program state 119
area qualification 12
arithmetic-expression 16
assign AID output file 91
assign PLAM library 120

B
base qualification 11, 26, 29, 33, 38, 45, 58, 65, 75, 77, 93, 101, 109, 121, 127
base 28, 29
branch 71
brief description of commands 62
BS2000 catalog name of a PLAM library 121
byte boundary, search at 61
byte offset 40, 48, 60, 67, 78, 84, 112

C
C qualification 94
CALL statement 100
cataloging the output file 91, 92
chaining of subcommands 64
character literal 56, 57, 123
CHECK 22
checking the storage types 107
close AID output file 91
close dump file 54
close PLAM library 120
COBOL special register 73, 83
COBOL statement 65
COBOL verbs, abbreviations for AID 19
coded program sequence, deviation from 71
COLUMBUS, statement name 66
command mode 119
command sequence 35, 87
compl-memref 40, 48, 60, 67
continuation address, %FIND 56
continuation address %JUMP 71
continuation 71, 72
continue program 30, 36, 68, 87, 99, 124
control of the output file 89, 123
control 30, 64, 68
control-area 31, 33
creating an AID output file 91
criterion 31, 32, 124
CSECT 79
current interrupt point 33, 90, 119, 124, 125, 127
current program 43

D
data element 108
data item 43, 74, 83, 102
data output 43, 89
data 43
dataname 15, 45, 59, 75, 83, 102, 109
declare global settings 22
default address interpretation 26
define a continuation address 71
define page header for SYSLST 123
define prequalification 93
definition in the source program 44
delete, %CONTROLn 96
delete event 97
delete subcmdname 97
delete test point 97
delete test-point 68
delete %CONTROLn 31
DELIM 22, 25
delimiter of AID output fields 22
display addresses 43
display lengths 43
display memory contents 43
doubleword boundary, search at 61
dump area 100
dump file 54
dynamic loading of LSD records 120

E
dead address 35
error message 62, 71
event table 86
event 81, 85, 86
execution condition 87
execution control 36, 87, 119, 124
Index

F
  F6 91
  feed to SYSLST 43
  feed-control 49
  figurative constants 17, 46, 76, 103, 110
  file 54, 91, 92
  filename 102
  find literal 56
  find-area 56, 57

G
  global data name, locally hidden 45, 75, 109
  global declaration, define 93

H
  halfword boundary, search at 61
  halt the program 119
  help texts 62
    output 62
  hexadecimal literal 56, 57
  hit address 56

I
  identifier 15, 76
  identifier 45
  In message number 63
  index 83
  index 15, 46, 76, 102, 110
  index-name 16
  indirect addressing 40, 48, 60, 67, 78, 84, 112
  individual command 54, 89
  info-target 62
  information on error messages 62
  information on the operation of AID 62
  input file 54
  instruction 37
  interpretation of indirect addresses 26
  interpretation of the hyphen 22
  interrupting the program 68, 119
  INVALID OPCODE 37
Index

K
K2 key 119, 124
keyword 26, 103
keyword 47, 77, 85, 112

L
LANG 22
LANG 25
length 43, 74, 79, 108
length function 48, 79
length modification 40, 48, 60, 67, 78, 84, 112
length selector 48, 78, 113
LIFO principle 64, 85
line feed 49
link name, assign 54, 91
link name F6 80
link 54, 91
LMS UPDR record 79
localization information, symbolic 47
LOW 22, 25
lowercase/uppercase 22
LSD names, abbreviations of COBOL verbs 19
LSD records 7, 14, 100, 120
dynamic loading 120

M
machine code level 43, 44, 73, 107
matching numeric values 107
medium-a-quantity 43, 62, 89, 90, 100
memory area 57
memory contents, modify 73, 107
memory references 11
message number AID0n 62
message number IDA0n 62
messages from AIDSYS 62
metasyntax 5
modifying memory contents 73, 107
monitor COBOL statements 32
monitor program addresses 64
monitoring 64
monitoring function 31, 33
monitoring statements 31
Index

N
number of lines per print page 123
number 37, 38, 124
numeric receiver 107
numeric transfer 107

O
object structure list 79
open AID output file 91
open dump file 54
open PLAM library 120
opening the output file 92
output
  current call hierarchy 100
  data areas 100
  file 49, 90, 103
  hardcopy 49, 90, 103
  program name 103
  terminal 49, 90, 103
output %DISASSEMBLE log 40
output %TRACE log 129
output commands 89
output literal 56
output medium 37, 43, 49, 62, 63, 89, 103, 124
output of hits 56
output of hits with %FIND 56
output type 44, 48
OV 22, 24
overlay 22

P
page counter for SYSLST 123
page feed 49
page-header 123
paragraph name 17
period 27, 33, 38, 45, 58, 65, 74, 82, 93, 101, 108, 121, 127
permissible combinations for %SET 114
PLAM library 100
  assign 120
  close 120
  open 120
pointer operator 47
PREPARE-FOR-JUMPS, test point 66
prequalification 27, 33, 38, 45, 58, 65, 74, 82, 93, 101, 108, 121, 127
prequalification 93
PROC qualification 13, 39, 45, 75, 94, 109, 128
PROG qualification 34, 58, 65, 83, 93, 101, 109, 128
program
  continue 99, 124
  start 99
program addresses, monitoring 64
program area to be monitored 33, 127
program counter 77, 112
program end 81
program error 81
program register 47
program start 124
program state
  alter 30
  change 99
program termination 81
programs with overlay structure 22

Q
qualification-a-lib 120

R
receiver 73, 74, 107, 108
register 108
REMOVE 36
REP file 80
REP record, generate 79
REP 22, 73, 79
retranslate memory contents 37
runtime control 68
runtime system 119

S
S qualification 13, 94, 101, 109, 128
search criterion 56
search string 56
search string length 56
search-criterion 57
section name 17
segment 33, 127
segmentation 33, 127
segmentname 75, 83, 109
segmentname 14, 34, 39, 58, 128
sender 107
sender 73, 74, 108
shared code 34
sharename 14, 128
sharename 39, 58, 66, 75, 83, 109
show-target 118
single command 62
source-reference 18, 35, 129
source-reference 40, 47, 66, 72, 77, 111
special register 83
special registers 17, 46, 59, 76, 84, 102, 110
specify global definitions 26
start %TRACE 124
start address 35
start address of the loaded operands 38
start program 30, 99
start 37, 38
statement 43, 72
statement name, COLUMBUS 66
statement-name 17, 34, 39, 46, 66, 72, 76, 111, 128
STOP message 119
storage type 44, 48
storage types, check 73
subcmd 31, 35, 64, 81, 87
subcommand 35, 57, 67, 68, 85, 93, 99, 119
subcommand chaining 68, 87
subcommand condition 35
subcommand name 35, 87
subcommand nesting 68, 87
subcommand termination 124
subprogram nesting 100
supervisor call (SVC) 81
symbolic constant 108
SYMCHARS 22, 24
SYSLST 49, 90, 103, 123
SYSOUT 56
system table 47

T
TALLY 16
target 96
target-cmd 89
test object 37
test point, PREPARE-FOR-JUMPS 66
test-point 64, 65, 66
trace area consisting of one statement 129
Index

trace-area 124
tracing 99, 124
transfer while retaining values 107
type modification 40, 43, 48, 60, 67, 78, 84, 112

U
update dialog 73, 108
uppercase/lowercase 22

V
verb 18

W
wildcard symbol 57
word boundary, search at 61
write-event 81
  overwriting 81

X
XS computers 26
## Contents

1 Preface .......................................................... 1
1.1 Target group ................................................. 2
1.2 Structure of the AID documentation ...................... 2
1.3 Changes made since AID V2.0A .......................... 3

2 Metasyntax .................................................... 5

3 Prerequisites for symbolic debugging ....................... 7
3.1 Compilation ................................................ 7
3.2 Linking, loading and starting ............................. 9
3.3 Commands at the start of a debugging session ............ 9

4 COBOL-specific addressing .................................. 11
4.1 Qualifications .............................................. 11
4.2 Symbolic memory references .............................. 14

5 AID commands ................................................ 21
%AID .......................................................... 22
%AINT ...................................................... 26
%BASE ...................................................... 28
%CONTINUE ................................................. 30
%CONTROLn ................................................. 31
%DISASSEMBLE .............................................. 37
%DISPLAY .................................................. 43
%DUMPFILE ................................................ 54
%FIND ....................................................... 56
%HELP ....................................................... 62
%INSERT .................................................... 64
%JUMP ....................................................... 71
%MOVE ...................................................... 73
%ON ........................................................ 81
%OUT ......................................................... 89
%OUTFILE .................................................. 91
%QUALIFY .................................................. 93
%REMOVE ................................................... 96
%RESUME ................................................... 99
%SDUMP .................................................... 100
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SET</td>
<td>107</td>
</tr>
<tr>
<td>%SHOW</td>
<td>117</td>
</tr>
<tr>
<td>%STOP</td>
<td>119</td>
</tr>
<tr>
<td>%SYMLIB</td>
<td>120</td>
</tr>
<tr>
<td>%TITLE</td>
<td>123</td>
</tr>
<tr>
<td>%TRACE</td>
<td>124</td>
</tr>
<tr>
<td><strong>6</strong> Sample application</td>
<td><strong>131</strong></td>
</tr>
<tr>
<td>6.1 Source listing</td>
<td><strong>131</strong></td>
</tr>
<tr>
<td>6.2 Contents of the input file</td>
<td><strong>133</strong></td>
</tr>
<tr>
<td>6.3 Test run</td>
<td><strong>133</strong></td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td><strong>143</strong></td>
</tr>
<tr>
<td><strong>References</strong></td>
<td><strong>153</strong></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td><strong>159</strong></td>
</tr>
</tbody>
</table>
AID V2.1A (BS2000)

Debugging of COBOL Programs
User Guide

Target group
COBOL programmers

Contents
- Description of the AID commands for symbolic debugging of COBOL programs
- Sample application

Applications
Testing of COBOL programs in interactive or batch mode

Edition: July 1994
File: AID_COB.PDF

BS2000 is a registered trademark of Siemens Nixdorf Informationssysteme AG.

Copyright © Siemens Nixdorf Informationssysteme AG, 1994. All rights reserved.
The reproduction, transmission, translation or exploitation of this document or its contents is not permitted without express written authority. Offenders will be liable for damages.
Delivery subject to availability; right of technical modifications reserved.
Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format …@ts.fujitsu.com.

The Internet pages of Fujitsu Technology Solutions are available at http://ts.fujitsu.com/...


Copyright Fujitsu Technology Solutions, 2009

Hinweise zum vorliegenden Dokument


Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.


Copyright Fujitsu Technology Solutions, 2009